Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

GESTÃO DE DADOS PARTILHADOS EM AMBIENTES DE COMPUTAÇÃO MÓVEL

NUNO MANUEL RIBEIRO PREGUIÇA

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
2003

Orientador: Professor José Legatheaux Martins, Professor Associado do Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa.

Agradecimentos

Para a realização deste trabalho contribuíram, directa ou indirectamente, diversas pessoas a que eu desejo agradecer.

Em particular, quero agradecer ao meu orientador, José Legatheaux Martins, pelo constante apoio, disponibilidade e conselhos dados. A liberdade que me concedeu para prosseguir o meu caminho e o seu encorajamento foram igualmente importantes na realização deste trabalho.

Ao Henrique João, Sérgio Duarte, Carlos Baquero, Luís Caires e João Seco, com os quais trabalhei em diversos projectos, quero agradecer pelas discussões tidas sobre tópicos relacionados com este trabalho.

Os alunos Miguel Cunha, Inês Vicente, Paulo Albuquerque e Filipe Leitão, merecem o meu agradecimento pela contribuição prestada no desenvolvimento e teste dos protótipos descritos nesta dissertação.

Quero igualmente agradecer ao Marc Shapiro, pela recepção e apoio prestado durante a minha estadia em Cambridge. A colaboração no projecto IceCube abriu-me novas oportunidades de investigação. Quero igualmente agradecer aos restantes elementos do grupo de sistemas distribuídos pelo apoio que me prestaram sempre que necessário.

Os meus pais, que sempre me incentivaram e apoiaram merecem um agradecimento especial. À Elsa, pelo incentivo e compreensão demonstrada durante este longo percurso. Aos meus amigos pelo incentivo prestado.

Finalmente, não posso deixar de agradecer ao Luís Caires, Luís Monteiro e José Legatheaux Martins pelo papel decisivo que tiveram na minha decisão de voltar à faculdade.

Este trabalho, assim como as deslocações a conferências e reuniões apenas foi possível com o apoio financeiro do Departamento de Informática, Centro de Informática e Tecnologia da Informação e Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, Fundação para a Ciência e Tecnologia através dos projectos Mobisnap (PRAXIS XXI/P/EEI/12188/1998) e Databricks (33924/99 FCT/MCT), Comissão Europeia e Fundo Social Europeu, Fundação Luso-Americana para o Desenvolvimento.

Sumário

A generalização da utilização de dispositivos computacionais portáteis e de sistemas de comunicação sem fios deu origem a um novo ambiente de computação distribuída que se designa por ambiente de computação móvel. Este ambiente de computação apresenta características próprias que obrigam à adaptação dos vários elementos de um sistema informático, entre os quais o sistema de gestão de dados.

Esta dissertação aborda os problemas relativos à gestão de dados partilhados em ambientes de computação móvel. Para lidar com estes problemas, propõe-se, como princípio fundamental, a utilização generalizada da informação semântica associada aos dados e às operações executadas pelos utilizadores.

Nesta dissertação identifica-se, ainda, um conjunto de princípios gerais que, explorando a informação semântica disponível, permitem a múltiplos utilizadores partilharem informação num ambiente de computação móvel. Estes princípios podem agrupar-se em dois grandes grupos. O primeiro grupo tem por objectivo permitir aos utilizadores acederem e modificarem os dados que necessitam em qualquer situação de conectividade recorrendo a: replicação optimista; replicação parcial secundária; e continuação da operação apesar das faltas na replicação – invocação cega. O segundo grupo tem por objectivo lidar e controlar as modificações concorrentes recorrendo a: prevenção de conflitos; propagação das operações e reconciliação dependente da situação; e tratamento da informação sobre a evolução dos dados.

Estes princípios gerais foram desenvolvidos e validados durante o desenho, implementação e teste de dois sistemas de gestão de dados com características distintas. O sistema DOORS é um repositório de objectos desenhado para simplificar a criação de aplicações cooperativas assíncronas. O sistema Mobsinap é um sistema de bases de dados desenhado para suportar o desenvolvimento de aplicações típicas de bases de dados relacionais num ambiente de computação móvel.

Para colocar em prática os princípios identificados, explorando a informação semântica disponível, foi necessário desenvolver um conjunto de técnicas de gestão de dados específicas para cada um dos sistemas anteriores. Estas técnicas são igualmente apresentadas nesta dissertação.

Abstract

The widespread use of mobile devices and wireless communications has created a new distributed computing environment, usually called mobile computing environment. Computing systems, including data management systems, must be adapted to address the new and intrinsic characteristics of these environments.

This dissertation studies data management for mobile computing, and, in particular, the problems involved in data sharing. To address these problems, we propose the extensive use of semantic information.

This dissertation also identifies a set of principles that explore the available semantic information to allow multiple users to share data in a mobile computing environment. We divide these principles into two groups. The first group aims at providing near-permanent read and write data availability in the presence of any connectivity conditions relying on: optimistic replication; partial caching; and operation during cache misses – blind invocation. The second group deals with concurrent operation relying on: conflict avoidance; log propagation and situation-specific reconciliation; and handling of awareness information.

These principles were developed and validated during the design, implementation and evaluation of two data management systems for mobile computing with different characteristics. DOORS is an object repository designed to simplify the development of new asynchronous groupware applications. Mobisnap is a relational database system designed to support typical database applications in a mobile computing environment.

The principles identified in this dissertation were applied, in each system, using different data management techniques that explore the available semantic information. These specific techniques are also presented in this dissertation.

Conteúdo

1 Introdução **1**

1.1 Motivação . . . 2

1.1.1 Aplicações cooperativas . . . 2

1.1.2 Outras aplicações . . . 2

1.2 Sistemas distribuídos de gestão de dados . . . 2

1.3 Visão geral e contribuições . . . 4

1.3.1 DOORS . . . 5

1.3.2 Mobisnap . . . 7

1.4 Índice . . . 8

2 Princípios gerais **9**

2.1 Introdução . . . 9

2.2 Motivação - algumas aplicações . . . 10

2.3 Princípios . . . 12

2.3.1 Replicação optimista . . . 12

2.3.2 Reconciliação dependente da situação . . . 13

2.3.3 Replicação baseada na propagação de operações . . . 14

2.3.4 Informação sobre a evolução dos dados e da actividade cooperativa . . . 16

2.3.5 Replicação secundária parcial . . . 17

2.3.6 Invocação cega . . . 19

2.3.7 Prevenção de conflitos . . . 20

2.3.8 Suporte para os programadores . . . 21

2.4 Sumário . . . 22

3 Apresentação do sistema DOORS **23**

3.1 Modelo geral . . . 24

3.1.1 Modelo de manipulação dos *coobjectos* . . . 25

3.1.2 Modelo de funcionamento do sistema 26
3.2 *Framework* de componentes: princípios gerais 28
3.2.1 Subobjectos 28
3.2.2 Funcionamento global 30
3.3 *Framework* de componentes: componentes 36
3.3.1 Atributos do sistema 36
3.3.2 Atributos 37
3.3.3 Registo 38
3.3.4 Reconciliação 39
3.3.5 *Awareness* 39
3.3.6 Adaptação 41
3.3.7 Cápsula 42
3.3.8 Gestor de subobjectos 42
3.3.9 Subobjectos 44
3.3.10 Definição de um *coobjecto*: exemplo 46
3.4 *Framework* de componentes: implementação 47
3.4.1 Pré-processador 48
3.4.2 Sumário 49

4 Descrição das funcionalidades principais do sistema DOORS 51

4.1 Reconciliação 51
4.1.1 Informação de ordenação das operações 52
4.1.2 Sem ordem 54
4.1.3 Ordem causal 55
4.1.4 Ordem total 55
4.1.5 Transformação de operações 59
4.1.6 Operações do sistema 60
4.2 Replicação secundária parcial 60
4.3 Invocação cega 61
4.3.1 Cópias de substituição 61
4.4 Integração de sessões síncronas 62
4.4.1 Diferentes operações para sessões síncronas e assíncronas 64
4.4.2 Discussão 66

5 Avaliação do modelo do sistema DOORS **69**

5.1 Editor multi-síncrono de documentos 70

5.1.1 *Coobjecto* documento estruturado 71

5.1.2 Replicação secundária parcial 74

5.1.3 Invocação cega 74

5.1.4 Edição assíncrona 75

5.1.5 Edição síncrona 75

5.2 Agenda partilhada 76

5.2.1 *Coobjecto* agenda 77

5.2.2 Replicação secundária parcial 79

5.2.3 Invocação cega 79

5.3 Outras aplicações 80

6 Núcleo do sistema DOORS **81**

6.1 Coobjectos 81

6.1.1 Criação de um *coobjecto* 82

6.1.2 Criação de um subobjecto 82

6.1.3 Remoção de um *coobjecto* 83

6.1.4 Remoção de um subobjecto 83

6.1.5 Versão dos *coobjectos* 83

6.1.6 Sistema de nomes 84

6.2 Servidores 85

6.2.1 Volumes e filiação 85

6.2.2 Sincronização epidémica dos servidores 89

6.2.3 Serviço de descoberta e disseminação de eventos 91

6.2.4 Interacção com os clientes 91

6.2.5 Recursos associados aos *coobjectos* 94

6.2.6 Suporte para múltiplas bases de dados 94

6.2.7 Serviço de *awareness* 95

6.3 Clientes 96

6.3.1 Replicação secundária parcial 96

6.3.2 Detalhes do funcionamento dos *coobjectos* 98

6.3.3 Execução síncrona e garantias de sessão 100

6.3.4 Serviços básicos 100

6.3.5 Comunicação entre clientes 101

6.3.6 Serviço de *awareness* 102
6.4 Sumário 102

7 Apresentação do sistema Mobisnap 103
7.1 Modelo geral 103
7.2 Arquitectura 107
7.2.1 Servidor 108
7.2.2 Cliente 109
7.2.3 Protótipo 110
7.3 Transacções móveis 111
7.3.1 Processamento 111
7.3.2 Processamento alternativo 112

8 Reservas 115
8.1 Tipos de reservas 115
8.1.1 Reservas *value-change* e *slot* 115
8.1.2 Reserva *value-lock* 116
8.1.3 Reservas *value-use* 116
8.1.4 Reservas *escrow* 117
8.1.5 Reservas *shared value-change* e *shared slot* 119
8.2 Concessão e garantia de respeito pelas reservas 119
8.3 Processamento das transacções móveis no cliente 122
8.3.1 Verificação da possibilidade de garantir uma transacção móvel 122
8.4 Processamento das transacções móveis no servidor 125
8.5 Exemplos 126

9 Avaliação do modelo básico do sistema Mobisnap 131
9.1 Aplicações 131
9.1.1 Suporte a uma força de vendas 131
9.1.2 Agenda partilhada 133
9.1.3 Sistema de reserva de bilhetes de comboio 134
9.1.4 Metodologia de concepção das transacções móveis 134
9.2 Reservas 134
9.2.1 Modelo das experiências 135
9.2.2 Previsão fiável 138
9.2.3 Previsão não fiável 143

10 Sistema de reconciliação SqlIceCube **149**

10.1 Modelo geral . . . 149

10.2 Relações estáticas . . . 151

10.2.1 Relações da aplicação . . . 151

10.2.2 Relações dos dados . . . 152

10.2.3 Transacções independentes . . . 152

10.3 Algoritmo de reconciliação . . . 153

10.3.1 Heurística . . . 153

10.3.2 Algoritmo básico . . . 154

10.3.3 Complexidade . . . 157

10.4 Optimização da reconciliação . . . 158

10.4.1 Partições . . . 158

10.4.2 Compressão de uma sequência de transacções . . . 159

10.5 Extracção automática de relações . . . 160

10.5.1 Extracção de informação . . . 161

10.5.2 Inferir relações . . . 162

10.5.3 Exemplos . . . 166

10.5.4 Extensões . . . 168

10.6 Observações finais . . . 172

11 Trabalho relacionado **173**

11.1 Replicação . . . 173

11.1.1 Modelos . . . 173

11.1.2 Arquitecturas e tipos de réplicas . . . 175

11.1.3 Unidade de replicação . . . 176

11.1.4 Replicação antecipada . . . 178

11.1.5 Invocação cega . . . 178

11.1.6 Propagação das modificações: propagar o quê? . . . 179

11.1.7 Detecção das modificações . . . 180

11.1.8 Propagação das modificações: propagar como? . . . 181

11.1.9 Reconciliação . . . 183

11.1.10 Prevenção de conflitos . . . 187

11.2 Informação sobre a evolução dos dados . . . 189

11.3 Integração de sessões síncronas . . . 190

12 Conclusões **193**

12.1 Sumário 193

12.2 Trabalho futuro 196

A DOORS **197**

A.1 Filiação 197

A.1.1 *Coobjecto* de filiação 197

A.1.2 Protocolo local de mudança de vista 200

A.1.3 Protocolo de entrada no grupo de replicadores de um volume 201

A.1.4 Protocolo de saída voluntária do grupo de replicadores de um volume 202

A.1.5 Operação de *disseminação e execução* 204

A.1.6 Protocolo de eliminação do identificador de um servidor removido 206

A.1.7 Protocolo de saída forçada do grupo de replicadores de um volume 207

A.1.8 Protocolo de execução dos blocos *só-uma-vez* em ordenações por verificação da estabilidade 211

A.2 Sincronização epidémica 212

A.2.1 Informação utilizada 213

A.2.2 Protocolo de propagação epidémica bilateral 213

A.2.3 Protocolo de propagação epidémica unilateral 218

A.2.4 Sincronização epidémica durante as mudanças na filiação 218

A.2.5 Disseminação de novas operações 220

B Mobisnap **221**

B.1 Transacções móveis:linguagem 221

Lista de Figuras

3.1 Arquitectura do sistema DOORS . . . 24
3.2 *Framework* de componentes DOORS. . . . 27
3.3 Interface do *coobjecto*. . . . 31

5.1 Dissertação representada como um documento estruturado. . . . 70
5.2 Organização de um documento estruturado em subobjectos. . . . 71
5.3 Edição de um documento LaTeX. . . . 73
5.4 Cópias de substituição na edição de um documento estruturado. . . . 75
5.5 Elementos de edição síncrona no editor multi-síncrono. . . . 76
5.6 Aplicação de agenda partilhado. . . . 77

6.1 API do sistema DOORS para manipulação dos *coobjectos*. . . . 82
6.2 Interface do servidor. . . . 92
6.3 API do sistema DOORS. . . . 96

7.1 Transacção móvel *nova encomenda* com duas alternativas . . . 104
7.2 Arquitectura do sistema Mobisnap. . . . 107

8.1 Transacção móvel *nova encomenda* . . . 118
8.2 Transacção móvel *reserva de bilhete* . . . 127
8.3 Transacção móvel *reserva de bilhete*: segunda alternativa . . . 129
8.4 Transacção móvel *nova marcação* numa agenda . . . 130

9.1 Transacção móvel *cancela encomenda* . . . 132
9.2 Transacção móvel *remove marcação* numa agenda . . . 133
9.3 Transacções aceites localmente (cenário *MOV:PEQ:BOM*) . . . 139
9.4 Transacções aceites localmente (cenário *MOV:GRD:BOM*) . . . 139
9.5 Transacções aceites localmente (cenário *DIS:PEQ:BOM*) . . . 140
9.6 Transacções aceites imediatamente no cliente ou no servidor (cenário *MOV:PEQ:BOM*) . 141

9.7 Transacções aceites imediatamente no cliente ou no servidor (cenário *DIS:PEQ:BOM*) . 141
9.8 Transacções aceites imediatamente no cliente ou no servidor (cenário *MOV:GRD:BOM*) 142
9.9 Mensagens enviadas na obtenção de novas reservas (cenários *MOV:PEQ:BOM* e *MOV:GRD:BOM*) 142
9.10 Transacções aceites localmente (cenário *MOV:PEQ:MAU*) 143
9.11 Transacções aceites imediatamente no cliente ou no servidor localmente (cenário *MOV:PEQ:MAU*) 144
9.12 Transacções aceites localmente sem obtenção inicial de reservas (cenário *MOV:PEQ:MAU*) 144
9.13 Transacções aceites localmente sem obtenção inicial de reservas (cenário *MOV:GRD:MAU*) 145
9.14 Transacções aceites imediatamente no cliente ou no servidor sem obtenção inicial de reservas (cenário *MOV:PEQ:MAU*) 145
9.15 Mensagens enviadas na obtenção de novas reservas (cenários *MOV:PEQ:MAU* e *MOV:GRD:MAU*) 146
9.16 Mensagens enviadas na obtenção de novas reservas sem obtenção inicial de reservas (cenários *MOV:PEQ:MAU* e *MOV:GRD:MAU*) 146

10.1 Ordenação de um conjunto de operações executadas concorrentemente. 150
10.2 Algoritmo de reconciliação (sem partições) 155
10.3 Algoritmo de criação de uma única solução. 156
10.4 Algoritmo de reconciliação (com partições). 157
10.5 Informação extraída de uma transacção móvel nova encomenda 163
10.6 Informação extraída da transacção móvel *cancela encomenda* (sem indirecções) 166
10.7 Informação extraída da transacção móvel *nova marcação* simples 167
10.8 Informação extraída da transacção móvel *cancela marcação* usando os detalhes da marcação 168
10.9 Divisão de uma transacção composta por uma sequência de instruções *if* 171
10.10 Divisão de uma transacção composta por uma conjunto de instruções *if* encadeadas . . . 172

A.1 Funções básicas e informação usada no protocolo de sincronização bilateral (simplificado). 215
A.2 Funções do protocolo de sincronização bilateral (simplificado). 216
A.3 Descrição da sincronização de um *coobjecto* usando o protocolo de sincronização bilateral. 217
A.4 Funções do protocolo de sincronização unilateral (simplificado). 219

Lista de Tabelas

8.1 Tabela de compatibilidade das reservas. 120
8.2 Reservas obtidas para garantir encomendas 127
8.3 Reservas obtidas para garantir reservas de bilhetes 128
8.4 Reservas obtidas para garantir reservas de bilhetes: segunda alternativa 128
8.5 Reservas obtidas para garantir novas marcações numa agenda 129

9.1 Parâmetros das experiências: duração e falhas. 136
9.2 Parâmetros das experiências: configuração do sistema simulado. 136
9.3 Parâmetros das experiências: exactidão das previsões. 136

Capítulo 1

Introdução

Os avanços tecnológicos produzidos na última década permitiram a generalização da utilização de computadores portáteis e de comunicações sem fios. É expectável que esta tendência se mantenha e acentue nos próximos anos, com um incremento do número e capacidade dos dispositivos móveis, uma melhoria das comunicações sem fios e a generalização da integração destes dois elementos. Em resultado da fusão destes desenvolvimentos tecnológicos e da viabilidade económica da sua generalização, criou-se um novo ambiente de computação, a que se chama genericamente (ambiente de) computação móvel.

A computação móvel apresenta características intrínsecas distintas das características dos sistemas distribuídos tradicionais [149, 7, 124]. Entre estas, podem destacar-se as restrições de disponibilidade energética, a conectividade de qualidade variável e os reduzidos recursos de *hardware* de, pelo menos, uma parte significativa dos dispositivos móveis (quando comparados com os computadores de secretária). Assim, os sistemas informáticos desenhados para estes ambientes têm de tomar em consideração e adaptar-se a essas características. Nesta dissertação aborda-se o problema da gestão de dados em ambientes de computação móvel.

A computação móvel apresenta, além das diferenças de hardware e conectividade, uma característica adicional potencialmente distintiva: os utilizadores têm acesso (quase) permanente aos dispositivos móveis. Ao nível da gestão de dados, esta característica sugere a necessidade de uma disponibilidade permanente dos dados e potencia o acesso concorrente a dados partilhados.

Esta dissertação apresenta soluções de gestão de dados partilhados para ambientes de computação móvel que tomam em consideração as características especiais deste novo tipo de ambiente de computação.

Este capítulo estabelece o contexto desta dissertação. Primeiro, apresenta-se um conjunto de aplicações para ambientes de computação móvel em que a partilha de dados é uma necessidade. De seguida, discutem-se as limitações das soluções tradicionais no suporte às aplicações anteriores. Finalmente, apresentam-se brevemente as soluções propostas nesta dissertação, realçando as suas contribuições.

1.1 Motivação

Um sistema distribuído de gestão de dados armazena informação que pode ser acedida a partir de diferentes computadores interligados por uma rede de comunicações. Esta informação, ou, pelo menos, uma parte desta informação é acedida e modificada por mais do que um utilizador. De seguida apresentam-se alguns exemplos desta situação.

1.1.1 Aplicações cooperativas

Numa actividade cooperativa, um conjunto de utilizadores actua em conjunto para alcançar um objectivo comum. Nas aplicações de suporte às actividade cooperativas (geralmente designadas por *groupware*), o estado da actividade cooperativa é geralmente mantida sob a forma de dados partilhados. Os utilizadores produzem as suas contribuições para a actividade cooperativa modificando os dados partilhados.

Nas aplicações de trabalho em grupo assíncrono, os utilizadores produzem as suas modificações em diferentes momentos ou sem conhecimento imediato das modificações produzidas pelos outros utilizadores. Assim, estas aplicações apresentam um cenário paradigmático da necessidade de múltiplos utilizadores acederem e modificarem dados partilhados.

As seguintes aplicações são exemplos típicos de aplicações de suporte ao trabalho cooperativo assíncrono [11, 47, 144]. Num sistema de conferência assíncrono, múltiplos utilizadores cooperam na discussão de vários assuntos usando um espaço de mensagens partilhado. Num editor cooperativo de documento, vários utilizadores cooperam na criação de um documento. Numa agenda partilhada, múltiplos utilizadores coordenam as suas marcações.

1.1.2 Outras aplicações

A necessidade de múltiplos utilizadores acederem a dados partilhados não é uma característica apenas das aplicações de trabalho em grupo. Muitas outras aplicações e serviços apresentam este requisito.

Por exemplo, numa aplicação de suporte a uma força de vendas móvel, os vendedores necessitam de aceder e modificar a informação sobre os clientes e sobre os produtos disponibilizados pela empresa, para além das encomendas realizadas. No serviço de reservas de lugares de avião, múltiplos clientes necessitam de aceder à informação sobre os lugares disponíveis e introduzir e cancelar reservas.

1.2 Sistemas distribuídos de gestão de dados

As aplicações descritas anteriormente exemplificam a necessidade de aceder e modificar dados partilhados. Para que os utilizadores possam executar as suas acções, o sistema de gestão de dados deve fornecer uma elevada disponibilidade de serviço, permitindo, idealmente, que qualquer utilizador aceda aos dados

partilhados para leitura e escrita em qualquer momento. Num sistema distribuído, e, em particular, num ambiente de computação móvel, é impossível garantir o acesso permanente aos dados a partir de um único servidor [30] devido às falhas nas comunicações e nos servidores. Assim, o sistema necessita de recorrer à replicação dos dados para fornecer uma elevada disponibilidade.

Num sistema distribuído de grande-escala, que inclua computadores interligados por redes de longa-distância (*WAN*), pode ser impossível coordenar os acessos executados por múltiplos utilizadores devido à impossibilidade de contactar alguns computadores. Num ambiente de computação móvel, esta impossibilidade é reforçada pelas seguintes características. Primeiro, a inexistência ou reduzida qualidade da conectividade em algumas zonas e períodos de tempo. Segundo, a necessidade de economizar energia pode levar à desconexão dos dispositivos móveis ou dos subsistemas de comunicação. Terceiro, razões económicas podem condicionar a conectividade em alguns períodos de tempo.

A coordenação de múltiplos utilizadores, mesmo quando possível, pode ser indesejável devido à elevada latência induzida por essa coordenação ou devido às características da aplicação. Assim, os esquemas de replicação tradicionais [17], que coordenam os acessos de múltiplos utilizadores, são inapropriados, porque incorrem numa elevada latência e/ou reduzida disponibilidade [30]. Os sistemas de gestão de dados para ambientes móveis devem, alternativamente, recorrer a um modelo de replicação optimista, que lhes permita fornecer uma elevada disponibilidade de leitura e escrita.

Num sistema de gestão de dados baseado em replicação optimista, permite-se que cada utilizador modifique uma cópia dos dados sem coordenação prévia. Como vários utilizadores podem modificar concorrentemente os dados, levando à divergência das várias réplicas, é necessário que o sistema de gestão de dados inclua um mecanismo de reconciliação de réplicas que permita lidar com a situação. Este mecanismo de reconciliação deve, em geral, garantir a convergência final dos dados, i.e., deve garantir que após todas as modificações serem reflectidas em todas as réplicas, o estado de todas as réplicas é idêntico. Outra propriedade a satisfazer é a preservação das intenções dos utilizadores, i.e., o efeito de uma modificação deve reflectir a intenção do utilizador quando este a executou.

Várias soluções de reconciliação foram propostas e utilizadas em sistemas de gestão de dados [147, 124]. Em geral, a utilização de informação semântica associada aos dados e às operações produzidas para modificar os dados é a chave para o desenvolvimento de uma boa solução. No entanto, nenhuma solução proposta parece ser ideal em todas as situações. Assim, a reconciliação continua a ser uma área problemática no desenvolvimento de um sistema de gestão de dados baseado em replicação optimista.

Muitos sistemas de gestão de dados baseados em replicação optimista permitem a definição de soluções específicas de reconciliação (baseadas numa dada estratégia adoptada). No entanto, apesar de em alguns casos ser possível reutilizar a mesma aproximação em problemas diferentes (com pequenas adaptações), não existe geralmente suporte para a reutilização de soluções pré-definidas. Assim, pa-

rece desejável a existência de um repositório de soluções pré-definidas que possam ser utilizadas em diferentes situações, de forma a simplificar o desenvolvimento de novas soluções de gestão de dados.

A necessidade de reconciliar as modificações produzidas concorrentemente por vários utilizadores leva à impossibilidade de garantir o resultado ou efeito final de uma modificação aquando da sua execução pelo utilizador. Apenas após o processo de reconciliação, o resultado e efeito final da modificação é determinado. No entanto, em algumas aplicações, o conhecimento imediato do resultado de uma operação é desejável ou mesmo imprescindível. As soluções propostas anteriormente para lidar com este problema [111, 168] apresentam algumas limitações, entre as quais se destaca a aplicabilidade a apenas um limitado conjunto de tipos de dados. Adicionalmente, a não integração com um modelo de reconciliação genérico impede a definição de um modelo global de gestão de dados replicados.

Nas actividades cooperativas, o conhecimento do modo como os dados partilhados evoluem foi identificado como importante para o sucesso das contribuições produzidas [43, 65, 99]. No entanto, os sistemas de gestão de dados tendem a ignorar este problema, levando as aplicações a desenvolver soluções próprias e sem suporte adequado do sistema. Um caso particular deste problema é o conhecimento do resultado final das operações produzidas pelo próprio utilizador — note-se que o processo de reconciliação pode terminar num momento em que o utilizador já não está a utilizar a aplicação/sistema. Apesar do modo como esta informação é produzida, tratada e transmitida aos utilizadores dever depender da aplicação ou situação, é desejável que o sistema integre suporte para o seu tratamento.

Outra limitação comum nos sistemas distribuídos de gestão de dados prende-se com o tratamento de situações de faltas na replicação. Em geral, qualquer acesso a um objecto do qual não se possui um cópia local é impossibilitado. No entanto, em algumas situações, um utilizador pode produzir contribuições positivas nestas situações. Por exemplo, é possível requisitar a marcação de uma reunião numa agenda apesar de ser impossível aceder ao seu conteúdo. Desta forma, o sistema de gestão de dados deve incluir suporte para lidar com estas situações.

1.3 Visão geral e contribuições

Esta dissertação apresenta um conjunto de contribuições sobre métodos e estratégias para lidar com o problema da gestão de dados partilhados em ambientes de computação móvel e formula um conjunto de princípios gerais desenvolvidos e validados durante o desenho, implementação e teste de dois sistema de gestão de dados. O DOORS é um repositório de objectos desenhado com o objectivo de suportar a criação de aplicações cooperativas assíncronas. O Mobsinap é um sistema de bases de dados desenhado para suportar o desenvolvimento de aplicações típicas de bases de dados relacionais num ambiente de computação móvel.

Apesar das diferenças entre os dois sistemas, eles são construídos com base num conjunto de prin-

cípios comuns usados para lidar com as características dos ambientes de computação móvel. Entre os princípios identificados, podem destacar-se os seguintes:

- A utilização de um modelo de replicação optimista para aumentar a disponibilidade de leitura e escrita dos dados. Estendendo o modelo tradicional de replicação optimista, permite-se que, em algumas situações, os utilizadores submetam operações sobre dados não replicados localmente.

- A replicação parcial dos dados nos clientes como forma de lidar com os limitados recursos disponíveis (em termos de armazenamento local e de comunicações) em alguns clientes móveis.

- A utilização de técnicas de reconciliação que exploram a informação semântica disponível na resolução de conflitos. A propagação das modificações como sequências de operações, permitindo maximizar a informação semântica disponível.

- A utilização de técnicas de prevenção de conflitos para obter, de forma independente, o resultado final das operações submetidas em clientes móveis (e desconectados).

- A integração de mecanismos para tratamento da informação sobre a evolução dos dados partilhados de forma a fornecer, aos utilizadores, informação sobre as modificações executadas concorrentemente e sobre o resultado ou efeito final das suas próprias modificações. Como se referiu anteriormente, este aspecto é geralmente negligenciado nos sistema de gestão de dados.

Estes princípios, assim como as técnicas propostas para os colocar em prática, adoptam como aproximação elementar a utilização da informação semântica disponível para criar soluções adaptadas às características do ambiente de computação móvel e das aplicações que se pretende desenvolver. De seguida, resumem-se as características de cada sistema, destacando as contribuições desta dissertação em cada uma das aproximações.

1.3.1 DOORS

O sistema DOORS é um repositório de objectos baseado numa arquitectura cliente estendido/servidor replicado. O sistema gere objectos estruturados de acordo com o *framework*[1] de componentes DOORS, a que se chamam *coobjectos*. Os *coobjectos* são entidades de granularidade elevada que representam tipos de dados complexos — por exemplo, um documento estruturado ou uma agenda. Um *coobjecto* pode ser decomposto num conjunto de elementos individuais, os subobjectos, cada um composto por um

[1] Em contextos semelhantes, o termo *framework* tem sido traduzido, entre outros, por modelo ou organização. No entanto, por se considerar que a utilização destas traduções não incluem o sentido completo dado à utilização do termo original, que inclui não apenas a definição do conjunto de componentes usados mas também o modo como se organizam e actuam conjuntamente para implementar uma dada funcionalidade, decidiu-se utilizar o termo inglês em itálico.

grafo de objectos elementares — por exemplo, um documento estruturado é composto por um conjunto de secções. Os *coobjectos* são agrupados em conjuntos de *coobjectos* relacionados (representando um espaço de trabalho partilhado ou os dados de uma actividade cooperativa).

Para fornecer uma elevada disponibilidade combinam-se as seguintes técnicas. Os servidores replicam conjuntos completos de *coobjectos* para mascarar falhas nos servidores e no sistema de comunicações. Os clientes mantêm cópias parciais de um subconjunto de *coobjectos* para permitir o funcionamento durante os períodos de desconexão. O acesso aos dados é baseado numa aproximação optimista em que qualquer cliente pode modificar os *coobjectos*: as modificações são propagadas sob a forma de operações (invocação de métodos definidos na interface dos subobjectos).

O núcleo do sistema apenas inclui os serviços mínimos de gestão de dados, delegando nos *coobjectos* o tratamento dos aspectos específicos relacionados com a partilha dos dados. Para tratar destes problemas, o *framework* de componentes DOORS decompõe o funcionamento de um *coobjecto* em vários componentes, cada um lidando com um aspecto relacionados com a partilha dos dados. Os principais aspectos considerados são a reconciliação de modificações concorrentes e o tratamento da informação sobre a evolução dos dados. Esta aproximação permite utilizar, em diferentes *coobjectos*, diferentes estratégias de resolução dos vários problemas e não apenas adaptar uma única estratégia (como é usual nos sistemas de gestão de dados). Por exemplo, relativamente à reconciliação, é possível usar uma estratégia simples baseada na ordenação total das operações, ou soluções mais complexas, como a utilização da transformação de operações [46]. Esta aproximação permite, ainda, reutilizar as soluções definidas, em múltiplos *coobjectos*, assim simplificando a tarefa dos programadores. A definição do *framework* de componentes com as propriedades descritas é uma das contribuições principais do sistema DOORS.

O sistema DOORS inclui um mecanismo de *invocação cega* que permite a continuação da operação na presença de faltas na replicação. Além de permitir às aplicações submeterem operações sobre subobjectos dos quais não se dispõe de uma cópia local, permite, ainda, verificar o resultado provisório da sua execução através da criação de cópias de substituição. Este mecanismo é outra das contribuições originais do sistema DOORS, permitindo minorar os problemas decorrentes das faltas na replicação secundária.

O sistema DOORS propõe também um modelo de integração de sessões de trabalho cooperativo síncrono de curta duração, no âmbito de sessões assíncronas de longa duração. Este modelo pode ser decomposto em duas partes. Primeiro, define-se um mecanismo, integrado no *framework* de componentes, que permite manter um conjunto de cópias de um *coobjecto* sincronamente sincronizadas. Segundo, propõe-se a conversão das operações síncronas em assíncronas para permitir a utilização de diferentes operações nos dois modos de interacção (quando necessário). Esta conversão pode ser efectuada internamente pelo *coobjecto* ou externamente pela aplicação. Esta aproximação permite que se utilizem as

técnicas mais adequadas em cada modo de interacção, incluindo a utilização de operações com diferentes granularidades, diferentes técnicas de reconciliação e tratamento da informação sobre a evolução dos dados. O sistema DOORS é o primeiro que apresenta uma solução de integração das sessões síncronas e assíncronas com estas características.

1.3.2 Mobisnap

O Mobisnap é um sistema de bases de dados relacionais baseado numa arquitectura cliente/servidor. O servidor único mantém o estado oficial da base de dados. Os clientes replicam cópias parciais da base de dados para suportar a operação durante os períodos de desconexão.

As aplicações, que executam nos clientes, modificam o estado da base de dados submetendo pequenos programas escritos em PL/SQL [112], a que se chamam transacções móveis. Durante a operação desconectada, as transacções móveis são executadas provisoriamente no cliente. Mais tarde, as transacções são propagadas para o servidor onde o seu resultado final é determinado reexecutando o programa da transacção móvel. Ao contrário da execução e validação de transacções baseada nos conjuntos de leitura e escrita, a execução do programa das transacções móveis no servidor permite a definição de regras de detecção e resolução de conflitos que exploram a semântica das operações.

Para permitir determinar o resultado de uma transacção no cliente de forma independente (por exemplo, durante os períodos de desconexão), o sistema Mobisnap inclui um mecanismo de reservas. Uma reserva fornece, dependendo do seu tipo, uma promessa temporária em relação ao estado da base de dados. Um cliente pode obter um conjunto de reservas antes de se desconectar. Quando uma aplicação submete uma transacção móvel, o sistema verifica, de forma transparente, se as reservas que o cliente possui são suficientes para garantir o seu resultado final. Se o cliente possuir reservas suficientes, o resultado local pode ser considerado final porque as reservas garantem a inexistência de conflitos aquando da execução da transacção móvel no servidor.

O mecanismo de reservas do sistema Mobisnap apresenta as seguintes contribuições. Primeiro, inclui vários tipos de reservas (ao contrário de propostas anteriores que apenas definem um único tipo [111]). Como se mostra nesta dissertação, esta característica é fundamental para garantir o resultado de muitas transacções móveis. Segundo, o sistema verifica de forma transparente a possibilidade de usar as reservas disponíveis para garantir o resultado de uma transacção móvel. Assim, não é necessário usar funções especiais para manipular os elementos de dados reservados e qualquer transacção pode explorar todas as reservas disponíveis (e não apenas as que a transacção conhece). Terceiro, o sistema combina a implementação do mecanismo de reservas com o modelo de execução das transacções móveis num sistema baseado em SQL.

O sistema Mobisnap inclui também um novo sistema de reconciliação de transacções móveis, o sis-

tema SqlIceCube. Este sistema procura maximizar o conjunto de transacções que podem ser executadas com sucesso, usando para tal um conjunto de relações semânticas definidas entre cada par de transacções. O sistema SqlIceCube estende a aproximação proposta no sistema IceCube [85] e aplica-a no âmbito dum sistema de bases de dados relacionais. Além das adaptações necessárias, que incluem a definição de um novo conjunto de relações para exprimir a semântica das operações de uma aplicação em problemas de reconciliação, o sistema SqlIceCube inclui uma contribuição original significativa: a extracção automática das relações semânticas a partir do código das transacções móveis. O SqlIceCube é o primeiro a utilizar a análise estática das operações para extrair automaticamente a informação semântica necessária para o sistema de reconciliação.

O sistema Mobisnap, ao ser implementado como uma camada de sistema intermédia (*middleware*), representa uma aproximação evolutiva em relação à mobilidade (em vez de uma aproximação revolucionária). Os novos mecanismos — transacções móveis, reservas e reconciliação de um conjunto de transacções — podem ser usados pelos clientes que executam no sistema Mobisnap. No entanto, os clientes legados podem continuar a aceder ao servidor de bases de dados directamente, sem modificações.

1.4 Índice

O resto desta dissertação está organizada da seguinte forma. No capítulo 2 discutem-se os princípios gerais adoptados nas soluções de gestão de dados apresentadas nesta dissertação.

Nos capítulos seguintes apresenta-se de forma detalhada o sistema DOORS. O capítulo 3 apresenta o modelo geral de funcionamento do sistema e o *framework* de componentes DOORS. O capítulo 4 discute as características fundamentais do sistema DOORS, incluindo a reconciliação, replicação secundária parcial, invocação cega e integração de actividades síncronas e assíncronas. O capítulo 5 apresentae uma avaliação qualitativa do modelo do sistema com base num conjunto de aplicações implementadas. O capítulo 6 detalha a arquitectura do sistema e descreve o protótipo implementado para verificar a exequibilidade do modelo proposto.

Nos capítulos seguintes detalha-se o sistema Mobisnap. O capítulo 7 descreve o modelo geral de funcionamento do sistema, incluindo o funcionamento básico das transacções móveis. O capítulo 8 detalha o mecanismo de reservas, incluindo a sua interacção com o processamento das transacções móveis. O capítulo 9 apresenta uma avaliação qualitativa dos mecanismos básicos do sistema, com base num conjunto de aplicações. Apresenta igualmente uma avaliação quantitativa do mecanismo de reservas no âmbito de uma aplicação de suporte a uma força de vendas móvel. O capítulo 10 descreve o sistema genérico de reconciliação para transacções móveis usado no sistema Mobisnap: o sistema SqlIceCube.

No capítulo 11 discute-se o trabalho relacionado e no capítulo 12 apresentam-se as conclusões e indicam-se algumas direcções para trabalho futuro.

Capítulo 2

Princípios gerais

No capítulo anterior apresentaram-se algumas das possíveis motivações para aceder a dados partilhados em situações de desconexão. Neste capítulo discutem-se os requisitos necessários para permitir esse acesso e os princípios fundamentais adoptados para responder a esses requisitos. Estes princípios constituem a base das soluções apresentadas nesta dissertação. A forma como estes princípios são adaptados às características específicas é detalhada nos capítulos seguintes.

2.1 Introdução

As soluções apresentadas nesta dissertação assumem a existência dum sistema informático com as seguintes propriedades. O sistema é composto por um conjunto de computadores heterogéneos com características próprias (processador, memória, dispositivos de comunicação, etc). Os computadores conseguem comunicar entre si através da troca de mensagens transmitidas através de uma rede de comunicação de dados.

Assume-se que a rede de comunicação de dados tem características semelhantes à *Internet*. Assim, é impossível determinar o tempo de propagação de uma mensagem entre dois computadores, t_p, mas este valor é inferior a uma dada constante caso a mensagem seja recebida no destino[1] (i.e., $t_p \in]0, t_{p_{max}}]$ para uma mensagem entregue e $t_p = \infty$ para uma mensagem não entregue). Desta forma, as mensagens enviadas entre dois computadores podem não chegar ao destino, ou, quando chegam, podem chegar fora de ordem e podem-se duplicar.

Diz-se que um computador i está desconectado em relação a outro computador j quando i não consegue enviar mensagens para j, ou seja, o tempo de propagação de todas as mensagens enviadas de i para

[1] Estas características são semelhantes às de um sistema assíncrono [102], com a diferença que existe um limite superior para o tempo de propagação duma mensagem assíncrona. Esta propriedade pode ser obtida facilmente a partir dum sistema assíncrono se os relógios dos computadores estiverem, pelo menos, fracamente sincronizados: basta ignorar todas as mensagens cujo tempo de propagação seja superior ao tempo máximo de propagação considerado

j é $t_p = \infty$. Diz-se simplesmente que está desconectado quando está desconectado em relação a todos os outros computadores do sistema. A desconexão de um computador pode ser voluntária (desconexão dos dispositivos de comunicação) ou causada por uma falha (falha na rede).

O sistema fornece um serviço de gestão de dados. O serviço de gestão de dados fornece dois tipos de operação: leitura e escrita. Uma operação de leitura permite obter o valor actual dos dados. Diz-se que os dados foram lidos quando foi executada uma operação de leitura. Uma operação de escrita permite modificar o valor dos dados. Diz-se que os dados foram escritos ou modificados quando foi executada uma operação de escrita. Diz-se, genericamente, que os dados foram acedidos quando foi executada uma operação de leitura ou escrita.

Os utilizadores usam este serviço (através das aplicações que utilizam) para armazenar a informação que necessitam para as suas actividades. Diz-se que um utilizador submeteu ou executou uma operação de leitura (escrita), quando a aplicação que utiliza invoca uma operação de leitura (escrita) no sistema de gestão de dados.

A informação armazenada no sistema de gestão de dados está organizada de acordo com o modelo de dados suportado pelo sistema. No entanto, independentemente do modelo de dados utilizado, pode identificar-se que a informação é composta por um conjunto de unidades de informação indivisíveis para o sistema (do ponto de vista da manipulação dos dados). A estas unidades indivisíveis chamam-se genericamente objectos ou elementos de dados. Por exemplo, para dados organizados segundo o modelo relacional, o valor de uma coluna de uma linha (registo) de uma tabela é uma unidade indivisível que se denomina por objecto ou elemento de dados.

O sistema de gestão de dados pode manter várias cópias dos dados. A cada uma das cópias dá-se o nome de réplica. De forma informal, define-se a consistência entre duas réplicas como o número de modificações que as réplicas sofreram desde a última vez que foram sincronizadas. Duas réplicas são fortemente consistentes quando o número de modificações desde a última sincronização tende a ser nulo, i.e., o algoritmo de gestão das réplicas procura que todas as réplicas mantenham sempre o mesmo valor. Duas réplicas são fracamente consistentes quando o número de modificações desde a última sincronização tende a ser não nulo.

No resto deste capítulo discutem-se as propriedades fundamentais que um serviço de gestão de dados para permitir o acesso aos dados independentemente da conectividade existente.

2.2 Motivação - algumas aplicações

Nesta secção vamos descrever brevemente algumas aplicações que servem de motivação para os princípios enunciados neste capítulo.

Num sistema de conferência assíncrono[2] existe um espaço de mensagens partilhado entre vários utilizadores (semelhante ao serviço de *news* da *Internet*). Estas mensagens estão organizadas em temas de discussão. Um utilizador pode inserir uma mensagem como resposta ou adenda a outra mensagem ou iniciar um novo tema de discussão. Um sistema deste tipo pode funcionar mantendo várias cópias do espaço de mensagens partilhado. Os utilizadores devem poder ler as mensagens conhecidas na réplica que estão a aceder e inserir novas mensagens. Apesar de não ser necessário que todas as réplicas apresentem o mesmo estado, é conveniente que as mensagens sejam apresentadas de forma coerente, respeitando, pelo menos, as dependências entre mensagens (i.e., uma resposta a uma mensagem não deve aparecer antes da mensagem original).

Uma agenda partilhada permite coordenar as marcações executadas por múltiplos utilizadores. Existem vários cenários em que uma agenda partilhada pode ser útil, entre os quais: uma agenda pessoal actualizada por mais do que uma pessoa (por exemplo, o próprio e um secretário); uma agenda para um recurso partilhado (por exemplo, uma sala de reunião). Numa agenda partilhada, múltiplos utilizadores podem inserir novas marcações e remover marcações existentes (sendo estas operações sujeitas, obviamente, a restrições de controlo de acessos). De preferência, os utilizadores devem ser autorizados a modificar a agenda de forma independente. Estas novas marcações devem ser consideradas provisórias [161] até poderem ser confirmadas através de alguma forma automática de consenso global. Quando o solicitem, os utilizadores devem receber a confirmação das suas marcações.

Um editor cooperativo assíncrono permite a um grupo de utilizadores editar conjuntamente um documento (por exemplo, um artigo). Os vários utilizadores podem modificar o documento independentemente[3] (por exemplo, dois utilizadores podem modificar diferentes secções). As modificações concorrentes devem ser unificadas tendo em consideração todas as modificações produzidas (por exemplo, o documento final deve incluir as novas versões das várias secções). Um aspecto importante a considerar é que a consistência sintáctica [42] deve ser mantida mesmo quando se consideram conflitos semânticos, i.e., o serviço de gestão de dados deve manter os dados estruturalmente consistentes de forma a permitir que a actividade dos utilizadores continue (por exemplo, se dois utilizadores modificam a mesma secção, duas versões dessa secção devem ser criadas e mantidas).

Um sistema de reserva de lugares para comboios mantém informação sobre um conjunto de comboios e respectiva reserva de lugares. Os utilizadores podem consultar as alternativas existentes e inserir uma nova reserva ou remover uma reserva existente. As operações de consulta devem estar disponíveis

[2] Assíncrono é aqui tomado no sentido informal, em que o tempo que medeia entre a emissão e a recepção das mensagens pode ser significativo, na ordem dos minutos, horas ou mesmo dias.

[3] Para que um trabalho cooperativo seja bem sucedido deve existir alguma coordenação entre os elementos do grupo (neste exemplo, cada utilizador deve saber que secções deve modificar). O problema da coordenação entre os elementos dum grupo cooperativo está fora do âmbito desta dissertação mas foi abordado no âmbito do projecto DAgora [40, 39].

sempre, ainda que alguns dados (por exemplo, o número de lugares disponíveis) possam estar desactualizados. Relativamente às operações de inserção e remoção de reservas é conveniente obter imediatamente o seu resultado final. Para tal, pode, por exemplo, executar-se a operação na réplica "oficial" dos dados. Quando não é possível obter o resultado de imediato, deve ser possível pedir ao sistema para processar a operação de forma diferida — neste caso, o utilizador deve ser notificado do resultado do seu pedido quando ele é definitivamente executado.

Um sistema de suporte a uma força de vendas móvel mantém informação sobre os produtos disponíveis para venda. As operações disponibilizadas e as características de funcionamento esperadas são muito semelhantes às do sistema de reserva de lugares. No entanto, existe uma diferença que deve ser tida em consideração. Enquanto no sistema de reserva de lugares, cada lugar pode ser identificado univocamente, num sistema de venda de bens fungíveis (por exemplo, CDs), todos os elementos do mesmo tipo são indistinguíveis.

2.3 Princípios

No decurso do trabalho que conduziu a esta dissertação, identificaram-se alguns princípios que permitem criar boas soluções de gestão de dados para ambientes de computação móvel. Estes princípios, discutidos nesta secção, foram utilizados na criação das soluções apresentadas nesta dissertação.

Um princípio geral explorado nas soluções apresentadas nesta dissertação consiste na utilização da informação semântica associada aos dados e às operações executadas pelos utilizadores. Este princípio geral é transversal a todos os princípios identificados e permite criar soluções especialmente adaptadas a cada problema, como se discute na apresentação dos vários princípios.

2.3.1 Replicação optimista

Num sistema distribuído é impossível garantir a permanente acessibilidade armazenando os dados apenas num computador [30]. Esta impossibilidade tem múltiplas causas. Primeiro, pode existir uma real impossibilidade física, devido à ausência de conectividade ou falhas na rede ou ainda à falha do servidor. Segundo, pode ser inconveniente efectuar a comunicação devido a razões económicas ou problemas de bateria em dispositivos móveis. Por fim, existem situações em que esse acesso pode ser inapropriado, devido à excessiva latência das comunicações, excesso de carga nos servidores durante picos de actividade ou mesmo devido ao modelo das aplicações (por exemplo, num editor cooperativo assíncrono o utilizador deve produzir as suas contribuições de forma independente [116]).

Para fornecer uma elevada disponibilidade de serviço é usual recorrer à replicação dos dados. A existência de várias cópias dos dados, apesar de eliminar o ponto central de falha, não é suficiente para

garantir a disponibilidade dos dados em todas as situações (por exemplo, em situações de desconexão). Para ultrapassar situações de desconexão (com todas as réplicas) é necessário manter cópias locais dos dados. Nestas condições, o serviço é assegurado recorrendo às cópias locais.

O serviço de gestão de dados fornece dois tipos de operação: leitura e escrita. Embora existam aplicações em que apenas é necessário ler os dados, num número significativo de aplicações é necessário permitir aos utilizadores modificarem os dados. Assim, a disponibilidade do serviço de gestão de dados deve considerar não só a possibilidade de ler os dados, mas também de os modificar (independentemente da conectividade existente).

Numa situação em que a conectividade não é total e em que existem computadores desconectados durante longos períodos, os esquemas que garantem uma consistência forte [37] (por exemplo, baseados na utilização de trincos (*locks*)) apresentam uma baixa taxa de disponibilidade para escrita [30]. Em particular, torna-se impossível garantir uma consistência forte para as réplicas mantidas num computador desconectado sem impedir que os dados possam ser modificados. Por um lado, as modificações executadas após o momento da desconexão por outros computadores são desconhecidas. Por outro lado, as modificações executadas localmente não podem ser propagadas para os outros computadores. Assim, para permitir a continuidade do serviço em situações de desconexão, é necessário recorrer a cópias fracamente consistente, permitindo aos utilizador ler e modificar qualquer réplica[4].

Quando existe conectividade entre os computadores que mantêm réplicas dos dados, podem existir condições que tornam aceitável ou recomendável a fraca consistência das réplicas. Neste caso é possível evitar os custos da sincronização entre várias réplicas para modificar os dados e utilizar comunicação retardada (*lazy*) para propagar as modificações.

2.3.2 Reconciliação dependente da situação

A utilização de um modelo de replicação optimista com uma política de acesso aos dados *lê qualquer/escreve qualquer* réplica permite a execução concorrente de modificações sobre um mesmo objecto. Estas modificações, efectuadas em vários fluxos de actividade [41], podem originar a divergência das réplicas dos objectos. Para lidar com este problema, o serviço de gestão de dados deve possuir um mecanismo de reconciliação.

Dos exemplos apresentados na secção 2.2 é possível verificar que existem diferentes formas de tratar as modificações concorrentes e que as propriedades desejáveis da reconciliação de várias réplicas variam para diferentes aplicações. Por exemplo, tanto no sistema de conferência assíncrono como na agenda partilhada, as várias réplicas podem ser actualizadas aplicando as modificações efectuadas por uma or-

[4]Note-se que a consistência fraca entre as réplicas de um sistema não impede que possa existir uma réplica que mantenha o estado "oficial" dos dados, como se discute na secção 4.1.

dem apropriada. No sistema de conferência podem inserir-se as novas mensagens usando uma ordem causal porque não é necessário obter um estado comum (mas apenas manter as dependências entre as mensagens). Numa agenda partilhada é necessário usar uma ordem total para garantir a convergência das várias réplicas da agenda. Finalmente, no exemplo do editor cooperativo, a simples execução das operações por uma dada ordem não é suficiente porque, no caso de existirem modificações conflituantes, se pretendem manter as várias versões.

Muitos algoritmos foram propostos para lidar com as modificações concorrentes. Entre estes, podem referir-se como exemplo os algoritmos baseados na técnica *desfazer-refazer* (*undo-redo*) [17, 81], na transformação de operações [46, 159], na exploração de propriedades semânticas [161, 79, 168], na ordenação de operação usando propriedades semânticas [85, 123], na criação de múltiplas versões [24, 101, 88]. No entanto, parece não existir nenhum método ideal para todas as situações. Os exemplos anteriores sugerem que diferentes situações devem ser tratadas de forma diferente. A melhor estratégia deve ser escolhida em função das características específicas de cada aplicação ou de cada operação executada.

Muito sistemas permitem que a estratégia de reconciliação seja adaptada por cada aplicação. Por exemplo, no sistema Coda [87], cada aplicação pode fornecer os seus programas de reconciliação que actuam com base no estado das réplicas dos ficheiros. No sistema Bayou [161], a estratégia de reconciliação consiste em executar, em todas as réplicas, todas as operações (deterministas) pela mesma ordem, podendo cada operação conter código específico de detecção e resolução de conflitos.

Nesta dissertação propõe-se que, além de permitir a adaptação da estratégia às características específicas de cada aplicação, se permita também que, em cada aplicação, seja usada a estratégia mais apropriada face à semântica da aplicação. Assim, é possível usar em cada aplicação a estratégia que permite criar a solução desejada de forma mais simples. No capítulo 3 apresenta-se um sistema que permite esta adaptabilidade. Esta aproximação permite ainda criar catálogos de soluções reutilizáveis em diferentes aplicações com características semelhantes.

2.3.3 Replicação baseada na propagação de operações

Num sistema baseado em replicação optimista é necessário lidar com modificações executadas concorrentemente. Existem duas alternativas para propagar estas modificações entre as várias réplicas de um objecto. A primeira consiste em propagar o novo estado da réplica após a execução das modificações [80, 87, 66]. Neste caso, o mecanismo de reconciliação usa o estado das várias réplicas dos dados para obter o estado final de cada réplica (que pode, eventualmente, conter múltiplas versões dos dados). A segunda alternativa consiste em propagar as operações executadas para modificar os dados [161, 79, 85]. Neste caso, o mecanismo de reconciliação usa o estado inicial dos dados e as operações efectuadas para

obter o estado final de cada réplica.

Um dos aspectos a considerar sobre a forma de propagação das modificações é o tráfego na rede induzido por cada um dos métodos. Quando se propaga o estado é necessário propagar uma cópia do objecto entre as réplicas. Por exemplo, se o objecto que estamos a considerar é um ficheiro, é necessário propagar o novo conteúdo do ficheiro (utilizando algumas optimizações é possível propagar apenas as partes dos ficheiros que foram alteradas [164]). No caso de terem existido várias modificações ao mesmo objecto numa réplica, apenas o estado final é propagado. Na propagação de operação, todas as operações efectuadas são propagadas. No entanto, é possível reduzir o número de operações a propagar removendo as operações cujos efeitos são absorvidos por outras operações ou comprimindo várias operações numa só (por exemplo, o algoritmo de compressão de uma sequência de operações apresentado na secção 10.4.2).

O tráfego induzido por cada um dos métodos depende, em grande parte, do modo como o sistema é utilizado, da dimensão dos objectos e das operações disponíveis no sistema, e da eficiência das optimizações executadas. Este facto é demonstrado de forma simples pelos exemplos seguintes. Num primeiro exemplo, consideremos um inteiro ocupando 4 bytes que é incrementado 1000 vezes durante um período de desconexão. No caso da propagação do estado, apenas é necessário propagar o estado final do objecto — 4 bytes. No caso de propagação de operações é necessário propagar as 1000 operações efectuadas. Se for possível comprimir estas operações numa só (incrementar 1000 vezes um inteiro é equivalente a adicionar 1000 a esse inteiro), então apenas teremos de propagar uma operação. Neste caso, a propagação do estado é mais eficiente (ou pelo menos tão eficiente no caso de ser possível comprimir as operações executadas). Num segundo exemplo, consideremos uma lista de 1000 inteiros que se pretendem incrementar. Propagando o estado é necessário propagar todos os inteiros. Propagando as operações, basta propagar uma operação que incremente cada um dos 1000 inteiros.

Este problema foi estudado no contexto do sistema de ficheiros distribuído Coda [96] utilizando várias aplicações não interactivas. Neste contexto, obtiveram reduções do tráfego gerado por um factor de 20 quando utilizavam a propagação de operações (por comparação com a tradicional propagação dos ficheiros).

Um segundo aspecto a considerar é o da relação entre o modo de propagação das modificações e as estratégias de reconciliação usadas. Embora, como já discutimos anteriormente, não exista nenhum método ideal para tratar as modificações concorrentes, o uso de informação semântica foi identificado como a chave para uma boa solução [147, 161, 79, 87]. Dois tipos de informação semântica podem ser usados durante a reconciliação. Primeiro, a informação semântica associada aos dados manipulados — este tipo de informação pode ser usada em qualquer uma das aproximações. Segundo, a informação semântica associada às operações executadas. Quando se propaga o estado, apenas é possível tentar

inferir quais as operações executadas a partir do novo estado do objecto (por comparação com o estado inicial). Quando se propagam as operações é possível associar a cada operação informação semântica usada durante a reconciliação. Por exemplo, podem incluir-se regras de detecção e resolução de conflitos. Assim, e para maximizar a informação semântica que pode ser usada durante a reconciliação, as soluções apresentadas nesta dissertação são baseadas na propagação de operações.

2.3.4 Informação sobre a evolução dos dados e da actividade cooperativa

Quando se permite que múltiplos utilizadores modifiquem concorrentemente dados partilhados, é muitas vezes importante fornecer informação sobre a sua evolução. No exemplo do editor cooperativo apresentado anteriormente, um utilizador pode estar interessado em saber quais as alterações efectuadas ao documento recentemente. O acesso a este tipo de informação é importante para o sucesso das actividades cooperativas, como foi reconhecido na literatura de trabalho cooperativo há algum tempo [47, 65, 43]. A esta informação dá-se o nome de informação de *awareness*[5].

Num sistema baseado em replicação optimista é necessário reconciliar modificações conflituantes. Assim, é comum que o resultado final duma operação (i.e., o modo como a operação afecta o estado "oficial" dos dados) apenas seja definitivamente conhecido após a reconciliação das várias réplicas.

Das características anteriores resultam três factos importantes. Primeiro, os utilizadores não têm geralmente conhecimento imediato do resultado das operações que submetem. Este facto sugere a necessidade de existir um mecanismo que permita notificar os utilizadores do resultado final das operações. Por exemplo, na agenda partilhada os utilizadores devem ser informados do resultado definitivo das suas marcações.

Segundo, a informação sobre a evolução dos dados apenas pode ser criada aquando do processo de reconciliação porque apenas neste momento é determinado o resultado final da operação. Assim, a criação da informação deve estar integrada com o mecanismo de reconciliação.

Finalmente, os utilizadores podem não estar a usar o computador ou a aplicação em que executaram as operações quando o resultado final destas é determinado. Assim, se se pretender informar imediatamente o utilizador do resultado das suas operações, pode não ser suficiente propagar o resultado para a máquina em que foram executadas, porque esta máquina pode estar desligada/desconectada ou o utilizador pode não estar a utilizá-la. Actualmente, com o uso generalizado de telefones móveis, é geralmente possível enviar informação para os utilizadores através de mensagens curtas (por exemplo, o serviço SMS nas redes GSM). Assim, o sistema deve usar os meios disponíveis para propagar a informação

[5] O termo inglês *awareness* é, por vezes, traduzido pelos termos consciência, conhecimento ou sensibilização. No entanto, por se considerar que a utilização destas traduções não incluem o sentido completo do termo original, obscurecendo o texto, decidiu-se utilizar o termo inglês em itálico.

sobre a evolução dos dados para os utilizadores, mesmo aqueles que podem ser considerar exteriores ao sistema.

Dos exemplos apresentados, é necessário realçar que não existe um modo único de fornecer informação sobre a evolução dos dados. Nuns casos, esta informação deve ser apresentada quando se acede aos dados — o editor cooperativo é um exemplo. Noutros casos é necessário propagar a informação directamente para os utilizadores — a agenda partilhada é um exemplo. Assim, o modo como esta informação é tratada pelo sistema deve ser definida em função das características específicas da aplicação (e muitas vezes dentro de uma mesma aplicação é necessário adaptar o modo de tratamento à situação e ao utilizador).

Embora as características apresentadas anteriormente sejam comuns à generalidade dos sistemas que suportam operação desconectada [115, 85, 59, 161, 79, 87], o problema de fornecer informação sobre a evolução dos dados é geralmente negligenciada. Este facto leva os programadores a implementar soluções *ad-hoc* sem o conveniente suporte por parte do sistema.

Nesta dissertação apresentam-se soluções em que a criação e o tratamento da informação sobre a evolução dos dados estão completamente integrados no sistema. Todas as operações efectuadas pelos utilizadores podem gerar informação e o modo como esta informação é tratada pode depender da aplicação ou situação. O sistema fornece suporte para a propagação da informação para os utilizadores através de mecanismos exteriores ao sistema. O sistema também permite que esta informação seja mantida com os dados para posterior acesso pelos utilizadores. Alguns sistemas de suporte ao trabalho cooperativo [16, 44, 97] incluem este tipo de mecanismos. No entanto, nesta dissertação apresenta-se uma nova aproximação através da integração destes mecanismos ao nível dos tipos de dados e das operações definidas.

2.3.5 Replicação secundária parcial

A replicação dos dados em vários servidores não garante a disponibilidade dos dados durante os períodos de desconexão. Para tal, é comum recorrer a um mecanismo de replicação secundária (*caching*) para manter cópias parciais dos dados nos computadores portáteis.

No sistema Coda [87], bem como noutros sistemas de ficheiros distribuídos, a unidade de replicação secundária é o ficheiro. Esta aproximação permite uma semântica simples e clara de acesso aos ficheiros: ou é possível aceder ao ficheiro completo ou não é possível aceder a nenhuma parte do ficheiro (quando não existe uma cópia local). Quando os ficheiros são muito longos, pode ser impossível a um dispositivo móvel com recursos limitados manter cópias de ficheiros completos. Adicionalmente, o tempo necessário para transmitir uma cópia total dum ficheiro pode ser inaceitável — este problema acentua-se para dispositivos móveis que utilizam redes de comunicação sem fios com reduzida largura de banda.

Nas bases de dados orientadas para os objectos (BDOO), os objectos são a unidade de manipulação dos dados. No entanto, os objectos são geralmente agrupados em páginas aquando da sua criação e, em alguns sistemas, reagrupados usando algoritmos de agrupamento (*clustering*) [165, 27, 55]. Para melhorar o desempenho, as BDOO incluem um subsistema de replicação secundária (*caching*), no qual os clientes ou os servidores mantêm em memória cópias de um conjunto de objectos ou páginas de objectos [23, 109].

Para permitir a operação durante períodos de desconexão é necessário obter cópias de todos os objectos que vão ser acedidos. No entanto, como os objectos são geralmente de pequena dimensão (por exemplo, na base de dados dos testes de referência *OO7* [22], a dimensão média dos objectos é 51 bytes no sistema Thor [100]) e em muito elevado número, determinar os objectos que devem ser replicados é extremamente complexo. Note-se que o agrupamento dos objectos em páginas não simplifica o processo de análise porque os objectos e as suas ligações têm de ser tratados individualmente.

Apesar de os objectos serem a unidade de manipulação, as aplicações tendem a manipular objectos complexos compostos por um grafo de objectos simples. O sistema DOORS, apresentado no capítulo 3 desta dissertação, adopta uma solução de replicação secundária parcial em que se explora esta observação. Assim, a unidade de replicação secundária é o grupo de objectos fortemente interligados que definem um objecto complexo — por exemplo, num documento estruturado, uma secção representa uma unidade de replicação secundária (*caching*), embora seja composta por múltiplos objectos. Como cada objecto complexo define uma caixa negra (i.e., apenas pode ser acedido através da sua interface), o sistema de replicação apenas lida com objectos complexos. Os programadores, ao desenharem os objectos manipulados pelas aplicações são responsáveis por identificar os objectos complexos. Assim, a granularidade da replicação secundária é definida tendo em conta a semântica dos dados.

Nos sistemas de bases de dados relacionais que suportam computação móvel [115, 142] é usual manter cópias de subconjuntos dos dados, os quais são seleccionados através de interrogações à base de dados (usando variantes do comando *select*). Desta forma, pode adaptar-se a granularidade das cópias locais às necessidades específicas de cada utilizador. A semântica dos dados e suas interligações devem ser exploradas nas interrogações para obter subconjuntos coesos de dados (i.e., mantendo sempre que possível a integridade referencial). No sistema Mobisnap, apresentado no capítulo 7 desta dissertação, usa-se uma solução semelhante.

Nas soluções apresentadas nesta dissertação apenas se fornecem os mecanismos que permitem definir, obter e manter cópias parciais locais dos dados. O problema de determinar de forma eficiente quais os dados que devem ser obtidos para suportar a operação desconectada está fora do âmbito deste trabalho. Algumas soluções são apresentadas em [87, 91].

2.3.6 Invocação cega

Durante os períodos de desconexão, o acesso aos dados é garantido recorrendo a cópias locais — os utilizadores acedem e modificam estas cópias. No entanto, é possível que não exista uma cópia local dos dados que os utilizadores pretendem manipular. Nesta situação, é obviamente impossível ao sistema disponibilizar operações de leitura que devolvam o estado actual dos dados (ou mesmo, uma versão recente). Nos sistemas de gestão de dados distribuídos, este facto impede normalmente os utilizadores de executarem quaisquer operações sobre os dados.

No entanto, existem situações em que é possível permitir aos utilizadores uma forma limitada de acesso aos dados, assim minimizando os problemas decorrentes das faltas na replicação secundária, i.e., da inexistência de uma cópia local de um objecto acedido pelo utilizador. Por exemplo, numa agenda partilhada parece razoável permitir a submissão de novas marcações sem acesso a uma cópia da agenda. Estes pedidos devem ser confirmados através do mecanismo automático de consenso global, como todas os outros pedidos.

O princípio subjacente consiste em permitir a submissão de operações que modifiquem objectos não disponíveis localmente, deixando aos utilizadores a decisão sobre quais as situações em que é aconselhável produzir novas contribuições[6]. No âmbito da evolução do estado global dos dados, estas operações devem ser tratadas de modo idêntico a todas as outras, i.e., estas operações são propagadas para os servidores onde são integradas no estado dos dados (de acordo com o mecanismo de reconciliação usado).

Um aspecto a considerar é o modo como estas operações são tratadas localmente. Podemos considerar dois casos distintos. No primeiro caso, o tipo de objectos manipulados pela aplicação é desconhecido[7] Nesta situação, apenas é possível aceitar as operações executadas pelo utilizador. O sistema não pode sequer verificar localmente se as operações são válidas (por exemplo, numa sistema de bases de dados é impossível verificar se as operações actuam apenas sobre tabelas existentes).

Uma segunda situação é aquela em que, apesar de não se ter uma cópia local dos dados, são conhecidos os seus tipos. Neste caso, existem situações em que pode ser interessante criar uma cópia de substituição dos objectos e executar as operações localmente sobre essa cópia. No exemplo do editor cooperativo descrito anteriormente, pode ter-se uma cópia parcial dum documento estruturado que contenha apenas a estrutura do documento e algumas secções. Se o utilizador quiser criar uma nova versão duma secção não presente localmente, é possível criar um objecto de substituição que represente a secção. O utilizador pode então executar operações sobre o objecto de substituição, introduzindo novas secções ou modificando as secções introduzidas anteriormente. O resultado observado na cópia de substituição deve

[6]Note-se que um utilizador pode ter um conhecimento aproximado do estado de um objecto apesar de o computador que usa não possuir nenhuma cópia desse objecto. Este conhecimento pode ter sido obtido num acesso recente efectuado noutro computador ou através de meios exteriores ao sistema.

[7]Numa sistema de bases de dados relacionais, esta situação equivale a ser desconhecido o esquema da base de dados.

ser considerado provisório como acontece normalmente com as invocações executadas no cliente. O resultado definitivo das operações apenas é determinado após a sua integração no estado "oficial" do objecto. A criação de cópias de substituição permite às aplicações manipular de forma semelhante todos os objectos (embora a interface da aplicação deva reflectir o facto de se tratar de uma cópia de substituição).

Nas soluções apresentadas nesta dissertação é possível executar (submeter) operações sobre dados não replicados localmente. No capítulo 3 apresenta-se um sistema que permite também a utilização de objectos de substituição. Embora esta técnica não possa ser usada em todas as situações, os exemplos apresentados demonstram a sua utilidade quando usada convenientemente. No capítulo 7 apresenta-se um sistema que permite a submissão de transacções que não podem ser confirmadas com os dados disponíveis localmente.

Que o autor tenha conhecimento, nenhum outro sistema implementa estas ideias, apesar de vários serem baseados na propagação de operações [123, 108, 161]. Por exemplo, no sistema Rover [79], as operações executadas nas cópias locais dos objectos são propagadas para o servidor através dum mecanismo de RPC diferido. No entanto, apenas é possível executar operações sobre objectos disponíveis localmente.

2.3.7 Prevenção de conflitos

A necessidade de reconciliar as modificações executadas independentemente leva a que o resultado ou efeito de uma operação não seja geralmente conhecida imediatamente quando a operação é executada. O resultado definitivo de uma operação apenas é determinado após a sua integração final no estado oficial dos dados. Este facto é comum nos sistemas baseados em replicação optimista [87, 161, 59, 115].

O conhecimento imediato dos resultados das operações executadas é importante em algumas aplicações. Por exemplo, num sistema de reserva de lugares para comboios é importante que os utilizadores possam saber imediatamente o resultado dos seus pedidos.

A possibilidade de obter o resultado de uma operação de forma independente elimina a necessidade de comunicar de forma síncrona com o servidor. Assim, é possível reduzir o tempo de processamento de uma operação executando-a localmente. Dado que a latência das comunicações tende a manter-se sem alterações significativas (ao contrário da fiabilidade e largura de banda), esta propriedade pode ser importante mesmo quando é possível contactar o servidor.

Adicionalmente, é possível utilizar os recursos comunicacionais disponíveis de forma mais eficiente propagando conjuntos de operações de forma assíncrona. O servidor pode também adiar o processamento das operações para os momentos de carga mais reduzida.

A forma de garantir o sucesso de uma operação de forma independente consiste em restringir as operações que podem ser executadas concorrentemente, assim garantindo que não surgem conflitos (in-

solúveis). As técnicas de trincos (*locks*) [17, 60] usadas tradicionalmente em sistemas distribuídos são um exemplo extremo desta estratégia: o utilizador que possui um trinco pode modificar os dados enquanto todos os outros estão impedidos de efectuar qualquer modificação.

Outro exemplo dessa estratégia consiste em utilizar a informação semântica associada aos tipos de dados representados e às operações que podem ser executadas sobre esses tipos de dados para garantir o sucesso de operações executadas concorrentemente. Por exemplo, é sempre possível garantir o sucesso de uma operação de incremento executada sobre um número inteiro (de dimensão não limitada). Esta ideia pode ser estendida limitando as operações que podem ser executadas (ou, o que é equivalente, restringindo os valores possíveis de cada réplica). Estas ideias foram propostas anteriormente para garantir o sucesso de operações executadas concorrentemente [111, 168].

No capítulo 7 apresentamos um sistema de reservas que permite garantir independentemente o sucesso (de algumas) das operações executadas numa base de dados para computação móvel. Quando é impossível garantir o sucesso de uma operação, é possível executá-la de forma optimista (sendo o resultado final determinado aquando da sua integração na versão oficial dos dados). Este sistema adapta e estende algumas ideias propostas anteriormente [60, 58, 111] a um ambiente de computação móvel e a um sistema de bases de dados SQL.

2.3.8 Suporte para os programadores

Nesta dissertação apresentam-se dois sistemas de gestão de dados para ambientes de computação móvel. Nas secções anteriores discutiram-se os princípios gerais adoptados nesses sistemas. Existem duas características importantes que devem ser tidas em consideração. A primeira, é a necessidade de permitir o desenvolvimento de soluções específicas para cada aplicação. A segunda, é a necessidade de permitir que as diferentes soluções sejam definidas e utilizadas da forma mais simples possível.

A necessidade de permitir o desenvolvimento de soluções específicas para cada aplicação foi justificada nas secções anteriores, com especial destaque para o problema da reconciliação e tratamento da informação sobre a evolução dos dados. Duas aproximações podem ser consideradas para fornecer esta flexibilidade. Primeiro, é possível definir um sistema com uma estratégia única que pode ser adaptada a cada situação específica. Esta aproximação é adoptada em vários sistema distribuídos de gestão de dados — os sistemas Coda [87] e Bayou [161] são apenas dois exemplos. A solução apresentada no capítulo 7 desta dissertação também adopta esta aproximação. A segunda aproximação consiste em permitir a adopção de estratégias diferentes em diferentes situações. A ideia consiste em permitir que se adopte a estratégia que permite a solução mais simples em cada situação. O sistema Globe [12] e o sistema proposto em [20] usam esta aproximação. A solução apresentada no capítulo 3 desta dissertação também adopta esta aproximação.

No entanto, a flexibilidade no desenvolvimento de novas solução não é suficiente para suportar convenientemente os programadores na criação de novas aplicações. Um aspecto a ter em consideração é o facto de os programadores que vão criar novas aplicações não serem (necessariamente) especialistas em sistemas distribuídos. Desta forma, é necessário permitir que estes programadores utilizem as técnicas de sistemas distribuídos de forma simples.

Nos sistemas que apenas permitem a adaptação duma estratégia pré-definida, é necessário que essa adaptação possa ser efectuada de forma simples. Por esta razão, as soluções apresentadas no sistema Mobisnap (capítulo 7) são baseados em mecanismos utilizados de forma generalizada — transacções e linguagem PL/SQL. Assim, os programadores podem definir as suas soluções usando conceitos que lhe são familiares.

Nos sistemas que permitem a utilização de diferentes estratégias é necessário que os programadores de aplicações possam seleccionar e adaptar a estratégia que pretendem utilizar de forma simples. É também necessário que seja possível definir novas estratégias. No sistema Doors (capítulo 3) definimos um *framework* de componentes que permite aos programadores compor uma solução global, reutilizando soluções pré-definidas para os diferentes problemas. Para cada um dos problemas, é possível definir novas soluções criando novos componentes (a decomposição dos problemas relacionados com a gestão dos dados em vários componentes permite concentrar a atenção num único problema de cada vez).

2.4 Sumário

Nesta secção apresentaram-se os princípios fundamentais adoptados nas soluções de gestão de dados propostas nesta dissertação. Para motivar estes princípios, descreveu-se a utilização de um conjunto de aplicações típicas de um ambiente de computação móvel.

Vários sistemas propuseram anteriormente soluções de gestão de dados para ambientes de computação móvel que colocavam em prática os princípios identificados. As soluções apresentadas, inicialmente muito rígidas, foram-se tornando mais flexíveis e adaptadas aos vários problemas através da exploração da informação semântica.

As soluções propostas nesta dissertação para colocar em prática os vários princípios identificados têm em comum a preocupação de explorar ao máximo a informação semântica disponível. No próximo capítulo apresentam-se as soluções propostas no âmbito de um repositório de objectos desenhado para suportar o desenvolvimento de actividades cooperativas tipicamente assíncronas.

Capítulo 3

Apresentação do sistema DOORS

Nos próximos capítulos descreve-se um sistema de gestão de dados desenhado com o objectivo de gerir dados partilhados em actividades cooperativas tipicamente assíncronas: o sistema DOORS [129, 130, 131, 132] (*DAgora Object-Oriented Replicated Store*). Este sistema foi desenvolvido no âmbito do projecto *DAgora* [34]. O objectivo deste projecto era o estudo e desenvolvimento de suporte computacional para actividades cooperativas. No âmbito deste projecto foram estudados vários problemas, entre os quais: o suporte para aplicações cooperativas síncronas [156]; a coordenação de actividades cooperativas [40, 39]; e a disseminação de informação de *awareness* [45]. Estes problemas estão fora do âmbito desta dissertação, a qual se restringe aos problemas de gestão de dados em ambientes distribuídos de larga-escala.

O sistema DOORS é um sistema de gestão de dados que fornece elevada disponibilidade de leitura e escrita num ambiente de computação móvel usando um modelo de replicação optimista. O sistema permite o desenvolvimento de soluções específicas para cada tipo de dados. Para tal, definiu-se um *framework* de componentes que decompõe os vários aspectos relacionados com a gestão de dados partilhados em vários componente. Neste capítulo apresenta-se brevemente o funcionamento do sistema e detalha-se o *framework* de componentes.

No próximo capítulo discutem-se, de forma mais pormenorizada, alguns dos aspectos de gestão de dados que dependem do tipo de aplicação. Entre estes, abordam-se o problema da reconciliação, da invocação cega e da integração de sessões síncronas. No capítulo 5 apresentam-se várias aplicações que exemplificam a utilização do sistema DOORS e, em particular, do seu *framework* de componentes. A arquitectura do sistema DOORS, incluindo os protocolos implementados no protótipo do sistema, são detalhados no capítulo 6.

De seguida, descreve-se o modelo geral do sistema, incluindo uma breve descrição da sua arquitectura e do modo de funcionamento dos objectos geridos pelo mesmo.

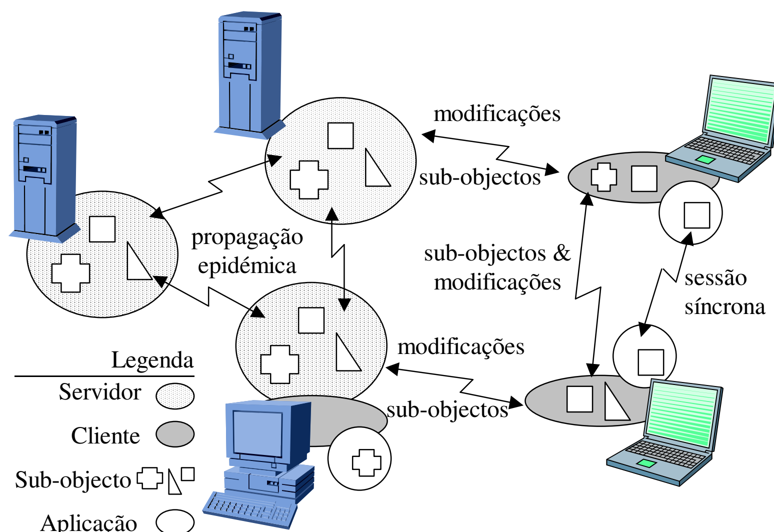

Figura 3.1: Arquitectura do sistema DOORS composta por computadores com diferentes configurações.

3.1 Modelo geral

O sistema DOORS é um repositório distribuído de objectos baseado numa arquitectura *cliente estendido/servidor* [78]. O sistema manipula objectos estruturados segundo o modelo de componentes DOORS, que se designam por *coobjectos*. Um *coobjecto* representa um tipo de dados possivelmente complexo, que pode ser formado por uma composição de outros objectos. Estes objectos são agrupados em conjuntos (*clusters*) de objectos relacionados, que se designam por subobjectos. Por exemplo, um *coobjecto* pode representar um documento estruturado como esta dissertação. A estrutura do documento e cada um dos elementos básicos (por exemplo, os capítulos desta dissertação) podem ser definidos como subobjectos.

Os *coobjectos* relacionados são agrupados em conjuntos que representam um espaço de trabalho cooperativo (*collaborative workspace*) e incluem os dados associados a uma sessão de trabalho cooperativo ou a um grupo de trabalho cooperativo. Estes conjuntos de *coobjectos* são designados por volumes [87].

O sistema DOORS é composto por dois componentes, clientes e servidores, como se representa na figura 3.1. Em geral, os servidores executam em máquinas fixas com boa conectividade. Os clientes podem executar em máquinas fixas ou móveis e a sua conectividade é geralmente variável. Qualquer máquina pode actuar simultaneamente como cliente e servidor. Por vezes, e quando essa designação não cause confusão, também se designa de servidor (cliente) a máquina em que executa o componente servidor (cliente) do sistema DOORS.

Para fornecer uma elevada disponibilidade de leitura e escrita, os *coobjectos* do sistema DOORS são replicados usando um modelo de replicação optimista. Assim, qualquer réplica pode ser modificada de forma independente, maximizando a possibilidade de os utilizadores efectuarem as suas contribuições

para a actividade cooperativa.

Cada servidor replica um conjunto de volumes de *coobjectos*. Para cada volume, replica todos os *coobjectos* que estão contidos nesse volume. O estado destas réplicas é sincronizado durante sessões de sincronização epidémicas estabelecidas entre pares de servidores. A replicação nos servidores tem por objectivo mascarar as falhas nos servidores e no sistema de comunicação.

Os clientes executam um mecanismo de replicação secundária (os *caching*), mantendo uma *cache*[1] com cópias parciais de um conjunto de *coobjectos*. Uma cópia parcial de um *coobjecto* inclui apenas um subconjunto de todos os subobjectos que compõem um *coobjecto*. Os clientes sincronizam o estado das cópias locais com os servidores e com outros clientes. A replicação parcial nos clientes tem por objectivo mascarar os períodos de desconexão, permitindo aos utilizadores continuarem a ler e modificar os dados partilhados.

As aplicações que utilizam o sistema DOORS executam nos clientes e modificam os *coobjectos* através da execução de métodos dos subobjectos — os utilizadores cooperam entre si modificando dados partilhados. De seguida, apresenta-se o modelo de acesso aos *coobjectos*.

3.1.1 Modelo de manipulação dos *coobjectos*

Em geral, as aplicações acedem aos *coobjectos* usando o modelo de acesso "obtém/modifica localmente/submete modificações". Assim, quando uma aplicação solicita o acesso a um *coobjecto*, o cliente, caso não possua uma cópia local, obtém-na a partir de um servidor (ou, caso tal não seja possível, de outro cliente). De seguida, o cliente cria uma cópia privada do *coobjecto* e entrega-a à aplicação.

Uma aplicação manipula a cópia privada do *coobjecto* e os seus subobjectos de forma semelhante a um grafo de objectos comuns, i.e., usando os seus métodos para aceder e modificar o estado dos objectos e navegar na estrutura definida. As operações de modificação executadas pelas aplicações são registadas no *coobjecto* de forma transparente para a aplicação e sem intervenção do cliente do sistema DOORS (ver detalhes na secção 3.2.2.2). Este registo consiste na sequência de invocações de métodos[2] efectuadas pela aplicação.

Inicialmente a cópia privada de um *coobjecto* inclui apenas referências para um (ou vários) subobjecto raiz. Os subobjectos apenas são instanciados quando necessário (i.e., quando são acedidos). Nesse momento, o cliente cria uma cópia privada do subobjecto[3] e liga-a à cópia privada do *coobjecto* — todo este processo é transparente para a aplicação.

[1]Usa-se o termo *cache* para designar o espaço usado para guardar réplicas secundárias dos dados, neste caso, réplicas parciais dos *coobjectos*.

[2]A invocação de um método é também denominada de operação.

[3]Para tal, o cliente pode ter necessidade de contactar um servidor se a sua cópia parcial do *coobjecto* não incluir o subobjecto em questão.

Nas situações em que o cliente não consegue obter uma cópia do subobjecto a que a aplicação está a aceder, o mecanismo de invocação cega do sistema DOORS permite que as operações de modificação executadas sejam igualmente registadas, apesar de não serem executadas. O mecanismo de invocação cega permite ainda criar cópias de substituição dos subobjectos. Assim, é possível observar o resultado esperado das operações executadas pelos utilizadores.

O mecanismo de invocação cega, cujos princípios foram discutidos na secção 2.3.6, pode ser considerado como uma extensão ao modelo de replicação optimista, permitindo submeter operações de modificação sobre dados dos quais não se possui uma réplica local. Esta técnica permite mascarar as falhas na replicação secundária e maximizar a disponibilidade dos dados.

Após manipular a cópia privada de um *coobjecto*, se o utilizador decidir gravar as modificações efectuadas, a sequência de invocações registada na cópia privada do *coobjecto* é transferida para o cliente. O cliente armazena esta sequência de invocações de forma estável até ter possibilidade de a propagar para um servidor.

O servidor, ao receber de um cliente uma sequência de invocações, entrega-a à réplica local do *coobjecto*. É da responsabilidade da cópia do *coobjecto* presente no servidor armazenar e processar estas invocações.

Os servidores sincronizam o estado das suas réplicas, sincronizando o conjunto de invocações conhecidas. Para tal, estabelecem-se sessões de sincronização epidémicas entre pares de servidores. Este modelo de sincronização garante que, no final, todos os servidores receberam todas as invocações efectuadas, quer directa, quer indirectamente. As sessões de sincronização podem ser estabelecidas sobre vários transportes, síncronos ou assíncronos. Os servidores obtém as invocações a propagar, e entregam as invocações recebidas, às réplicas locais dos *coobjectos*. É sempre da responsabilidade das réplicas dos *coobjectos* armazenar e processar estas invocações.

Adicionalmente, cada servidor propaga, para os outros servidores que replicam o volume, através dum sistema de disseminação de eventos não fiável, as sequências de invocações recebidas directamente dos clientes (após estas terem sido estampilhadas pela réplica local). Esta propagação tem por objectivo diminuir a divergência momentânea entre as várias réplicas (resultado de conhecerem diferentes subconjuntos de invocações), aumentando a rapidez de propagação das modificações.

3.1.2 Modelo de funcionamento do sistema

A breve descrição apresentada anteriormente permite constatar que o funcionamento do sistema DOORS se baseia na cooperação entre o núcleo do sistema, responsável pelas funcionalidades de gestão de dados comuns, e os *coobjectos*, responsáveis pelas soluções específicas.

O *núcleo do sistema* apenas executa as tarefas comuns relacionadas com a gestão de dados partilha-

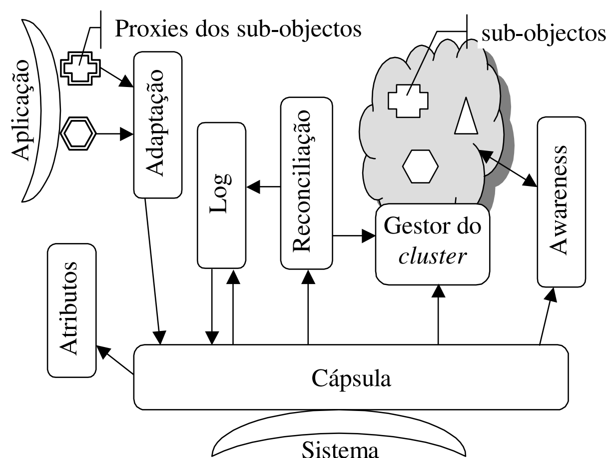

Figura 3.2: *Framework* de componentes DOORS.

dos. Por um lado, implementa as funcionalidades necessárias para que o sistema apresente uma elevada disponibilidade de acesso, entre as quais: a gestão da filiação de cada volume (i.e., do conjunto de servidores que replica cada volume); a gestão da sincronização dos dados entre os vários servidores; e a gestão das réplicas secundárias (*cache*) nos clientes. Por outro lado, o núcleo do sistema executa os serviços necessários ao funcionamento dos *coobjectos*, entre os quais: a gestão dos recursos locais associados a cada *coobjecto* (no cliente e no servidor); o serviço de descoberta (de volumes); o serviço de propagação de mensagens de *awareness*; o serviço de propagação síncrona de operações (entre um cliente e um servidor).

O sistema delega nos *coobjectos* a implementação da maioria dos aspectos relacionados com a gestão dos dados partilhados. Entre estes aspectos, pode destacar-se a gestão da informação sobre as modificações executadas pelos utilizadores, o mecanismo de reconciliação das várias réplicas nos servidores e a gestão da informação de *awareness*.

Esta solução contribui para a flexibilidade do sistema DOORS, permitindo a definição de soluções específicas para cada problema. No entanto, coloca um enorme responsabilidade na implementação dos *coobjectos*, os quais têm de tratar de aspectos que normalmente são da responsabilidade do núcleo do sistema. Para fazer face a este problema, o sistema DOORS define um *framework* de componentes em que se decompõe o funcionamento de um *coobjecto* em vários componentes (ver figura 3.2).

Usando este *framework*, é possível criar um *coobjecto* compondo um componente que implementa o tipo de dados a criar, com um conjunto de componentes em que cada um implementa uma solução específica para um dos problemas identificados no *framework*. Assim, é possível reutilizar a mesma solução em diferentes *coobjectos* através da simples reutilização do componente que implementa essa solução.

Esta aproximação adapta-se às características das aplicações cooperativas, como apresentadas no ca-

pítulo 2, em que diferentes grupos de aplicações usam as mesmas estratégias relativamente a diferentes problemas, como por exemplo a reconciliação e o tratamento da informação de *awareness*. Adicionalmente, esta aproximação permite aos programadores reutilizar técnicas complexas de sistemas distribuídos sem terem de lidar com os detalhes de implementação dessas soluções (os quais são encapsulados nos componentes pré-definidos).

De seguida, apresenta-se o *framework* de componentes DOORS, explicando o modo como é possível criar um novo *coobjecto*. Exemplos detalhados de *coobjectos* criados no âmbito de aplicações cooperativas são descritos no capítulo 5. O núcleo do sistema é detalhado no capítulo 6.

3.2 *Framework* de componentes: princípios gerais

O *framework* de componentes DOORS decompõe um *coobjecto* num conjunto de componentes, cada um tratando dum aspecto específico do funcionamento do *coobjecto*. Os vários componentes interactuam entre si para implementar uma solução global de gestão de dados. As responsabilidades de cada componente são bem-definidas e, tanto quanto possível, isoladas. No entanto, algumas implementações de um componente podem exigir a implementação de uma dada funcionalidade noutro componente, restringindo as composições possíveis.

De seguida apresenta-se de forma breve o *framework* DOORS e o modo como os vários componentes interagem. Em particular, descreve-se o funcionamento de um *coobjecto* composto por um conjunto de subobjectos. Na próxima secção detalha-se cada um dos componentes do *framework* e descrevem-se alguns dos componentes pré-definidos no protótipo do sistema DOORS.

3.2.1 Subobjectos

Um *coobjecto* define um tipo de dados, possivelmente complexo, especialmente desenhado para ser partilhado por vários utilizadores e manipulado concorrentemente — por exemplo, um documento estruturado. Uma instância de um *coobjecto* mantém uma representação do valor actual do tipo de dados através de uma composição de subobjectos — por exemplo, um subobjecto com a estrutura do documento e um subobjecto por cada elemento.

Um subobjecto pode ser um objecto complexo constituído por um conjunto de outros objectos (definidos ao nível da linguagem de programação usada para definir o subobjecto). No entanto, no âmbito de um *coobjecto*, um subobjecto representa uma "caixa preta" com um estado interno que apenas pode ser acedido e modificado através da invocação de métodos da interface que implementa. O subobjecto constitui a unidade de manipulação dentro de um *coobjecto*. Os subobjectos são acedidos por navegação a partir de um subobjecto base (i.e., apesar de internamente os subobjectos terem um identificador

único, uma aplicação nunca acede a um subobjecto usando directamente este identificador). Para tal, os subobjectos podem manter referências para outros subobjectos.

Embora um *coobjecto* inclua instâncias de vários outros componentes, são os subobjectos que representam o valor do tipo de dados que se pretende definir — os outros componentes são usados para controlar os vários aspectos da partilha dos dados. Assim, pode dizer-se que o estado de um *coobjecto* é representado pelo estado dos subobjectos que o constituem e que reflectem a execução de uma determinada sequência de operações. No entanto, de forma mais exacta, dever-se-ia considerar que o estado de um *coobjecto* inclui também o estado dos outros componentes.

As aplicações e os subobjectos não mantêm referências directas para um subobjecto. Estas referências são implementadas através de representantes de subobjectos (*proxies*) que implementam a mesma interface dos subobjectos. O uso de representantes elimina a necessidade de interceptar as invocações de métodos num subobjecto ao nível do sistema de execução dos programas.

Os representantes (*proxies*) permitem executar de forma simples as seguintes funcionalidades necessárias ao funcionamento do sistema DOORS. Primeiro, os representantes permitem interceptar as invocações de métodos de um subobjecto e criar o necessário registo de invocações (esta características é fundamental num sistema de replicação baseado na propagação das modificações como sequências de operações). Segundo, os representantes permitem que apenas se instanciem os subobjectos quando é necessário executar métodos desses subobjectos. Terceiro, os representantes eliminam a necessidade de modificar os apontadores usados para referenciar outros subobjectos quando um subobjecto é instanciado [4]. Quarto, os representantes encapsulam os identificadores dos subobjectos, os quais não são visíveis pelas aplicações — o acesso aos subobjectos efectua-se sempre por navegação (a partir de um objecto base), como é normal nas linguagens orientadas para os objectos.

A instanciação de um subobjecto é executada pelo gestor de subobjectos (*cluster manager*) apenas quando necessário. O gestor de subobjectos mantém referências para todos os subobjectos presentes em memória de forma a garantir que apenas existe uma instância de cada subobjecto.

Adicionalmente, o gestor dos subobjectos é responsável por manter referências para o subobjecto base (ou conjunto de subobjectos base) — por exemplo, no documento estruturado, a estrutura principal do documento é o subobjecto base, o qual permite aceder aos outros subobjectos por navegação na estrutura do documento. Todos os outros subobjectos são acedidos a partir destes por navegação. O gestor dos subobjectos é ainda responsável por controlar a persistência dos subobjectos em cada um dos servidores.

[4] Nas bases de dados orientadas para os objectos, este processo chama-se *pointer swizzling* [171]

3.2.2 Funcionamento global

Nesta subsecção apresentam-se brevemente os componentes que constituem um *coobjecto* e descreve-se o funcionamento de um *coobjecto*, explicando as interacções que se estabelecem entre os vários componentes e entre o núcleo do sistema e um *coobjecto*.

Além dos subobjectos, representantes dos subobjectos (*proxies*) e gestor de subobjectos, apresentados na subsecção anterior, o *framework* de componentes DOORS inclui os seguintes componentes:

Atributos do sistema Mantém as propriedades fundamentais do *coobjecto*/subobjecto, as quais permitem ao sistema manipulá-lo. A implementação deste componente é comum a todos os *coobjectos*/subobjectos.

Atributos Mantém um conjunto de propriedades adicionais do *coobjecto*, as quais podem ser definidas de forma específica para cada *coobjecto*.

Registo Armazena as operações executadas (submetidas) pelos utilizadores.

Reconciliação/controlo de concorrência Controla a execução das operações armazenadas no registo. No servidor, este componente é responsável por garantir que as várias réplicas de um mesmo *coobjecto* evoluem de acordo com as expectativas (não necessariamente para o mesmo estado).

Awareness Controla o processamento da informação de *awareness* produzida durante a execução das operações — em algumas situações, este componente define a estratégia base utilizada, a qual pode ser adaptada para cada utilizador.

Adaptação Controla o modo como são processadas as invocações executadas nos representantes dos subobjectos. Usando este componente é possível propagar uma invocação para execução imediata num servidor.

Cápsula Define a composição de um *coobjecto* e controla o seu funcionamento. Adicionalmente, controla a interacção de um *coobjecto* com o núcleo do sistema.

O núcleo do sistema interage com um *coobjecto* através da interface de sistema do *coobjecto* (ou simplesmente interface do *coobjecto*). Esta interface, apresentada na figura 3.3, é implementada pelo componente cápsula. Para ser mais exacto, o *interface do coobjecto* consiste em duas interfaces distintas: uma a ser implementada no servidor e outra no cliente.

3.2.2.1 Cópias de um *coobjecto*

Um *coobjecto* é representado em memória através de um conjunto de (instâncias de) componentes. No protótipo do sistema DOORS, os componentes são representados por objectos do sistema Java.

```
//=======================================================
// Atributos do sistema
//=======================================================
interface CoobjectSysAttrib {
    GOID coobjectCode();     // Coobjecto com código necessário para instanciar este coobjecto
    String factoryName();    // Fábrica usada para criar/instanciar coobjectos
}

interface SubObjectSysAttrib {
    String factoryName();    // Fábrica usada para criar/instanciar sub-objectos
}

//=======================================================
// Parte comum do interface de sistema de um coobjecto
//=======================================================
interface CommonSysCoobject {
    void initNewID( ObjectHandle oH, MbrshipInfo site);  // Inicia nova identidade
    void executeOp();                                    // Força a execução das operações (opt)
    void writeObject();                                  // Armazena coobjecto localmente
    void serialize( short type, Object info, ObjectOutputStream oos)
                                                         // Serializa coobjecto para propagação
    void coobjectRemove();                               // Coobjecto vai ser removido
}

//=======================================================
// Cliente
//=======================================================
interface ClientSysCoobject extends CommonSysCoobject {
    OperationSeq checkChanges();   // Operações executadas desde o último commit
    void changesCommitted();       // Indica que operações obtidas anteriormente
                                   // estão armazenadas estavelmente para serem propagadas
    boolean enableDirty();         // Operações associadas a múltiplas cópias existentes no cliente
    void setPolluted();
    boolean isDirty();
}

interface ClientCoobject {
    CoobjectSysAttrib sysAttrib();
    ClientSysCoobject sysInterface();
    ....
}

//=======================================================
// Servidor
//=======================================================
interface ServerSysCoobject extends CommonSysCoobject {
    Object executeQueryOp( OperationSingle op);      // Executa operação sincronamente (opt)
    ExecLoggedOperation insertOp( OperationSeq op);  // Entrega operações recebidas dum cliente
    void insertOp( LoggedOperation[] ops);           // Insere operações recebidas de outro servidor
    LoggedOperation[] checkChanges( Timevector tv);  // Obtém operações não reflectidas pelo sumário dado
    void cleanupOp();                                // Força a reciclagem das oeprações desnecessárias
    void flushAwarenessMsgs();                       // Força a propagação de mensagens de awareness
    void mbrshipChange( MbrshipChange change);       // Operações associadas com as mudanças de vista
    void preRemoveMember( int sitePos, int patPos);
    void removeMember( int sitePos, int patPos, boolean forced);
    boolean discardable( int sitePos);
}

interface ServerCoobject {
    CoobjectSysAttrib sysAttrib();
    ServerSysCoobject sysInterface();
    ....
}
```

Figura 3.3: Interface do *coobjecto*.

Os clientes e os servidores mantêm uma cópia de cada *coobjecto* que replicam em memória estável. Para tal, o sistema mantém um conjunto de recursos (ficheiros e bases de dados) associados a cada *coobjecto*. O modo como o estado de um *coobjecto* é armazenado nestes recursos pode ser redefinido por cada *coobjecto* (com excepção dos atributos do sistema). No protótipo do sistema DOORS, esta gravação consiste, por omissão, na gravação do estado dos objectos que representam os componentes e os subobjectos (cada um num ficheiro diferente).

Um *coobjecto* é sempre manipulado em memória. Assim, para manipular um *coobjecto*, é necessário criar uma cópia desse *coobjecto* (em memória)[5]. O núcleo do sistema cria esta cópia com base nas propriedades definidas nos atributos do sistema. Estas propriedades especificam a fábrica a usar na criação da cópia do *coobjecto* e a localização do código necessário. O estado do *coobjecto* é obtido a partir dos recursos associados ao *coobjecto*.

A cópia de um *coobjecto* pode ser manipulada directamente pelo núcleo do sistema ou entregue a uma aplicação. Quando apropriado, o novo estado do *coobjecto* pode ser gravado nos recursos associados ao *coobjecto*.

3.2.2.2 Funcionamento comum no cliente

Num cliente, uma aplicação manipula uma cópia privada de um *coobjecto*. Como se referiu anteriormente, esta cópia é criada pelo núcleo do sistema e entregue à aplicação para uso exclusivo.

Após obter a cópia do *coobjecto* (ou simplesmente o *coobjecto*), a aplicação deve obter uma referência (representante) de um subobjecto base. Este representante (*proxy*) é usado para aceder e modificar o estado do subobjecto e permite à aplicação navegar através dos subobjectos que constituem o estado actual do *coobjecto*, obtendo referências para outros subobjectos.

Quando uma aplicação invoca um método num representante de um subobjecto, o representante cria um objecto que codifica a invocação executada, incluindo informação sobre o método a executar e os parâmetros da invocação. O subobjecto alvo da invocação não necessita de estar instanciado nesse momento. O representante do subobjecto entrega o objecto que codifica a invocação executada ao componente de adaptação. Vários métodos estão disponíveis no componente de adaptação dependendo de: (1) o método invocado ser de leitura ou escrita[6]; e (2) a invocação ter sido efectuada directamente pela aplicação ou resultar da execução de outro método do *coobjecto* (note que um subobjecto pode manter referências para outros subobjectos e o código de um método definido num subobjecto pode invocar métodos de outros subobjectos).

O componente de adaptação processa a operação recebida do representante (*proxy*) do subobjecto

[5]Por questões de eficiência, o sistema pode manter cópias em memória do conjunto de *coobjectos* mais utilizados.

[6]Um método de leitura não produz modificações no estado do subobjecto e é apenas usado para aceder ao estado actual do subobjecto. Um método de escrita produz modificações no estado actual do subobjecto.

em função do tipo de invocação executada (leitura/escrita, directa/indirecta) implementando a política definida pelo componente. Para o processamento de uma operação existem várias possibilidades. Por exemplo, é possível propagar a operação imediatamente para ser executada num servidor (usando os serviços do núcleo do sistema). Neste caso, o resultado da invocação é o resultado obtido remotamente — se este resultado incluir uma referência (representante) de um subobjecto, esta referência é devidamente integrada na cópia do *coobjecto*[7]. Para executar a operação localmente, o componente de adaptação propaga o objecto que representa a invocação executada para a cápsula do *coobjecto*.

A cápsula controla a execução local da operação. No caso de ser uma operação de leitura ou a invocação resultar da execução de outra operação, a execução local consiste, na execução do método da cópia privada do subobjecto (o resultado obtido, quando exista, é devolvido como resultado da invocação no representante do subobjecto)[8]. No caso de ser uma operação de escrita executada directamente por uma aplicação, o processamento consiste normalmente em dois passos. Primeiro, a operação é entregue ao componente de registo. Segundo, o componente de reconciliação é notificado da presença de uma nova operação no registo — em geral, no cliente, as operações são executadas imediatamente, o que consiste em executar o método da cópia privada do subobjecto.

Para executar localmente uma operação num subobjecto, é necessário obter a cópia local do subobjecto — para tal, usa-se o gestor de subobjectos. Se o subobjecto ainda não estiver instanciado, o gestor de subobjectos encarrega-se de instanciar uma cópia privada.

A execução de uma operação pode produzir informação de *awareness*. Esta informação é criada invocando, durante a execução da operação, um método específico da classe base do subobjecto. Este método apenas reenvia a invocação para o componente de *awareness*. O componente de *awareness* processa esta informação de acordo com a política definida — em geral, no cliente, esta informação é ignorada.

Para gravar as modificações executadas, as aplicações usam uma função da API do sistema. Esta função, além de gravar a versão provisória do estado do *coobjecto* na *cache* do cliente, extrai a sequência de operações a propagar para o servidor. Esta sequência de operações é extraída através de um método definido na interface de sistema do *coobjecto* (*checkChanges*), o qual se limita a obter esta informação no componente de registo. Como se descreveu anteriormente, o componente registo mantém a sequência de todas as operações de escrita executadas pela aplicação (e que foram executadas localmente).

[7]Esta integração, executada automaticamente, consiste em ajustar, no representante, a referência para o componente de adaptação.

[8]Para situações excepcionais, definiu-se um mecanismo que permite indicar durante a execução de uma operação que se pretende registar a próxima invocação executada noutro subobjecto. Deste modo, esta invocação será propagada para todas as réplicas em conjunto com a invocação original que levou à execução da operação que está a ser executar.

3.2.2.3 Funcionamento comum no servidor

Nos servidores, os *coobjectos* são manipulados em duas situações: quando o servidor recebe operações de um cliente e durante as sessões de sincronização. Para tal, o servidor cria uma cópia do *coobjecto* (em memória), executa as acções necessárias e grava o novo estado em memória estável. De seguida descrevem-se as acções efectuadas em cada uma das situações.

Quando o servidor recebe, de um cliente, uma sequência de operações, entrega essas operações ao *coobjecto* (através da interface de sistema do *coobjecto*). No *coobjecto*, a função que recebe as operações, implementada na cápsula, executa as seguintes acções. Primeiro, entrega a sequência de operações ao componente de registo. O componente de registo atribui um identificador à sequência de operações e armazena-a. Adicionalmente, o sumário das operações[9] conhecidas, mantido no componente de atributos e usado durante o funcionamento do *coobjecto*, é actualizado. Segundo, notifica o componente de reconciliação da existência de novas operações — este componente é responsável por executar as operações armazenadas no componente de registo de acordo com a política definida (garantindo que as várias réplicas de um mesmo *coobjecto* evoluem da forma esperada). A execução das operações é efectuada de forma idêntica à execução local de operações no cliente (e pode originar a produção e tratamento de informação de *awareness* ou a invocação de outras operações de outros subobjectos — estas invocações são processadas executando imediatamente as operações nos subobjectos respectivos).

Durante as sessões de sincronização os *coobjectos* podem ser acedidos em três situações: (1) para transmitir o sumário das operações conhecidas e a informação sobre as operações conhecidas noutros servidores; (2) para enviar o conjunto de operações desconhecidas no parceiro; (3) para receber um conjunto de operações desconhecidas do parceiro.

O sumário das operações conhecidas e a informação sobre as operações conhecidas nos outros servidores está guardada nos atributos do *coobjecto*. Para obter esta informação, o servidor não necessita de criar uma cópia do *coobjecto* — a API do sistema permite obter apenas os atributos de um *coobjecto*.

O conjunto de operações que um servidor conhece, mas que o parceiro desconhece, é obtido através da interface do *coobjecto* (*checkChanges*). Esta função obtém o conjunto de operações a enviar para o parceiro a partir do componente de registo. Nesta situação, o servidor utiliza a informação sobre o estado do parceiro (operações conhecidas e operações que o parceiro sabe que os outros servidores conhecem) para actualizar a sua informação local.

Quando um servidor recebe um conjunto de operações de outro servidor, esse conjunto de operações é entregue ao *coobjecto* através da interface do *coobjecto* (*insertOp*). Esta função efectua as seguintes operações. Primeiro, entrega as operações ao componente de registo, o qual as armazena. Segundo,

[9] O identificador de uma operação e os sumários de operações usados no sistema DOORS são baseados em relógios lógicos e vectoriais, como se detalha na secção 4.1.1.

informa o componente de reconciliação que existem novas operações no componente de registo. Como anteriormente, o servidor utiliza a informação sobre o estado do parceiro para actualizar a sua informação local.

Um *coobjecto* pode ainda ser acedido durante os protocolos de mudança de filiação (do conjunto de servidores que replicam um volume). Durante este processo, e de forma semelhante às situações anteriores, o núcleo do sistema limita-se a executar as operações respectivas na interface do *coobjecto*. Estas operações, implementadas na cápsula, acedem aos vários componentes do *coobjecto* para executarem as operações respectivas (por exemplo, para determinar se um servidor pode ser removido é necessário verificar se o componente de reconciliação já não necessita da informação sobre as operações submetidas nesse servidor).

3.2.2.4 Operações definidas nos componentes

A descrição anterior omitiu a possibilidade de definir, nos vários componentes, operações que devam ser processadas de forma semelhante às operações dos subobjectos, i.e., registando a sua invocação e propagando estas invocações para todas as réplicas. Por exemplo, a remoção de um *coobjecto* é controlada por uma operação definida no componente de atributos (ver detalhes na secção 6.1.3).

Esta funcionalidade é implementada por pré-processamento do código dos componentes. Para cada operação que deve ser registada (e propagada para as várias réplicas) são criados dois métodos: um método que se limita a registar a operação (de forma idêntica aos métodos definidos num representante de um subobjecto) e um método "sombra" com o código da operação.

Todas as outras funções definidas nos componentes são processadas como normais funções de um objecto e afectam apenas o estado da cópia na qual são executadas. Por exemplo, a actualização do sumário das operações conhecidas localmente efectuada durante uma sessão de sincronização apenas tem efeito sobre a cópia local do *coobjecto*.

3.2.2.5 Atributos de uma operação

Até ao momento descreveu-se o processamento comum das operações definidas nos subobjectos (e nos outros componentes de um *coobjecto*) cuja invocação é interceptada no âmbito do funcionamento de um *coobjecto*. Este processamento comum pode ser modificado através dum conjunto de atributos que o programador pode associar a uma operação quando define o código do subobjecto/*coobjecto* — este código é pré-processado, como se descreve na secção 3.4.1, de forma a que as operações sejam executadas da forma especificada. No sistema DOORS estão actualmente definidos os seguintes atributos:

Leitura Indica que a operação não modifica o estado do *coobjecto*. Estas operações não necessitam de ser registadas.

Escrita Indica que a operação modifica o estado do *coobjecto*. Estas operações devem ser registadas e propagadas para todas as réplicas.

Sistema As operações de sistema são usadas no funcionamento interno do *coobjecto*. Estas operações não são reflectidas no sumário das operações invocadas e são executadas numa réplica assim que são recebidas.

Último escritor ganha Indica que, independentemente da política de reconciliação geral usada, apenas deve ser executada uma invocação deste método, caso não tenha sido executada anteriormente uma invocação posterior (na ordem total definida pelos identificadores das operações)[10]. Caso tenha sido executada uma operação posterior, a execução deste método é equivalente à execução da operação nula (i.e., não modifica o estado do *coobjecto*). Este atributo é geralmente usado apenas com as operações de sistema.

Remota Indica que a operação deve ser executada imediatamente num servidor.

Sem *desfazer-refazer* Indica que a operação só deve ser executada uma vez, mesmo que o mecanismo de reconciliação use técnicas de *desfazer-refazer* (*undo-redo*) [17].

Silencioso e local Estes atributos são usados para controlar o funcionamento do mecanismo de invocação cega, descrito em detalhe na secção 4.3.

Os componentes do *coobjecto* responsáveis pelo processamento de uma invocação devem ter em conta estes atributos.

3.3 *Framework* de componentes: componentes

Nesta secção detalham-se os vários componentes do *framework* DOORS. Para cada um, descreve-se a sua funcionalidade e discutem-se algumas das possíveis implementações, focando os componentes pré-definidos no protótipo do sistema DOORS.

3.3.1 Atributos do sistema

Os atributos do sistema mantêm as propriedades fundamentais para a manipulação de um *coobjecto*/subobjecto pelo núcleo do sistema. Existe um componente de atributos do sistema associado ao *coobjecto* e a cada um dos subobjectos.

[10] Para implementar esta propriedade, o pré-processador adiciona ao código da operação: a gravação do identificador da (última) invocação executada; e a verificação se a invocação a executar é posterior à última executada.

Estas propriedades incluem a identificação do *coobjecto* que contém o código necessário para instanciar o *coobjecto*/subobjecto[11] (caso não pertença a um dos tipos básicos do sistema DOORS) e o nome da classe a usar como fábrica. Esta informação permite ao núcleo do sistema criar e manipular qualquer tipo de *coobjecto* de forma uniforme.

Os atributos do sistema mantém ainda um sumário das operações executadas. Para os atributos do sistema associados a um *coobjecto*, este valor reflecte as operações executadas em qualquer subobjecto. Para os atributos do sistema associados a um subobjecto, este valor reflecte as operações executadas que modificaram (directa ou indirectamente) esse subobjecto — este sumário é actualizado durante a execução das operações.

Adicionalmente, os atributos do sistema incluem uma lista de identificadores únicos que indicam *coobjectos* e subobjectos que devem ser replicados conjuntamente durante a replicação prévia. Para um *coobjecto*, esta lista inclui identificadores de *coobjectos*. Para um subobjecto, esta lista inclui identificadores de subobjectos.

Ao contrário de todos os outros componentes de um *coobjecto* que podem ter uma implementação específica (desde que implementem a interface do componente), os atributos do sistema têm uma definição fixa para permitir ao sistema obter o tipo do *coobjecto*.

3.3.2 Atributos

O componente de atributos mantém um conjunto de propriedades do *coobjecto*. O sistema DOORS fornece uma função que permite às aplicações aceder a este componente (e ao componente de atributos do sistema) para leitura de forma "leve", i.e., sem terem de obter uma cópia do *coobjecto*.

Existem dois tipos de propriedades. Primeiro, as propriedades do sistema, usadas pelo sistema DOORS no seu funcionamento — por exemplo, os sumários das operações conhecidas localmente e nos outros servidores, a informação sobre o estado de remoção do *coobjecto*, etc. Segundo, as propriedades específicas para cada tipo de *coobjecto*. Estas propriedades são definidas livremente pelos programadores que implementam o *coobjecto* e devem representar características fundamentais do *coobjecto*, às quais pode ser interessante aceder sem criar uma cópia do *coobjecto*.

Componentes pré-definidos No protótipo do sistema DOORS definiram-se dois componentes de atributos base, cuja diferença reside no espaço usado ($O(n)$ e $O(n^2)$, com n o número de servidores que replicam o *coobjecto*) para registar a informação relativa às operações conhecidas nos outros servidores (as duas técnicas são detalhadas na secção 6.2.2). A interface do componente de atributos define operações que permitem manipular esta informação de forma uniforme.

[11] O sistema DOORS define um tipo básico de *coobjectos* para manter o código de outros *coobjectos*. As instâncias deste *coobjecto* são imutáveis.

Os componentes de atributos base foram estendidos para incluir as operações de manutenção da identificação de uma réplica primária. Estes novos componentes devem ser usados sempre que se use técnicas de reconciliação baseadas num sequenciador.

Os atributos específicos de cada *coobjecto* são guardados numa tabela que associa nomes simbólicos a objectos. A operação de modificação utiliza, para cada nome simbólico, uma estratégia de "o último escritor ganha". Para tal, foi necessário estender a aproximação "o último escritor ganha" usada para as operações (secção 3.2.2.5). Sempre que esta funcionalidade genérica de manutenção de propriedades específicas não seja adequada, um programador pode estender um componente base para definir uma nova propriedade específica para o *coobjecto* que cria.

3.3.3 Registo

O componente de registo armazena as operações de escrita executadas. Em geral, um *coobjecto* usa diferentes registos no cliente e no servidor. No cliente, o registo limita-se a armazenar a sequência de operações executadas localmente pela aplicação. No servidor, o registo armazena as operações submetidas em todos os servidores.

O registo é ainda responsável por adicionar a seguinte informação para ordenar e traçar as dependências entre as operações, como se detalha na secção 4.1.1. No cliente, a cada sequência de operações executadas é adicionado o sumário das operações reflectida no estado actual — este sumário permite determinar exactamente as dependências de uma operação, e detectar as operações executadas concorrentemente. No servidor, a cada sequência de operações recebida de um cliente é atribuído um identificador único (composto pelo identificador da réplica e por uma estampilha temporal).

Componentes pré-definidos No protótipo do sistema DOORS definiram-se componentes base diferentes para o servidor e para o cliente. Como explicado anteriormente, estes componentes base adicionam às operações informação que permite ordenar e traçar as dependências entre as operações.

Para o servidor, criou-se um componente base que gere o conjunto de operações recebidas em cada servidor. Criou-se ainda um segundo componente base que gere adicionalmente as operações ordenadas por um sequenciador.

Para o cliente, criou-se apenas um componente base que mantém a lista das operações executadas pelos utilizadores. Este componente executa um algoritmo de compressão de operações usando as propriedades definidas nas mesmas (este algoritmo usa a aproximação descrita na secção 10.4.2).

3.3.4 Reconciliação

O componente de reconciliação (ou controlo de concorrência) é responsável por executar as operações armazenadas no componente de registo. Em geral, um *coobjecto* usa diferentes componentes de reconciliação no cliente e no servidor. No cliente, o componente de reconciliação apenas controla a execução imediata das operações de forma a que os utilizadores possam observar o resultado esperado (provisório) das suas acções.

No servidor, este componente controla a execução das operações em cada réplica do *coobjecto*. Em geral, pretende-se que as várias cópias de um mesmo *coobjecto* evoluam para estados consistentes que reflictam as modificações executadas pelos utilizadores. Como o sistema DOORS permite que vários utilizadores modifiquem concorrentemente o mesmo *coobjecto*, este componente deve reconciliar os fluxos de actividade (possivelmente) divergentes executados por vários utilizadores.

Este componente controla igualmente a execução de blocos “apenas-uma-vez” (*once-only*) — um bloco *once-only* define um bloco de instruções no código de uma operação que apenas deve ser executado uma vez, independentemente do número de cópias do *coobjecto* existentes (em caso da falha definitiva de um servidor, a semântica exacta implementada deve ser: “no máximo uma vez”). Este blocos podem ser usados para executar acções que tenham reflexo no mundo exterior ao *coobjecto* (por exemplo, a execução de uma operação noutro *coobjecto*).

Como se discutiu na secção 2.3.2, várias aproximações podem ser usadas para controlar a evolução das várias réplicas e garantir que as intenções dos utilizadores são respeitadas. Em geral, estas aproximações baseiam-se na execução das operações em todas as réplicas por uma determinada ordem explorando a informação semântica associada com os tipos de dados e com as operações executadas (por exemplo, verificando pré-condições e/ou explorando alternativas expressas pelos utilizadores).

Na secção 4.1 descrevem-se os componentes de reconciliação implementados no protótipo do sistema DOORS e discutem-se as suas propriedades (assim como as propriedades de outras técnicas de reconciliação que poderiam ser também implementadas).

Entre os componentes implementados, podem-se referir aqueles que permitem executar as operações em todas as réplicas segundo uma ordem causal ou segundo uma ordem total (recorrendo a técnicas pessimistas ou optimistas). Também foi criado um componente que permite manter as intenções dos utilizadores recorrendo à transformação de operações [159].

3.3.5 *Awareness*

O componente de *awareness* controla o processamento da informação de *awareness* produzida durante a execução das operações. Como se discutiu na secção 2.3.4, existem duas aproximações principais para processar esta informação. A primeira consiste em manter esta informação com o *coobjecto* de forma

a que possa ser apresentada nas aplicações. A segunda consiste em propagar esta informação para os utilizadores. Um *coobjecto* deve usar a aproximação apropriada para o tipo de dados que implementa, podendo igualmente combinar a utilização de ambas. Os utilizadores devem poder adaptar a aproximação usada às suas necessidades. Em particular, deve ser possível controlar a informação recebida e o modo de recepção.

Quando se mantém a informação de *awareness* com o *coobjecto*, esta informação deve estar disponível no cliente para ser apresentada nas aplicações. Assim, o componente de *awareness* respectivo deve ser usado no cliente e no servidor. A informação de *awareness* é actualizada definitivamente no servidor, podendo ser igualmente actualizada, de forma provisória, no cliente. Um aspecto a realçar é que esta informação é mantida de forma independente para cada subobjecto, pelo que no cliente apenas se mantém a informação relativa aos subobjectos replicados localmente.

Na propagação da informação de *awareness*, as possibilidades disponíveis no cliente e no servidor são diferentes. No cliente apenas é possível propagar a informação através do serviço de disseminação de eventos Deeds [45] — no entanto, em geral, como a informação produzida corresponde à execução provisória das operações, a informação não é propagada. No servidor é possível utilizar o serviço de *awareness* disponibilizado pelo núcleo do sistema (ver detalhes na secção 6.2.7) para propagar mensagens através dos vários transportes disponíveis. Em geral, quando se usam técnicas de reconciliação optimista, a propagação das mensagens apenas é executada após determinar que a informação produzida por uma operação corresponde ao resultado final da sua execução.

Componentes pré-definidos Para implementar as características anteriores, no protótipo do sistema DOORS foram criados vários componentes de *awareness* com diferentes funcionalidades.

Primeiro, um componente que mantém a informação de *awareness* como uma lista de mensagens. As mensagens relativas a cada subobjecto são mantidas num *subcomponente* independente associado ao subobjecto.

Segundo, um componente que propaga as mensagens de *awareness*. Este componente apenas entrega as mensagens ao serviço de *awareness* do sistema DOORS, o qual se encarrega de as entregar ao utilizador através do transporte indicado.

Terceiro, componentes auxiliares dos quais se destacam os seguintes. Um componente que permite combinar dois outros componentes, sendo as mensagens entregues a ambos. Um componente que descarta todas as mensagens e que pode ser usado quando não se pretende processar as mensagens de *awareness*. Um componente que permite propagar apenas uma mensagem relativa a cada operação — a implementação consiste em propagar a mensagem num bloco *once-only*. Um componente que permite propagar para outro apenas mensagens produzidas em operações estáveis. Para tal, este componente armazena as mensagens de *awareness* recebidas, apenas as propagando para o outro componente quando

as mesma se tornam estáveis (de acordo com o componente de reconciliação usado). Para que apenas se propaguem mensagens produzidas na execução final de uma operação, sempre que uma operação é desfeita, o componente de *awareness* (informado pelo componente de reconciliação desse facto) descarta as mensagens relativas a essa operação.

3.3.6 Adaptação

O componente de adaptação controla o processamento da invocação das operações (executada num representante de um subobjecto ou numa operação definida noutro componente que deva ser registada). Um *coobjecto* deve usar diferentes componentes de adaptação no cliente e no servidor.

No servidor, as invocações processadas são geralmente resultado da execução de outra operação (sob controlo do componente de reconciliação). Neste caso, apenas é necessário executar imediatamente a operação respectiva.

No cliente é possível utilizar diferentes aproximações para processar as invocações, de entre as quais se destacam as seguintes. Primeiro, é possível executar localmente todas as invocações (que não sejam explicitamente indicadas como de execução remota), fazendo reflectir as modificações na cópia do *coobjecto* que está a ser manipulada pela aplicação. Segundo, é possível propagar todas as invocações executadas para um servidor. Desta forma, acede-se directamente a uma cópia do *coobjecto* presente num servidor usando uma estratégia semelhante à execução remota de procedimentos.

Terceiro, é possível combinar a execução local com a execução remota em função das condições de conectividade. Esta possibilidade coloca algumas questões relativas à consistência entre a cópia local e remota que devem merecer atenção. Em geral, pretende-se que as aplicações observem uma evolução consistente do estado do *coobjecto*, i.e., que o resultado de uma operação seja obtido sempre numa cópia do *coobjecto* em que tenham sido executadas todas as operações reflectidas na cópia do *coobjecto* em que as operações anteriores foram executadas. Para tal, é necessário impor restrições aos servidores usados e/ou actualizar a cópia local com as novas modificações conhecidas na réplica remota.

Componentes pré-definidos No protótipo do sistema DOORS foram criados componentes para utilização no servidor e no cliente. Relativamente ao servidor, o componente de adaptação criado limita-se a executar localmente todas as operações. Relativamente aos componentes a utilizar no cliente foram implementadas as estratégias básicas descritas anteriormente, i.e., executar todas as operações na cópia local ou executar todas as operações imediatamente num servidor[12]. Para executar remotamente uma operação, o componente de adaptação usa o serviço disponibilizado pelo núcleo do sistema.

[12] O componente de execução local executa imediatamente num servidor as operações marcadas como remotas. Neste caso, não se garante a consistência entre os estados observados em invocações consecutivas.

3.3.7 Cápsula

A cápsula define a composição de um *coobjecto* e controla o seu funcionamento, definindo a forma como são processadas localmente as invocações executadas pelas aplicações (e que o componente de adaptação decide executar localmente). Adicionalmente, implementa as operações definidas na interface de sistema do *coobjecto*, definindo o modo como são processadas as operações executadas pelo núcleo do sistema.

O processamento comum no cliente e no servidor foi descrito, respectivamente, nas secções 3.2.2.2 e 3.2.2.3 e limita-se a combinar a execução de operações definidas nos outros componentes.

Ao controlar o funcionamento de um *coobjecto* e definir a sua composição, a cápsula permite criar *coobjectos* com diferentes composições como se verá de seguida.

Componentes pré-definidos Dois componentes cápsula foram implementados. O primeiro, a cápsula simples, mantém uma única versão do estado do *coobjecto*. Para tal, agrega um componente de cada tipo: atributos do sistema, atributos, registo, reconciliação, *awareness*, adaptação e gestor de subobjectos. Este componente permite criar um *coobjecto* comum.

O segundo componente cápsula, a cápsula dupla, mantém dois estados do *coobjecto*. Para tal, inclui dois componentes de reconciliação, dois componentes de *awareness* e dois gestores de subobjectos. Os dois gestores de subobjectos permitem manter duas versões de cada subobjecto, os quais são actualizados de acordo com as (possivelmente) diferentes políticas de reconciliação definidas para cada um. Sempre que existe uma nova operação no componente de registo, as funções definidas na cápsula notificam ambos os componentes de reconciliação para executarem a operação na versão respectiva.

Usando esta cápsula é possível criar facilmente um *coobjecto* que mantenha uma versão provisória e uma versão definitiva do seu estado (i.e. dos seus subobjectos) a partir de subobjectos comuns, executando as operações armazenadas no componente de registo por uma (mesma) ordem total optimista e pessimista, respectivamente.

Os dois componentes de *awareness* permitem tratar de forma diferente a informação produzida provisória e definitivamente (em geral, a informação produzida provisoriamente é descartada). A cápsula encarrega-se de garantir que as operações são executadas de forma independente em cada uma das versões e que as operações relativas a componentes não duplicados (por exemplo, ao componente de atributos) apenas são executadas na versão definitiva. Para tal, cria, para cada versão, um *coobjecto* comum virtual composto pelos elementos respectivos.

3.3.8 Gestor de subobjectos

O gestor de subobjectos controla a criação e remoção dos subobjectos contidos num *coobjecto*, executando as seguintes funções. Primeiro, controla a criação das cópias em memória dos subobjectos e

mantém referências para todos os subobjectos criados. Segundo, mantém referências para um conjunto de subobjectos raiz (acessíveis através de um nome). As aplicações navegam nos subobjectos contidos num *coobjecto* a partir destes subobjectos raiz. Finalmente, controla a persistência dos subobjectos no servidor. Para esta função duas técnicas básicas são geralmente adoptadas: a reciclagem automática (*garbage-collection*), em que um subobjecto é removido quando ele se torna inacessível a partir dos subobjectos raiz; e a remoção explícita, em que um subobjecto é removido pela execução explícita de uma operação de remoção.

Componentes pré-definidos No protótipo do sistema DOORS, implementou-se um gestor de subobjectos que combina a reciclagem automática com a remoção explícita de forma simples. Periodicamente, em cada servidor, determina-se o conjunto de subobjectos que não pode ser alcançado a partir dos subobjectos raiz. Estes subobjectos são marcados como removidos (os outros são marcados como activos) mas não são imediatamente eliminados. Assim, um subobjecto removido pode passar a activo se alguma operação não executada no servidor tornar esse subobjecto acessível novamente.

O processo de eliminação de um subobjecto inicia-se, em qualquer réplica, quando o sistema necessita de espaço livre ou um subobjecto permanece removido durante um período de tempo grande (actualmente, 30 dias). Quando uma das condições anteriores se verifica submete-se a operação de eliminação definida no gestor de subobjectos (a qual é propagada para todas as réplicas e executada de acordo com a política de reconciliação definida no *coobjecto*). A execução desta operação elimina o subobjecto. Esta aproximação garante que todas as réplicas eliminam o mesmo conjunto de subobjectos.

O bom funcionamento desta estratégia baseia-se no pressuposto que, quando se emite a operação de eliminação de um subobjecto, todas as operações que o podem tornar acessível já foram executadas em todos os servidores. A provável satisfação desta propriedade é consequência do período de tempo longo que separa a decisão de eliminar um subobjecto da detecção inicial da sua inacessibilidade (e em consequência da impossibilidade de executar operações que se refiram a esse subobjecto). Como se referiu, este período de tempo pode ser encurtado pela necessidade de um servidor obter espaço livre. No entanto, pressupõe-se igualmente que o espaço de armazenamento disponível em cada servidor é suficientemente grande para que a propriedade inicial seja satisfeita (o sistema de ficheiros Elephant [148] usa um pressuposto semelhante).

Quando uma operação torna acessível um subobjecto eliminado anteriormente, o gestor de subobjectos pode criar uma nova cópia do subobjecto. Para tal, o programador deve especificar, quando cria o subobjecto, o modo como esta cópia é criada (de forma semelhante às cópias de substituição no mecanismo de invocação cega descrito na secção 4.3).

Esta implementação do gestor de subobjectos não lida com os problemas da execução de operações em subobjectos não acessíveis a partir de uma raiz ou mesmo de subobjectos já eliminados. Assim,

deve definir-se a solução apropriada à aplicação que se está a criar no âmbito da implementação dos subobjectos. A criação de um gestor de subobjectos que execute uma política pré-definida nesta situação é igualmente possível. No entanto, este gestor de subobjectos obrigaria a que a informação mantida sobre a possibilidade de alcançar um subobjecto fosse exacta em todos os momentos, o que não acontece na implementação simples existente actualmente (em que esta detecção é efectuada periodicamente de forma ingénua).

3.3.9 Subobjectos

Os subobjectos mantêm o estado de um *coobjecto* e definem as operações para ler e modificar esse estado. Assim, os subobjectos que compõem cada *coobjecto* e a forma como se interligam internamente tendem a ser diferentes para cada *coobjecto*.

No entanto, o tratamento das modificações concorrentes pode ser efectuado através da cooperação entre o componente de reconciliação e o código das operações definidas nos subobjectos. Por exemplo, é possível definir um subobjecto que mantenha múltiplas versões de um elemento (opaco para o subobjecto). Neste caso, vários *coobjectos* podem usar versões especializadas do mesmo subobjecto base.

Componentes pré-definidos No protótipo do sistema DOORS foram criados alguns subobjectos base que podem ser especializados para utilização em diferentes *coobjectos*.

Um dos subobjectos base criados permite manter informação sobre o estado de remoção do subobjecto. Para tal, o subobjecto mantém uma variável que define o seu estado: removido ou activo. Adicionalmente, definem-se duas operações de modificação: remoção e restauro que colocam o estado do subobjecto em removido e activo respectivamente. O método de remoção grava, adicionalmente, a informação de ordenação (o identificador da operação e o sumário do estado do *coobjecto* aquando da submissão da operação) associada à operação que causa a sua execução. O código destas operações implementam conjuntamente a semântica "o último escritor ganha", garantindo que as várias réplicas do subobjecto convergem para o mesmo estado independentemente do componente de reconciliação.

Esta informação é independente da informação mantida pelo gestor de subobjectos para controlar a persistência dos subobjectos e pode ser usada para implementar estratégias de remoção específicas para uma dada aplicação.

Um segundo subobjecto base estende o subobjecto base anterior implementando a seguinte política de resolução de conflitos modificação/remoção: sempre que um subobjecto removido é modificado, ele é restaurado e adicionado ao conjunto de subobjectos raiz com um nome especial. Para tal, o subobjecto base define uma operação que verifica se o subobjecto se encontra removido, e, caso isso aconteça, executa a operação de restauro (localmente). Todas as operações de modificação definidas nos subobjectos

derivados deste subobjecto devem iniciar o seu código invocando este método. A operação de restauro, além de colocar o estado do subobjecto em activo, insere o subobjecto no conjunto de subobjectos raiz.

Um terceiro subobjectos base mantém um objecto (opaco) com múltiplas versões. Este subobjecto define duas operações básicas que alteram o seu estado: criar e remover uma versão. Internamente cada versão é imutável e é identificada por um identificador único. A modificação de uma versão do objecto é implementada removendo a versão antiga e criando uma nova versão. Esta aproximação leva à inexistência de conflitos entre operações executadas concorrentemente. Por exemplo, duas modificações concorrentes de uma mesma versão levam a uma dupla remoção da versão antiga (a segunda remoção é ignorada) e à criação de duas novas versões. Este subobjecto garante que as suas várias réplicas convergem para um estado comum desde que as operações sejam executadas nos servidores respeitando a ordem causal. Este subobjecto base pode ser especializado de forma a definir o tipo de objectos armazenados.

O quarto subobjecto base criado mantém uma lista de referências únicas para outros subobjectos, i.e., em cada lista de referências não existe nenhum elemento repetido. Duas operações básicas foram definidas: inserir e remover uma referência para um subobjecto. Adicionalmente, definiu-se uma operação para mover uma referência. A execução desta operação remove a referência no subobjecto em que executa e (em caso de sucesso na remoção) adiciona a referência no mesmo ou noutro subobjecto. A operação de remoção remove a referência indicada (quando ela existe) independentemente da sua posição na lista. A operação de inserção indica a posição em que a referência deve ser inserida. A existência de operações submetidas concorrentemente pode levar a que a posição de inserção de uma referência seja diferente da desejada, se o componente de reconciliação não executar um algoritmo de transformação de operações, como discutido na secção 4.1 (as operações incluem o código necessário para suportar esse algoritmo).

Para que as várias réplicas de um subobjecto convirjam para o mesmo estado é necessário executar todas as operações pela mesma ordem em todas as réplicas. O código das operações não efectua nenhum tratamento especial para as operações submetidas concorrentemente. Assim, para operações que actuam sobre referências diferentes, o efeito de todas as operações é reflectido no estado final do subobjecto. Relativamente às operações que actuam sobre uma mesma referência, a ordem de execução das operações determina o seu efeito no estado final do subobjecto.

O subobjecto lista de referências pode ser especializado para definir o tipo de referências que podem ser adicionadas.

O quinto subobjecto base implementa a interface de uma base de dados relacional SQL. Assim, este subobjecto define: uma operação de *leitura*, que permite executar interrogações SQL; e uma operação de *escrita*, que permite executar as instruções de modificação SQL. Para armazenar os dados e executar as invocações, este subobjecto utiliza uma base de dados gerida pelo núcleo do sistema, como um dos recur-

sos associado ao *coobjecto*. Este subobjecto permite a uma aplicação usar dados organizados segundo o modelo relacional e mostra que o sistema DOORS pode funcionar como um sistema de replicação genérico[13].

3.3.10 Definição de um *coobjecto*: exemplo

Nesta secção exemplifica-se a utilização do *framework* de componentes através da descrição de um *coobjecto* simples que mantém a associação entre um nome simbólico, definido num espaço de nomes hierárquico semelhante ao espaço de nomes de um sistema de ficheiros, e um identificador único.

O primeiro passo consiste em definir os subobjectos usados no *coobjecto*. Neste caso definiram-se dois subobjectos: um subobjecto referência, imutável, que apenas armazena um identificador único; e um subobjecto directório, que associa um nome simples com um subobjecto. A hierarquia de nomes é construída, a partir do subobjecto directório raiz, associando nomes simples a subobjectos directório (de forma semelhante ao usado num sistema de ficheiros).

O subobjecto directório define duas operações básicas. A primeira cria a associação entre um nome e um subobjecto. Se, quando a operação é executada, o nome já está associado a um subobjecto, é criado um novo nome (de forma determinista) e efectuada a associação entre o novo nome e o subobjecto. A segunda operação remove o nome associado a um subobjecto. Se, quando a operação é executada, não existir, no subobjecto, nenhum nome simples associado ao subobjecto indicado, o espaço de nomes é percorrido para remover a associação de qualquer nome ao subobjecto indicado. A remoção de um nome associado a um directório apenas é efectuada se, quando a operação é executada, o directório se encontrar vazio (i.e., não contiver nenhuma associação definida).

Usando as operações básicas anteriores, é possível modificar o nome associado a um subobjecto (referência ou directório) removendo a associação antiga e criando uma nova associação (em caso de sucesso da remoção). Neste caso, não se aplicam as restrições anteriores à remoção de um nome associado a um directório.

O segundo passo consiste em definir a composição do *coobjecto* no cliente e no servidor. Como estamos na presença de um *coobjecto* simples, ele inclui as implementações mais simples dos vários componentes para o cliente.

No servidor, queremos que as várias réplicas evoluam para um mesmo estado comum e que, em

[13]Note que esta aproximação difere da usada nos protocolos que serializam a execução concorrente de transacções num conjunto de réplicas recorrendo a um sistema de comunicação em grupo [83, 2]. Nestes sistemas, a execução de uma transacção é iniciada imediatamente na réplica local, podendo ser posteriormente abortada caso se verifique a impossibilidade de serializar a execução das várias transacções concorrente. No sistema DOORS, cada operação do subobjecto define uma transacção, a qual é executada em todas as réplicas pela ordem definida pelo componente de reconciliação (se o componente de reconciliação impuser uma ordem total, todas as réplicas executam a mesma sequência de transacções).

cada momento, apresentem um resultado provisório que reflicta todas as operações conhecidas. Assim, usa-se um componente de reconciliação que implementa uma ordem total optimista usando a técnica de *desfazer-refazer*. Como apenas queremos manter uma cópia dos dados, usamos uma cápsula simples. Como não se pretende processar a informação de *awareness*, usa-se o componente de *awareness* que descarta todas as mensagens produzidas. Para os outros componentes usou-se a implementação simples do servidor. O gestor de subobjectos mantém referência para um subobjecto directório, tratado como a raiz do espaço de nomes.

Assumindo que não é possível gerar dois subobjectos referência com o mesmo identificador único, podem surgir os seguintes conflitos na manipulação concorrente deste *coobjecto*.

Primeiro, é possível associar dois subobjectos diferentes ao mesmo nome. Este conflito é solucionado gerando um nome alternativo para um dos subobjectos. A execução das operações por ordem total garante a convergência final de todas as réplicas (apesar de, momentaneamente, o mesmo nome poder estar associado a diferentes subobjectos em diferentes réplicas e de, numa réplica, o nome associado a um dado subobjecto poder mudar).

Segundo, a remoção concorrente do mesmo subobjecto por dois utilizadores leva, como esperado, à remoção de qualquer associação entre um nome e o referido subobjecto.

Terceiro, a remoção de um directório (i.e., da associação entre um nome e um subobjecto directório) e a inserção concorrente de uma nova associação nesse directório é tratada da seguinte forma. Se a operação de inserção é ordenada antes da operação de remoção (na ordem total de execução das operações), a operação de remoção não terá sucesso porque o directório não está vazio quando a operação é executada. Neste caso a inserção de um novo nome é preservada e o directório manterá o nome anterior. Se a operação de remoção é ordenada antes da operação de inserção, quando se executa a inserção, o subobjecto já não pode ser alcançado a partir de um subobjecto raiz. Assim, apesar de a inserção poder ser executada, o seu efeito não seria visível porque o nome seria inserido num directório que já não existia. Para lidar com este problema usa-se a solução executada pelo segundo subobjecto base descrito na secção 3.3.9: o directório é recriado e adicionado ao conjunto de subobjectos raiz. Para tal, o subobjecto directório é uma especialização desse subobjecto base. Quando o nome associado a um directório é removido com sucesso, executa-se igualmente a operação de remoção definida no subobjecto base desse directório.

3.4 *Framework* de componentes: implementação

No protótipo do sistema DOORS, o *framework* de componentes foi implementado na linguagem Java como um conjunto de interfaces e classes.

Para cada tipo de componente, existe uma interface que define o conjunto mínimo de métodos a implementar. Cada componente deve funcionar como uma caixa preta e apenas ser acedido através da

interface definida. Esta aproximação permite criar um novo componente de um dado tipo de forma independente, i.e., sem conhecer a implementação dos componentes com os quais ele interage.

Em alguns componentes utilizaram-se classes base abstractas para definir a interface do componente (em vez de interfaces Java). Esta opção tem duas vantagens. Primeiro, permite criar definições típicas de alguns métodos. Segundo, permite implementar de forma definitiva os métodos que executam funções críticas para o sistema DOORS, como por exemplo, os métodos que lidam com a representação dos *coobjectos* e subobjectos usada no núcleo do sistema (por exemplo, os identificadores). Esta aproximação garante que o mau funcionamento de um *coobjecto* não corrompe outros *coobjectos*.

Sempre que necessário, um programador pode estender a interface básica definida para um dado componente com novos métodos. A nova interface estendida pode ser usada, não só na criação de novos componentes desse tipo, mas também na criação de outros componentes que necessitem de utilizar as novas funcionalidades disponibilizadas pelos novos métodos. Neste caso, um componente que requeira as novas funcionalidades apenas pode ser composto com um componente que implemente a interface estendida, impondo assim uma restrição nas composições possíveis.

Um *coobjecto* é definido como uma composição de componentes definidos de acordo com o *framework* de componentes DOORS. No protótipo do sistema DOORS, o código de criação e ligação dos vários componentes deve ser criado pelo programador como parte do código do componente cápsula. Como se disse anteriormente, as ligações entre os vários componentes devem respeitar as interfaces esperadas por cada um dos componentes.

No âmbito do projecto DataBricks [36] estão-se a investigar formas alternativas de definição dos *coobjectos*. Assim, recorrendo a uma linguagem de composição de componentes, o ComponentJ [153], a composição e ligação dos vários componentes de um *coobjecto* deve ser efectuada exteriormente ao código dos componentes do *coobjecto*. Esta linguagem deve permitir ainda verificar estaticamente a compatibilidade das ligações entre os vários componentes. Finalmente, deve ser possível criar esqueletos (*templates*) de *coobjectos*. Um esqueleto de um *coobjecto* define uma composição típica para um *coobjecto* no qual ficam por definir um ou mais componentes (em geral, os subobjectos). Assim, um programador pode criar um *coobjecto*, a partir dos subobjectos que pretende gerir, usando o esqueleto com a semântica desejada. Este trabalho está, no entanto, fora do âmbito desta dissertação.

3.4.1 Pré-processador

Actualmente, o protótipo do sistema DOORS inclui apenas um pré-processador para simplificar a criação de componentes e subobjectos. Os programadores definem um componente ou um subobjecto como uma classe na linguagem Java estendida com os seguintes elementos:

- Um conjunto de qualificadores de métodos (descritos na secção 3.2.2.5) que definem o tipo de

operação criada.

- Uma construção que permite associar a cada operação definida (método), um conjunto de métodos adicionais associados — por exemplo, um método que desfaça a execução da operação (a ser usado com estratégias de reconciliação de *desfazer-refazer*).

- Uma construção para a definição de blocos *só-uma-vez* (*once-only*).

A partir da definição de um subobjecto com o nome *X*, o pré-processador cria: uma interface com o nome *X* que define os métodos do subobjecto; o subobjecto, definido como uma classe Java que implementa a interface *X* e contém o código dos métodos; o representante do subobjecto, definido como uma classe Java que implementa a interface *X* e controla a intercepção das operações; e a fábrica do subobjecto, definida como uma classe Java. Para um componente, o pré-processador cria uma classe Java com o mesmo nome e que inclui o código necessário para efectuar a intercepção das operações (para cada operação cria-se um método com o código de intercepção e um método com o código da operação).

O código dos métodos definidos no subobjecto é modificado de forma a implementar a semântica associada aos qualificadores usados (se algum). As definições dos métodos associados às operações (por exemplo, o método *desfazer — undo*) substituem as definições criadas por omissão pelo pré-processador. O pré-processador adiciona, ainda, o código necessário a que os blocos *só-uma-vez* sejam executadas apenas uma vez (sob o controlo do componente de reconciliação).

As classes geradas pelo pré-processador são classes Java normalizado e devem ser posteriormente compiladas.

3.4.2 Sumário

O sistema DOORS disponibiliza um suporte de gestão de objectos replicados através de técnicas optimistas. Reconhecendo que não existe uma solução única para satisfazer as necessidades de todas as aplicações, o sistema DOORS inclui um núcleo mínimo que apenas executa as tarefas comuns essenciais para a gestão de dados partilhados. As soluções específicas, necessárias para cada aplicação, são executadas pelos *coobjectos* definidos em cada aplicação.

Neste contexto, a criação de novas aplicações é simplificada pela existência dos seguintes elementos. Primeiro, o *framework* de componentes DOORS permite estruturar uma solução de gestão de dados num conjunto de problemas bem identificados. Segundo, os componentes pré-definidos permitem reutilizar um conjunto de soluções de gestão de dados. Além dos componentes existentes, é possível adicionar, ao sistema, novos componentes que implementem diferentes propriedades. Terceiro, o pré-processador cria automaticamente o código necessário à intercepção das operações (criando os representantes dos

subobjectos e transformando a definição dos outros componentes) e à criação de subobjectos e *coobjectos* (criando as fábricas necessárias).

Capítulo 4

Descrição das funcionalidades principais do sistema DOORS

No capítulo anterior apresentou-se o sistema DOORS, detalhando o *framework* de componente definido. Neste capítulo detalham-se algumas das funcionalidade principais do sistema DOORS, incluindo uma discussão sobre reconciliação no âmbito do sistema DOORS, o modelo de replicação parcial definido, o mecanismo de invocação cega e o modelo de integração de sessões síncronas no âmbito de uma actividade tipicamente assíncrona.

4.1 Reconciliação

O mecanismo de reconciliação é responsável por garantir que a evolução das várias réplicas respeita as propriedades desejadas. O componente de reconciliação/controlo de concorrência desempenha um papel fundamental neste processo, porque decide a ordem de execução das operações, a qual determina, em grande parte, as propriedades da evolução do *coobjecto*.

No entanto, deve realçar-se que a ordem necessária para alcançar um dado conjunto de propriedades depende das operações definidas no *coobjecto*. Adicionalmente, algumas estratégias de reconciliação são implementadas por outros componentes do *coobjecto* e não estão directamente relacionadas com a ordem de execução das operações. Por exemplo, o subobjecto com múltiplas versões descrito na secção 3.3.9 funciona correctamente desde que a ordem de execução das operações respeite a ordem causal.

Algumas das propriedades possíveis na evolução de um *coobjecto* são as seguintes:

Equivalência final A equivalência final consiste na igualdade[1] do estado das várias réplicas de um *co-*

[1] A igualdade do estado de duas réplicas de um *coobjecto* é definida no contexto do tipo de dados que representam, i.e., duas réplicas são idênticas se representam exactamente o mesmo valor do tipo de dados, não sendo necessária a igualdade da representação interna. Por exemplo, um conjunto representado através dos elementos contidos num vector é igual a outro se o

objecto após a execução do mesmo conjunto de operações. Embora seja uma propriedade desejada na maioria das situações, pode não ser necessária em alguns *coobjectos*.

Réplica principal No conjunto de réplicas de um *coobjecto* pode existir uma réplica principal (ou grupo de réplicas) que mantém o seu estado oficial do *coobjecto*. Em geral, a réplica principal (ou grupo de réplicas) pode determinar imediatamente o resultado da execução de uma operação.

Evolução optimista/pessimista A evolução de um *coobjecto* é *optimista* quando todas as operações conhecidas numa réplica são executadas imediatamente. Neste caso, o estado de uma réplica deve ser considerado provisório até se determinar que as operações foram executadas de acordo com a ordem desejada. Quando isto não acontece, é necessário reparar a execução optimista, por exemplo, desfazendo o efeito das operações executadas fora de ordem e reexecutando-as na ordem correcta. A evolução de um *coobjecto* é *pessimista* quando as operações são executadas apenas após se determinar a ordem de execução correcta. Neste caso, o estado de uma réplica pode não reflectir todas as operações conhecidas nessa réplica.

De seguida apresentam-se as várias estratégias de execução de operações usadas no sistema DOORS. Antes, discute-se o modo como as operações podem ser ordenadas usando a informação disponível nas operações.

4.1.1 Informação de ordenação das operações

Para decidir a ordem das operações, o componente de reconciliação tem ao seu dispor a seguinte informação associada a cada sequência de operações.

Primeiro, o par (srv_{view}, n_seq), em que srv_{view} é o identificador do servidor[2] em que a operação é recebida de um cliente e *n_seq* é o valor do relógio lógico [94] do *coobjecto* nesse servidor — uma nova sequência de operações tem um (número de sequência) *n_seq* superior a todos os *n_seq* atribuídos anteriormente nesse servidor, mas não necessariamente imediatamente superior. Este par é atribuído a cada sequência de operações quando ela é recebida directamente do cliente num servidor. Este par permite identificar univocamente as operações e definir uma ordem total entre as operações ($(s_1, n_1) \prec (s_2, n_2)$ sse $n_1 < n_2 \bigvee (n_1 = n_2 \bigwedge s_1 < s_2)$).

vector contiver os mesmos elementos, independentemente da sua ordem no vector.

[2]Cada servidor tem um identificador único em cada vista, srv_{view}. Após um período inicial de instabilidade aquando da entrada do servidor no grupo de replicadores de um volume, em que o identificador do servidor pode mudar entre duas vistas instaladas consecutivamente numa réplica (sendo actualizados os identificadores usados na identificação das operações), o identificador do servidor permanece imutável até este deixar de replicar o volume. Todo o processo de gestão de filiação é descrito na secção 6.2.1.

Segundo, um vector versão [119], v, adicionado no cliente a cada sequência de operações e que contém um sumário das operações já executadas na cópia privada do *coobjecto* manipulada pela aplicação. Este vector permite determinar a dependência causal entre as operações (a sequência de operações op_1 com vector versão v_1 depende causalmente da operação op_2 com identificador (s,n) sse $v_1[s] \geq n$).

Finalmente, o componente de reconciliação pode utilizar, em cada réplica, a seguinte informação mantida no componente de atributos do *coobjecto*: o sumário (vector versão) das operações conhecidas localmente; o sumário (vector versão) das operações executadas localmente; e o sumário (vector versão ou matriz) das operações que se sabem ser conhecidas nas outras réplicas. Estes vectores são mantidos nos componentes de atributos dos *coobjectos* e actualizados aquando da execução local de operações e durante as sessões de sincronização.

A informação descrita anteriormente é mantida em todos os *coobjectos*. No entanto, é possível manter informação adicional para ultrapassar algumas limitações colocadas pela informação anterior, como se descreve de seguida.

Uma primeira limitação consiste no facto de apenas serem registadas dependências causais com operações já submetidas para um servidor (i.e., operações a que já foi atribuído um par (srv_{view}, n_seq)). Assim, não se registam as dependências entre as sucessivas modificações a um mesmo *coobjecto* num cliente desconectado. Para minorar este problema, o cliente propaga as operações para o mesmo servidor por ordem de execução. Assim, se se assumir que existe uma dependência causal entre uma operação identificada num servidor e todas as operações identificadas anteriormente nesse servidor, as dependências causais existentes são respeitadas.

A seguinte solução alternativa pode ser usada. Cada sequência de operações é identificada através de um identificador único. A cada sequência de operações é adicionado o conjunto (ou um resumo) dos identificadores das operações executadas anteriormente que ainda não foram submetidas para um servidor[3]. No caso esperado, o espaço usado por esta informação em cada sequência de operações é reduzido porque o número de elementos do conjunto é pequeno e é possível resumir esse conjunto eficientemente (por exemplo, um identificador (id_clt, n_seq), com *id_clt* um identificador único do cliente e *n_seq* um número de sequência para as operações executadas nesse *coobjecto* no cliente, permite resumir as operações de cada cliente usando o número de sequência inicial e final). No servidor, o componente de reconciliação deve guardar os identificadores das operações executadas para garantir a satisfação das dependências causais. Esta aproximação tem o problema da dificuldade de determinar o momento em que os identificadores podem ser descartados. Assumindo que o número de clientes em que são executadas modificações é limitado, uma aproximação aceitável consiste em manter um

[3]Uma aproximação semelhante é proposta por Fekete et al. [50], na qual cada operação inclui o conjunto das operações que a devem preceder na ordem final de execução.

resumo das operações executadas nos clientes por um período de tempo suficientemente grande. Outras aproximações possíveis foram propostas na literatura [5].

Uma segunda limitação da informação utilizada consiste no facto de os clientes apenas poderem propagar cada sequência de operações para um só servidor. Esta limitação resulta do identificador de uma sequência de operações ser atribuído quando as operações são recebidas no servidor (se uma operação fosse propagada para dois servidores seria tomada como duas operações diferentes). Uma solução para este problema consiste na atribuição de um identificador único a cada sequência de operações (como proposto anteriormente). O componente de reconciliação usa o identificador único para determinar se a operação já foi executada (uma aproximação semelhante é usada no sistema *lazy replication* [93]).

Uma terceira limitação consiste na impossibilidade de executar uma operação com identificador (s,n) sem executar todas as operações com identificador $(s,i) : i < n$. Esta limitação é consequência do sumário das operações executadas ser um vector-versão. Como consequência, a execução de algumas operações numa réplica pode ser atrasada. Esta limitação pode ser ultrapassada, por exemplo, definindo o sumário das operações executadas como a combinação do vector-versão com o conjunto dos identificadores das operações executadas não reflectidas no vector-versão. No entanto, como se antevê que, devido ao modelo de funcionamento do sistema (propagação epidémica entre servidores e os clientes tendem a contactar sempre o mesmo servidor), as situações em que se verifica algum atraso na execução de uma operação são diminutas, pensa-se que a complexidade adicional não compensa.

4.1.2 Sem ordem

A estratégia mais simples de execução das operações consiste em não impor nenhuma restrição à execução das operações, as quais podem ser executadas por ordens diferentes nas várias réplicas.

Para tal, o componente de reconciliação *sem ordem* executa todas as operações imediatamente após a sua recepção (de um cliente ou de outro servidor). Os blocos *só-uma-vez* apenas são executados no servidor em que a operação é recebida do cliente. Como uma operação recebida de um cliente é executada imediatamente, garante-se que todos os blocos *só-uma-vez* são executados.

Neste caso, cada réplica reflecte sempre todas as operações conhecidas. O resultado final de uma operação propagada para um servidor é obtido imediatamente. No entanto, como as operações são executadas por ordem diferente nas várias réplicas, de um modo geral, não é possível garantir que o resultado (efeito) da execução de uma operação é idêntico em todas as réplicas. Assim, caso se pretenda obter a equivalência final das réplicas, este componente apenas pode ser usados quando é possível executar as operações por ordem diferente (por exemplo, quando todas as operações definidas são comutativas entre si).

4.1.3 Ordem causal

Uma operação é executada por ordem causal quando apenas é executada após terem sido executadas todas as operações conhecidas no momento da sua submissão no cliente. A ordem causal não garante que as operações sejam executadas pela mesma ordem em todas as réplicas. Assim, não é possível, no caso geral, garantir que os *coobjectos* convergem para o mesmo estado.

O componente de reconciliação *ordem causal pessimista* executa imediatamente todas as operações conhecidas com as dependências causais satisfeitas (e que podem ser executadas face às limitações apresentadas anteriormente). Assim, cada réplica pode não reflectir todas as operações conhecidas. Para conhecer imediatamente o resultado de uma operação é necessário propagar a operação para um servidor que conheça todas as operações reflectidas na cópia do cliente. O servidor a partir do qual a cópia foi obtida satisfaz esta propriedade.

Os blocos *só-uma-vez* são executados no servidor que recebeu a operação do cliente. No entanto, como as operações podem não ser executadas imediatamente quando são recebidas, é necessário garantir que quando um servidor é removido do grupo de replicadores do volume, os blocos *só-uma-vez* de todas as operações recebidas nesse servidor são executadas.

No sistema DOORS, um servidor abandona voluntariamente o grupo de replicadores de um volume através de um protocolo que estabelece com apenas outro replicador do volume, que assume o papel de patrocinador. Durante este protocolo, apresentado na secção 6.2.1 e detalhado nos apêndices A.1.4 e A.1.8, o servidor a remover delega na réplica do servidor patrocinador a responsabilidade de executar os blocos *só-uma-vez* não executados localmente. Antes de ser removido, o servidor a remover executa ainda os blocos *só-uma-vez* relativos às operações que o patrocinador já tenha executado.

Quando se processa a remoção forçada de um servidor que falhou definitivamente, é impossível determinar as operações que esse servidor já executou. Assim, não se delega em nenhum servidor a execução dos blocos *só-uma-vez* das operações recebidas no servidor avariado. É por esta razão que a semântica exacta de execução destes blocos é *no máximo uma vez*.

4.1.4 Ordem total

As operações são executadas por ordem total quando, em todas as réplicas, são executadas pela mesma ordem. Neste caso, se as operações forem deterministas, garante-se que as várias réplicas do *coobjecto* convergem para o mesmo estado. De seguida, apresentam-se as várias estratégias utilizadas para garantir a execução das operações por ordem total.

4.1.4.1 Sequenciador

No conjunto de réplicas de um *coobjecto* designa-se uma, o sequenciador, que fica responsável por seriar as operações. Para tal, por cada nova operação recebida, o sequenciador submete uma operação especial (que será propagada para todas as réplicas) que apenas contém o identificador da nova operação recebida[4]. A execução desta operação especial corresponde à execução da operação cujo identificador ela contém. A operação especial é identificada como originária no servidor 0, i.e., tem identificador $(0, n_seq)$.

Em todas as réplicas, o componente de reconciliação pessimista, limita-se a executar as operações com identificadores $(0, n_seq)$ por ordem crescente de número de sequência.

O componente de reconciliação optimista usando a técnica *desfazer-refazer* executa três passos sempre que uma nova operação é recebida: desfaz as operações executadas de forma optimista; processa as operações com identificadores $(0, n_seq)$; executa de forma optimista todas as outras operações.

Em ambos os casos, é possível transferir a função de sequenciador entre duas réplicas. Para tal, o sequenciador submete uma operação especial que, ao ser executada, transfere a função de sequenciador para outro servidor.

Quando o servidor que contém o sequenciador decide abandonar o grupo de replicadores de um volume, o sequenciador transfere essa responsabilidade para o patrocinador[5]. Quando o servidor que contém o sequenciador é removido de forma forçada, a réplica do servidor que actua como patrocinador no protocolo de remoção forçada assume o papel de sequenciador.

Os blocos *só-uma-vez* são executados no sequenciador. Como após seriar uma operação, o sequenciador a executa imediatamente garante-se que os blocos *só-uma-vez* são executados uma e uma só vez.

Nesta aproximação, o sequenciador funciona como réplica principal que mantém o estado oficial do *coobjecto*. Adicionalmente, o resultado de uma operação propagada para o sequenciador é conhecido imediatamente.

A execução da ordenação total usando um sequenciador apresenta algumas limitações. Primeiro, por cada operação executada num cliente, o sequenciador executa uma operação adicional que define a ordem de execução da primeira. Segundo, o sequenciador funciona como ponto central de falha — quando o sequenciador falha não é possível ordenar novas operações.

---

[4] Actualmente, este componente limita-se a seriar as operações pela ordem pela qual elas são recebidas no servidor. No entanto, é possível adoptar estratégias mais complexas para optimizar o número de operações que podem ser executadas com sucesso, como proposto no sistema IceCube [134] e SqlIceCube descrito no capítulo 10

[5] Esta transferência é executada submetendo a operação de transferência de sequenciador quando é informado da remoção do servidor — passo 1 do protocolo de remoção.

4.1.4.2 Verificação de estabilidade

A ordenação total por verificação de estabilidade consiste na execução das operações assim que é possível determinar que não existe nenhuma operação anterior na ordem total definida.

O componente de reconciliação por verificação de estabilidade define a ordem total das operações usando o identificador atribuído no servidor: $(s_1,n_1) \prec (s_2,n_2)$ sse $n_1 < n_2 \bigvee (n_1 = n_2 \bigwedge s_1 < s_2)$. Para verificar a estabilidade das operações, cada réplica usa o sumário das operações conhecidas. Este sumário é actualizado durante as sessões de sincronização epidémicas e reflecte não só as operações recebidas, mas também, os números de sequência que se sabe não terem sido usados em cada uma das réplicas.

Em cada servidor, o relógio lógico usado para atribuir o número de sequência a cada operação é actualizado após cada sessão de sincronização de forma a que seja superior ao maior número de sequência de uma operação recebida. Em consequência, o valor relativo ao próprio servidor no sumário das operações recebidas é actualizado (este assunto é detalhado na secção 6.2.2). Este facto garante que, se todos as réplicas participarem em sessões de sincronização (que incluam transitivamente todos os servidores), todas as operações submetidas serão dadas como estáveis em todas as réplicas independentemente do número de operações recebidas em cada servidor.

Para garantir que todas as operações são executadas quando um servidor é removido, a réplica do *coobjecto* nesse servidor submete uma operação a indicar qual o último número de sequência utilizado. No caso da remoção forçada de um servidor, a réplica do patrocinador executa essa operação quando é informada da remoção do servidor (a sincronização do estado entre todas as réplicas garante que o patrocinador conhece todas as operações do servidor removido conhecidas no sistema). Em todas as réplicas, na ordenação das operações, consideram-se não só as réplicas activas, mas também todas as réplicas removidas até serem ordenadas todas as operações com número de sequência menor ou igual ao último número de sequência usado nessa réplica.

Para garantir que as operações são ordenadas correctamente quando um novo servidor se junta ao grupo de replicadores de um volume, é apenas necessário ter especial cuidado com o número de sequência que se atribui à primeira operação submetida no novo servidor (deve ser maior ou igual a $max + 2$, com max o valor do maior número de sequência que o novo servidor sabe ter sido usado em qualquer réplica) e à utilização da informação obtida nesse servidor (por o identificador de um servidor ser inicialmente provisório). A necessidade destas restrições, explicada no apêndice A.1.3.1, apenas ficará clara após a apresentação do mecanismo que controla o conjunto de replicadores de um volume, descrito na secção 6.2.1.

No protótipo do sistema DOORS, implementou-se um componente de reconciliação que usa técnicas de verificação de estabilidade com uma aproximação pessimista. Foi igualmente implementado um componente que usa uma aproximação optimista usando a técnica *desfazer-refazer* (de forma semelhante

à da ordenação total usando um sequenciador).

Para garantir a execução correcta dos blocos *só-uma-vez* usa-se a mesma aproximação utilizada no componente de reconciliação *ordem causal pessimista*.

Quando se usam técnicas de estabilidade, a falha definitiva de um qualquer servidor impede a ordenação de novas operações. No caso de todas as falhas nos servidores serem temporárias, garante-se que todas as operações são finalmente ordenadas desde que todos os servidores continuem a participar em sessões de sincronização, mesmo que não exista nenhum "momento" em que todos os servidores estejam activos.

As múltiplas réplicas de um *coobjecto* convergem para um estado comum passando pelos mesmos estados intermédios. Em cada momento existe uma réplica que executou um maior número de operações (e cujo estado se pode definir como estado oficial actual do *coobjecto*). No entanto, apenas é possível determinar a sua identidade comparando o estado das várias réplicas.

4.1.4.3 Consenso/eleição/quorum

As técnicas apresentadas anteriormente para ordenar totalmente as operações executadas num *coobjecto* apresentam algumas limitações na tolerância a falhas. Assim, no caso da utilização de uma réplica sequenciadora, a falha do sequenciador impede o progresso do sistema, i.e., a ordenação de novas mensagens. No caso da utilização de técnicas de verificação de estabilidade, a falha definitiva de um servidor impede o progresso do sistema (até o servidor ser removido do grupo de replicadores do volume).

Para ultrapassar as limitações anteriores é possível implementar técnicas de ordenação por consenso. Neste caso, o sistema pode progredir desde que exista um qualquer quorum [56, 121] de réplicas activas.

Usando técnicas de consenso, a ordenação das operações faz-se em rondas sucessivas. Em cada ronda, as réplicas de um *coobjecto* executam um algoritmo de consenso [102] no qual decidem qual a operação (ou conjunto de operações) a ordenar.

Os algoritmos de consenso propostos em [70] e [82], baseados na utilização de comunicação epidémica, podem ser implementados de forma imediata no sistema DOORS. Nestes algoritmos, em cada ronda, cada servidor deve votar em uma e uma só operação. A operação a ser executada é aquela que obtiver um quorum de votos em [70] ou a maioria dos votos em [82]. Enquanto no algoritmo [70] é possível a existência de rondas em que nenhuma operação é ordenada, em [82] selecciona-se sempre uma operação. Em ambos os algoritmos existem situações em que a decisão apenas pode ser tomada após conhecer o voto de todos os servidores, o que é consistente com a conhecida impossibilidade de definir um algoritmo de consenso que termine em todas as situações num sistema distribuído assíncrono [52]. Para contornar esta impossibilidade foram propostos vários algoritmos usando diferentes aproximações, entre as quais a utilização de detectores de falhas [3, 72, 110] e aleatoriedade [15].

No decurso do trabalho que conduziu a esta dissertação explorou-se a possibilidade de desenvolver e implementar um componente de reconciliação baseado na utilização de um algoritmo de consenso[6]. Este componente teria como objectivo ultrapassar as limitações na tolerância a falhas expostas anteriormente.

No entanto, um estudo desenvolvido num ambiente de larga escala (a Internet) [9] mostra que as falhas de comunicação limitam a disponibilidade dos sistemas de quorum. Nos ambientes de larga escala, o melhor sistema de quorum parece ser obtido limitando os servidores utilizados a uma única rede local. No âmbito de uma rede local, a forte correlação nas falhas dos servidores (as quais são explicadas pela dependência de um serviço de DNS comum, de uma fonte de energia comum, de um sistema distribuído de ficheiros comum, etc.) limita a melhoria de disponibilidade obtida pela utilização de um sistema de quorum face ao recurso a um servidor único. Assim, como os algoritmos de consenso requerem que exista um quorum de servidores activos, parece não ser clara a vantagem na utilização de um componente de reconciliação usando técnicas de consenso para ultrapassar as limitações na tolerância a falhas, face à utilização do componente de reconciliação que usa um sequenciador. Este facto, conjugado com o elevado número de mensagens envolvidas num algoritmo de consenso (i.e., mensagens necessárias para ordenar cada operação ou conjunto de operações), levou a que não se tivesse implementado nenhum componente de reconciliação utilizando técnicas de consenso.

4.1.5 Transformação de operações

A execução pela mesma ordem em todas as réplicas de um conjunto de operações executadas concorrentemente garante a equivalência final das várias réplicas. No entanto, como cada operação vai ser executada num estado do *coobjecto* diferente do observado aquando da execução (submissão) original da operação, os seus efeitos podem ser diferentes dos esperados pelo utilizador.

Para solucionar este problema foi proposto que, antes de executar uma operação numa réplica, a operação fosse transformada para incluir os efeitos das operações não conhecidas aquando da execução inicial da operação. Desta forma, as intenções do utilizador são respeitadas. Vários algoritmos foram propostos utilizando esta ideia [46, 159, 118, 158].

Esta aproximação tem o inconveniente de requerer a definição de uma ou várias funções capazes de transformar as operações. No caso geral, definir uma função de transformação de operações pode ser bastante complexo, pelo que esta técnica tem sido utilizada quase exclusivamente no âmbito dos editores cooperativos síncronos. No entanto, os exemplos apresentados na literatura e a experiência obtida com o sistema DOORS, parecem sugerir que esta técnica pode ser usada facilmente nas seguintes situações: anular a execução de operações idênticas; e ajustar parâmetros que contêm posições absolutas.

No protótipo do sistema DOORS estendeu-se o componente de reconciliação ordem total optimista

[6]No âmbito deste trabalho, chegaram a ser produzidos alguns resultado relativos à teoria de sistemas de quorum [127].

usando sequenciador para efectuar a transformação de operações de acordo com o algoritmo apresentado em [159]. Para utilizarem este componente, os utilizadores devem definir as funções de transformação de operações necessárias.

4.1.6 Operações do sistema

Como se referiu anteriormente, as operações definidas num *coobjecto* podem ser classificadas como operações do sistema. Estas operações são tratadas de forma especial pelos componentes de reconciliação: uma operação de sistema é executada imediatamente após ser recebida no servidor e não é reflectida no sumário das operações executadas. Assim, as operações de sistema podem ser executadas por diferentes ordens nas várias réplicas. O sumário das operações de sistema executadas é mantido de forma independente pelo componente de reconciliação de forma a garantir que cada operação de sistema apenas é executada uma vez.

4.2 Replicação secundária parcial

O mecanismo de replicação secundária parcial (ou *caching* parcial) permite aos clientes obter cópias parciais dos *coobjectos*. Uma cópia parcial de um *coobjecto* consiste na parte comum do *coobjecto* e num subconjunto dos subobjectos contidos no *coobjecto*. A parte comum do *coobjecto* consiste nos componentes necessários (cápsula, atributos do sistema, etc.) à criação de uma cópia do *coobjecto*.

O bom funcionamento deste mecanismo baseia-se no pressuposto que um subobjecto representa uma unidade de manipulação dos dados, sendo possível executar uma operação num subobjecto sem que ocorra nenhum falha na replicação. Desta forma, a divisão de um *coobjecto* em subobjectos simplifica o processo de replicação antecipada (*pre-fetching*).

A razão lógica desta aproximação consiste na seguinte observação: o programador que desenha o *coobjecto* é a pessoa que melhor conhece o modo como os vários objectos interagem entre si, e quais devem ser replicados conjuntamente. Assim, o programador pode tornar esta informação visível ao sistema, agrupando os objectos fortemente ligados em unidades de maior dimensão: os subobjectos. Adicionalmente, um subobjecto pode especificar um conjunto de outros subobjectos que devem ser replicados conjuntamente com esse subobjecto (através da lista de subobjectos relacionados mantida no componente de atributos do sistema associado ao subobjecto).

Um cliente obtém uma cópia parcial de um *coobjecto* a partir de um servidor. Posteriormente, pode actualizar a cópia parcial ou "aumentá-la" através da replicação de novos subobjectos. O funcionamento do sistema de replicação secundária parcial é detalhado na secção 6.3.1.

4.3 Invocação cega

O mecanismo de invocação cega tem como objectivo permitir que os utilizadores produzam contribuições úteis que afectem dados não replicados localmente durante os períodos de desconexão (como se discutiu na secção 2.3.6).

Para tal, os utilizadores podem submeter operações sobre subobjectos que não estão presentes localmente, desde que possuam uma referência (representante) para esse subobjecto. O processamento destas invocações é efectuado de forma semelhante ao processamento normal até ao momento em que as operações devem ser executadas localmente sobre a cópia privada do subobjecto. Neste momento a execução não é efectuada e como resultado da invocação é lançada uma excepção que explica a situação. No entanto, como a informação sobre a invocação é guardada no componente de registo, ela será transmitida para o servidor onde será executada de forma semelhante às outras invocações. Desta forma simples, os utilizadores podem submeter modificações sobre subobjectos dos quais não possuem uma cópia local, como pretendido.

O modo de processamento descrito é o modo normal utilizado para processar as invocações sobre subobjectos não replicados localmente. No entanto, é possível aos programadores especificar, para cada método, os seguintes comportamentos alternativos:

silencioso Neste modo, não são lançadas excepções como resultado de uma invocação sobre um subobjecto não presente localmente — para métodos que devolvam um resultado, o programador pode especificar um valor a devolver nessa situação.

local Neste modo, as invocações sobre subobjectos não presentes localmente falham, sem serem armazenadas no componente de registo.

A aplicação, ao obter a cópia privada do *coobjecto*, pode igualmente especificar o comportamento a utilizar. As opções especificadas sobrepõem-se às opções definidas por omissão nos subobjectos. O código para executar estes comportamentos alternativos é criado pelo pré-processador nos representantes dos subobjectos[7].

4.3.1 Cópias de substituição

Para permitir a uma aplicação observar o resultado de uma invocação cega que modifique o estado de um subobjecto, permite-se a criação de uma *cópia de substituição* desse subobjecto.

[7]No caso de a aplicação especificar o modo de operação "silencioso" e o resultado das operações não estiver especificado, as operações devolvem valores pré-definidos — no protótipo do sistema DOORS, implementado em Java, são devolvidos os valores de iniciação para o tipo considerado.

O processamento de uma invocação é tratado como normalmente até ao momento em que é necessário executar a operação na cópia privada do subobjecto. Neste momento, é criada uma cópia de substituição do subobjecto e a operação é executada sobre essa cópia, como seria executada normalmente sobe a cópia real do subobjecto. A partir da criação desta cópia, todas as invocações seguintes são executadas nessa cópia.

Os programadores, ao definirem os subobjectos, devem especificar se é possível criar cópias de substituição e o modo como estas cópias são criadas — em geral, a criação de uma cópia de substituição consiste na criação de um subobjecto em que o estado inicial é definido a partir dos parâmetros do construtor. O representante (*proxy*) do subobjecto contém o código necessário para criar a cópia de substituição do subobjecto[8]. Os parâmetros que definem o estado inicial da cópia de substituição podem ser definidos estaticamente, aquando da definição do código do subobjecto, ou dinamicamente, aquando da criação do subobjecto — neste caso, cada representante de um subobjecto obtém uma cópia destes parâmetros quando é criado.

A aplicação, ao obter a cópia privada do *coobjecto*, pode especificar que não pretende criar cópias de substituição — esta opção sobrepõe-se à opção especificada na definição dos subobjectos.

4.4 Integração de sessões síncronas

O sistema DOORS foi desenhado para suportar a partilha de dados em ambientes de trabalho cooperativo tipicamente assíncronos. Nestes ambientes, os utilizadores executam as suas contribuições concorrentemente sem observarem imediatamente as modificações que os outros utilizadores estão a efectuar. Estas modificações são posteriormente unificadas através de um mecanismo de reconciliação adaptado às características dos dados manipulados.

Assim, é possível reconhecer duas características fundamentais: o desconhecimento das modificações produzidas pelos outros utilizadores e a (significativa) dimensão das contribuições de cada utilizador. Considere-se o exemplo da edição cooperativa de um documento. Cada utilizador desconhece as modificações que o outro está a executar, embora, em algumas situações, os utilizadores possam ter conhecimento dos elementos do documento estruturado que estão a ser modificadas pelos outros utilizadores — este conhecimento pode ser obtido através da informação de *awareness* ou coordenação formal ou informal associada à tarefa cooperativa. Adicionalmente, quando um utilizador modifica um elemento (por exemplo, uma secção), as modificações tendem a ser significativas.

Num ambiente de trabalho cooperativo tipicamente assíncrono podem existir momentos pontuais em

[8]No cliente, a informação sobre uma invocação contém o representante no qual a invocação foi executada. Este facto permite criar a cópia de substituição usando o código definido no representante do subobjecto, apesar de ser o componente de reconciliação que executa a operação e o gestor do subobjecto o responsável por manter as cópias dos subobjectos.

que os utilizadores pretendem realizar sessões síncronas durante as quais manipulam os dados de forma síncrona. Por exemplo, durante a edição cooperativa, estas sessões síncronas podem ser usadas para coordenar o trabalho e unificar múltiplas versões dos dados.

Para responder a esta necessidade, o sistema DOORS permite que os *coobjectos* sejam manipulados durante sessões síncronas. Para tal, é necessário manter várias cópias privadas de um *coobjecto* sincronamente sincronizadas. A aproximação utilizada consiste em difundir as invocações executadas numa sessão síncrona para todas as cópias síncronas do *coobjecto* — um componente de adaptação especial implementa esta funcionalidade recorrendo a um mecanismo de comunicação em grupo[9] [18].

Um utilizador pode iniciar uma sessão síncrona a partir da sua cópia privada do *coobjecto*. Para tal, ele deve instalar o componente de adaptação síncrona no seu *coobjecto* (assim como um componente de reconciliação adaptado às características de uma sessão síncrona)[10]. O componente de adaptação síncrona cria um grupo para a sessão síncrona no sistema de comunicação em grupo utilizado. A partir deste momento é possível a outros utilizadores entrarem na sessão síncrona para manipular o *coobjecto*.

Quando um utilizador pretende entrar numa sessão síncrona, a aplicação deve-se juntar ao grupo associado à sessão síncrona (no protótipo do sistema, é apenas necessário conhecer o nome da sessão e o nome de um computador que participe na sessão). A cópia privada do *coobjecto* é criada a partir do estado actual do *coobjecto* na sessão síncrona — o estado actual inclui todos os subobjectos instanciados em memória e uma referência (*handle*) que permite criar os subobjectos de forma coerente em todos as réplicas síncronas (no protótipo actual, o estado inicial é obtido a partir de um elemento no grupo eleito como primário). Esta cópia privada é actualizada através da execução de todas as operações difundidas na sessão síncrona e não reflectidas no estado inicial.

Os elementos de uma sessão síncrona podem abandoná-la em qualquer momento que o desejem. Adicionalmente, o mecanismo de filiação do subsistema de comunicação em grupo pode forçar a remoção de elementos com os quais seja impossível comunicar (em situações de partição do grupo, o subsistema de comunicação em grupo apenas deve permitir que o grupo continue activo numa das partições). Em cada momento existe sempre um elemento do grupo que é designado de primário.

As aplicações manipulam os *coobjectos* da forma habitual, i.e., através da execução de operações nos representantes dos subobjectos. As operações de leitura são executadas localmente — o componente de adaptação propaga a invocação para execução local. As operações de modificação são difundidas para

[9]No protótipo do sistema DOORS foi utilizado um sistema de comunicação em grupo muito simples implementado a partir do sistema de disseminação de eventos Deeds [45], embora fosse possível ter optado por outros sistemas de comunicação em grupo — as únicas funcionalidades básicas utilizadas são a difusão de mensagens e o controlo da filiação.

[10]Esta substituição de componentes representa um cenário limitado de reconfiguração dinâmica, porque as possibilidades de reconfiguração estão pré-determinadas e a arquitectura está desenhada para que esta substituição possa ser efectuada sem problemas.

todos os elementos da sessão síncrona — o componente de adaptação executa esta difusão através do mecanismo de comunicação em grupo associado (a cada invocação é associada informação que permite traçar a dependência entre operações executadas na sessão síncrona).

Além de difundir as operações para todos os elementos do grupo, o componente de adaptação síncrona condiciona o processamento local das operações. Duas alternativas podem ser adoptadas.

Primeiro, o componente de adaptação pode propagar as operações para serem processadas localmente apenas após terem sido recebidas e ordenadas pelo subsistema de comunicação em grupo. Neste caso, o componente de adaptação induz uma estratégia pessimista para a execução das operações. O componente de reconciliação usa a ordem estabelecida para executar as operações. O componente de reconciliação poderia executar adicionalmente um mecanismo de transformação das operações [46] para garantir a preservação das intenções dos utilizadores nas operações executadas concorrentemente.

Segundo, o componente de adaptação propaga para execução local as invocações locais e as invocações recebidas pelo subsistema de comunicação em grupo. Neste caso, o componente de reconciliação deve executar a estratégia adequada à manipulação síncrona do respectivo tipo de dados. Esta estratégia pode ser optimista ou pessimista.

As aplicações podem registar funções (*callbacks*) no componente de adaptação para serem notificadas do processamento de uma invocação recebida de outro elemento do grupo — em geral, as aplicações usam este mecanismo para reflectir as modificações processadas na interface gráfica da aplicação.

As modificações executadas durante uma sessão síncrona apenas podem ser gravadas pelo primário do grupo associado à sessão síncrona. Em relação à evolução global dos *coobjectos*, as modificações executadas durante uma sessão síncrona são tratadas de forma semelhante às modificações executadas assincronamente por um único utilizador. Assim, a sequência de operações executada é propagada para os servidores, onde é integrada de acordo com a política de reconciliação usada pelo *coobjecto* nos servidores. Note-se que esta sequência de operações deve representar o modo como as operações foram executadas sequencialmente na cópia privada do *coobjecto* e pode ser resultado do processamento executado pelo componente de reconciliação no cliente. Por exemplo, quando se usam técnicas de transformação de operações [46], a sequência de operações enviada para o servidor inclui as operações após terem sido transformadas (i.e., como foram executadas na cópia privada do *coobjecto*).

4.4.1 Diferentes operações para sessões síncronas e assíncronas

Anteriormente descreveu-se a aproximação básica usada no sistema DOORS para permitir a manipulação de *coobjectos* em sessões síncronas. No entanto, esta aproximação não é suficiente para algumas aplicações em que as operações usadas durante a interacção síncrona e assíncrona devem ser diferentes (as razões subjacentes a esta diferença serão discutidas com maior detalhe na próxima subsecção). Consi-

dere-se, por exemplo, a edição cooperativa de documentos. Numa sessão síncrona são usadas operações com reduzida granularidade – por exemplo, inserir/remover um carácter. Num sessão assíncrona são usadas operações com uma maior granularidade – por exemplo, definir uma nova versão de um elemento (secção) do documento.

Em geral, os *coobjectos*, ao serem desenhados para serem manipulados em interacções assíncronas, definem apenas as operações adequadas a este tipo de interacção. Adicionalmente, as estratégias de gestão de dados usadas nos servidores (em particular, a estratégia de reconciliação) estão especialmente adaptadas a este tipo de operações e podem ser inadequadas para gerir outros tipos de operações. No sistema DOORS, existem duas alternativas para permitir a utilização de operações diferentes nas sessões síncronas e assíncronas.

A primeira alternativa consiste em estender as interfaces dos subobjectos para incluírem as operações usadas durante a interacção síncrona. No entanto, a utilização destas operações com pequena granularidade na evolução global dos *coobjectos* põe problemas em termos da gestão das operações [154] (devido ao espaço necessário para as armazenar e ao elevado número de operações existentes) e da estratégia de reconciliação usada (que deve ter em conta estas novas operações). Assim, propõe-se que, antes de serem propagadas para os servidores, as operações executadas durante uma sessão síncrona sejam convertidas numa pequena sequência de operações "assíncronas" que produzam o mesmo efeito. Por exemplo, uma sequência de inserções/remoções de caracteres é convertida numa operação que define o novo estado de uma secção. O componente de registo utilizado no cliente define um mecanismo de compressão de operações que pode ser utilizado para este efeito (através da definição das funções de compressão adequadas).

A segunda alternativa consiste em manipular as operações de pequena granularidade fora do controlo dos *coobjectos*. As modificações executadas são reflectidas no estado do *coobjecto* executando uma sequência de operações que produza o mesmo efeito (de forma semelhante ao resultado da compressão das operações). Para clarificar esta alternativa, considere-se o exemplo da edição cooperativa. Cada membro da sessão síncrona mantém um editor que manipula uma cópia do *coobjecto* — estas cópias são mantidas sincronizadas como se explicou anteriormente. A edição síncrona do texto de um elemento do documento (por exemplo, uma secção) é executada numa aplicação de edição síncrona a partir do estado actual do elemento no *coobjecto* (e independentemente do *coobjecto*). No fim da edição síncrona do elemento, o novo estado do elemento é reflectido no *coobjecto* executando a operação correspondente.

Embora esta segunda alternativa possa ser vista como uma "não solução", ela apresenta, na prática, algumas características interessantes. Primeiro, torna possível a utilização de aplicações já existentes especialmente desenhadas para a gestão das interacções síncronas. Segundo, permite simplificar o desenho dos *coobjectos*, definindo apenas um tipo de operações. Terceiro, pode possibilitar uma maior

eficiência nas comunicações, restringindo a propagação das operações síncronas de pequena granularidade aos elementos interessados nessas operações (por exemplo, as operações relativas à edição síncrona de um elemento de um documento estruturado apenas são propagadas para os elementos que participam nessa edição específica). No editor de documentos multi-síncrono desenvolvido no protótipo do sistema DOORS foi utilizada esta estratégia (ver secção 5.1).

4.4.2 Discussão

Para algumas aplicações, a cooperação síncrona e assíncrona apresentam diferenças intrínsecas que se reflectem na forma como os dados partilhados são alterados. Estas diferenças levam a que o suporte necessário para os dois tipos de interacção seja igualmente diferente.

Na cooperação síncrona, as contribuições executadas por um utilizador em cada passo são geralmente muito reduzidas (por exemplo, a inserção/remoção de um carácter). Adicionalmente, as contribuições de cada utilizador são propagadas para os outros utilizadores (quase) imediatamente — quando um utilizador executa uma contribuição, ele tem conhecimento de todas (ou quase todas) as modificações executadas pelos outros utilizadores nos dados partilhados. Este facto, permite que as contribuições de um utilizador sejam influenciadas pelas contribuições que os outros utilizadores estão a executar. Deste modo, os utilizadores podem coordenar facilmente as suas modificações.

Estas propriedades permitem a utilização de estratégias de reconciliação agressivas em que a consistência dos dados é o objectivo principal a obter (mesmo em detrimento de preservar todas as contribuições ou garantir a preservação das intenções dos utilizadores). Quando estas técnicas não produzem o efeito desejado pelos utilizadores, é possível aos utilizadores solucionar imediatamente o problema (embora estas situações devam ser evitadas) porque eles observam (quase) de imediato o resultado da unificação das operações concorrentes e porque as suas contribuições são de pequena dimensão.

Na cooperação assíncrona, as contribuições executadas são geralmente de grande dimensão (por exemplo, a modificação de uma secção de um documento). Adicionalmente, os utilizadores não têm informação precisa sobre as contribuições que os outros utilizadores estão a produzir. Desta forma, é impossível aos utilizadores coordenar fortemente as suas contribuições.

Assim, a estratégia de reconciliação deve ser menos agressiva do que na interacção síncrona — a consistência dos dados e a preservação das contribuições dos utilizadores são objectivos a alcançar. A preocupação com a preservação das contribuições é consequência de, em geral, por estas serem de dimensão significativa, não ser aceitável para um utilizador perder as suas contribuições (é por esta razão que vários sistemas que permitem a interacção assíncrona criam múltiplas versões dos dados quando não conseguem unificar as modificações concorrentes — por exemplo, o CVS [24] e o Lotus Notes [101]).

Estas propriedades levam a que, pelo menos para algumas aplicações e ao contrário do que é muitas

vezes sugerido na literatura [154, 42], seja impossível obter os efeitos desejados utilizando as mesmas técnicas de reconciliação para controlar a interacção síncrona e assíncrona. Por exemplo, considera-se que a transformação de operações é a técnica de reconciliação mais adequada ao controlo da edição cooperativa síncrona de um texto [46], definido como uma sequência de caracteres. No entanto, a utilização desta técnica durante a interacção assíncrona não parece capaz de produzir resultados apropriados. Um exemplo simples demonstra as limitações existentes. Suponha-se que dois utilizadores modificam a frase "Tu escreve um livro". O primeiro corrige o erro produzindo o resultado final "Tu escreves um livro". O segundo corrige o erro, mudando o sujeito da frase "Ele escreve um livro". Após a reconciliação das duas modificações o resultado final seria "Ele escreves um livro", ou seja, um resultado que não satisfaz nem o primeiro utilizador nem o segundo. Os problemas tendem a ser mais graves quando as modificações são mais significativas, podendo levar à completa incoerência do texto final. Na edição síncrona, este facto não coloca problemas porque os vários utilizadores podem coordenar fortemente as modificações que produzem.

Assim, embora seja possível traçar um contínuo entre os dois extremos da cooperação síncrona e assíncrona, baseado na rapidez com que as contribuições são propagadas entre os utilizadores, parece existir um ponto a partir do qual é necessário utilizar técnicas de gestão de dados diferentes. As observações anteriores sugerem a necessidade de usar diferentes técnicas de reconciliação/controlo de concorrência e operações com diferentes granularidades. Adicionalmente, a gestão da informação de *awareness* também necessita de ser adaptada. Na interacção assíncrona, é necessário que um utilizador possa ser informado das modificações que os outros utilizadores produziram no passado (e, eventualmente, das previsões de modificações a executar no presente/futuro). Pelo contrário, durante as sessões síncronas, a maior parte da informação de *awareness* resulta da observação imediata das modificações que os outros utilizadores estão a produzir e de outros canais de coordenação activos (por exemplo, ferramentas de troca interactiva de mensagens).

A solução adoptada no sistema DOORS permite que estas diferenças sejam tomadas em consideração na integração de sessões síncronas. Assim, durante a interacção síncrona é possível utilizar as técnicas de gestão de dados adequadas a esse tipo de interacção. As modificações produzidas nas sessões síncronas são integradas na evolução global dos dados através de operações adequadas ao tratamento assíncrono — a conversão entre as operações pode ser efectuada usando uma das duas alternativas apresentadas anteriormente. No âmbito da evolução global dos dados, estas operações são tratadas de forma semelhante às modificações executadas assincronamente por um único utilizador e processadas de acordo com as técnicas de gestão de dados adequadas à interacção assíncrona. Na secção 5.1 apresenta-se um editor cooperativo multi-síncrono que demonstra a utilização da solução do sistema DOORS.

Capítulo 5

Avaliação do modelo do sistema DOORS

O objectivo do sistema DOORS é suportar a criação de aplicações cooperativas tipicamente assíncronas num ambiente de larga-escala. O sistema pode ser dividido em duas partes: o núcleo do sistema, responsável por garantir a disponibilidade dos dados; e os *coobjectos*, responsáveis por implementar as soluções específicas relativas à partilha de dados num ambiente de larga-escala.

Para fornecer uma elevada disponibilidade dos dados, o núcleo do sistema combina a replicação dos *coobjectos* no servidor, a replicação secundária parcial nos clientes e o acesso aos dados usando uma aproximação optimista. Estas técnicas foram usadas anteriormente em diversos sistemas (por exemplo, Bayou [161], Lotus Notes [101] e Coda [87]) para alcançar objectivos semelhantes de forma satisfatória. Assim, os mecanismos do sistema DOORS parecem adequados para alcançar os objectivos propostos, quando combinados com a utilização de boas políticas de replicação prévia (*pre-fectching*), distribuição e sincronização das réplicas. Estes problemas não foram tratados no âmbito desta dissertação, mas soluções semelhantes às propostas em [91, 103, 84] podiam ser utilizadas no sistema DOORS. No próximo capítulo detalham-se os protocolos utilizados pelo núcleo do sistema.

O mecanismo de execução cega e (a implementação da) replicação secundária parcial representam duas diferenças importantes relativamente aos sistemas mencionados anteriormente. A motivação para a introdução destes mecanismos foi apresentada nas secções 2.3.6 e 2.3.5, respectivamente. As aplicações apresentadas nesta secção exemplificam a sua utilização em cenários concretos e a simplicidade da solução implementada no sistema DOORS.

O *framework* de componentes DOORS identifica um conjunto de aspectos relacionados com a partilha de dados em ambientes de larga-escala. Um *coobjecto* pode ser criado compondo uma solução global a partir de soluções particulares para cada um dos problemas identificados. Esta aproximação, não só, simplifica a criação de novos tipos de dados, mas também, permite a criação de uma solução adequada a cada tipo de dados. Desta forma, o *framework* de componentes DOORS é uma peça fundamental do sistema e representa uma diferença marcante relativamente à generalidade de sistemas de gestão de dados.

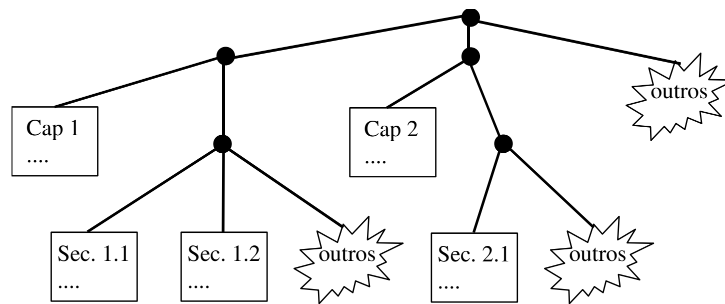

Figura 5.1: Dissertação representada como um documento estruturado (*outros* representa uma subárvore que foi omitida).

As aplicações apresentadas nesta secção exemplificam a utilização do *framework* de componentes.

O protótipo do sistema DOORS utiliza extensivamente os mecanismos de carregamento dinâmico de código e de serialização de objectos da linguagem Java. No entanto, o modelo proposto é independente da linguagem, como o demonstra uma implementação parcial do *framework* de componentes na linguagem Phyton, executada independentemente.

De seguida, descrevem-se várias aplicações que exemplificam a utilização do sistema DOORS como suporte à criação de aplicações cooperativas tipicamente assíncronas.

5.1 Editor multi-síncrono de documentos

O editor multi-síncrono de documentos permite a edição de documentos estruturados. Um documento estruturado é composto por uma árvore de elementos básicos. O conteúdo dos elementos básicos e as possíveis configurações da árvore dependem do tipo de documento definido. Por exemplo, esta dissertação é composta por uma sequência de capítulos; cada capítulo é composto pelo título e algum texto inicial seguido de uma sequência de secções; e assim sucessivamente. Os elementos básicos contêm texto (e figuras/tabelas). Esta estrutura está ilustrada na figura 5.1.

Para modificar um documento estruturado podem definir-se operações que modifiquem a sua estrutura e operações que modifiquem os seus elementos básicos.

No editor criado, os documentos estruturados são representados como *coobjectos*. Para tal, definiu-se o *coobjecto* documento estruturado base, que é utilizado como esqueleto para todos os documentos estruturados manipulados pelo editor. Este esqueleto inclui a definição dos subobjectos que mantêm a estrutura do documento e os seus elementos básicos.

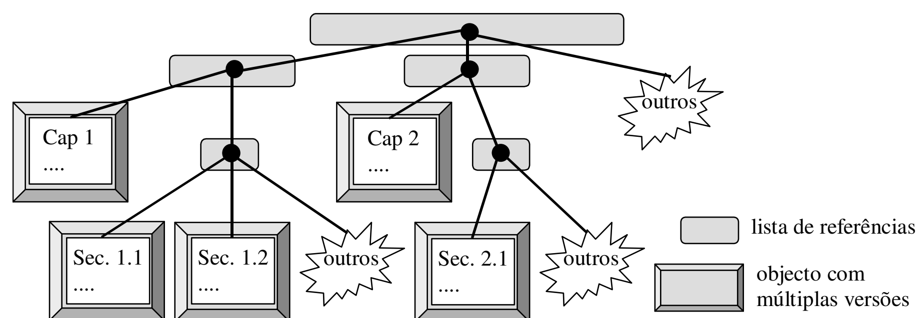

Figura 5.2: Organização de um documento estruturado em subobjectos.

5.1.1 *Coobjecto* documento estruturado

A informação de um documento estruturado é armazenada em dois tipos de elementos, implementados usando dois tipos de subobjectos descritos na secção 3.3.9: a lista de referências para outros subobjectos; e o objecto com múltiplas versões. A estrutura de um documento é construída compondo listas de referências numa organização em árvore. As folhas da árvore armazenam o conteúdo do documento em objectos com múltiplas versões. A figura 5.2 representa esta organização para o documento apresentado anteriormente.

No subobjecto *lista de referências* estão definidas as operações que permitem manipular a estrutura de um documento, adicionando um novo elemento, removendo ou movendo um elemento existente. No subobjecto *objecto com múltiplas versões* estão definidas as operações que permitem modificar o conteúdo de um elemento básico: criar nova versão, remover e modificar uma versão existente. O modo como as operações submetidas concorrentemente são tratadas está igualmente definido nos subobjectos (ver secção 3.3.9).

Relativamente aos elementos básicos, a estratégia utilizada recorre à criação de versões para que não se perca nenhuma modificação executada por um utilizador. Esta aproximação justifica-se porque se considera o conteúdo de um elemento básico uma unidade semântica de dimensões consideráveis. Assim, desconhece-se o modo de unificar modificações concorrentes[1]. Adicionalmente, pela sua dimensão, não é razoável descartar uma modificação produzida por um utilizador. Os utilizadores devem posteriormente criar uma modificação unificadora das várias versões.

Relativamente à estrutura, a estratégia utilizada consiste em unificar todas as modificações produzidas. O fundamento lógico desta estratégia consiste no pressuposto que as modificações concorrentes à estrutura são, em geral, modificações complementares. No caso de um documento estruturado existem duas situações em que este pressuposto pode não ser verdadeiro. Primeiro, quando dois utilizadores movem um mesmo elemento — neste caso, o resultado final reflecte apenas uma das operações, mas

[1]Quando exista um algoritmo que permita unificar duas versões, é possível estender este subobjecto para unificar automaticamente as várias versões.

nenhuma informação é perdida. Segundo, quando um utilizador move um elemento que é removido concorrentemente. Neste caso, a referência é removida[2], pelo que o subobjecto não reflecte o efeito da operação que move a referência. Esta situação é semelhante à remoção da referência de um elemento básico concorrentemente com a sua modificação: a modificação perde-se porque o subobjecto é removido.

Para diminuir a probabilidade destas situações, o editor força a remoção de um elemento em dois passos que devem ser executados explicitamente pelos utilizadores. Num primeiro passo, o utilizador pode pré-remover um elemento — a operação de pré-remoção é submetida. No segundo passo, o utilizador pode remover um subobjecto pré-removido — a operação que remove a referência do elemento é submetida na lista de referências respectiva. Relativamente a subobjectos pré-removidos, o editor apenas permite executar duas operações: remover e restaurar.

O *coobjecto* documento estruturado base é um *coobjecto* comum baseado na cápsula simples (os componentes utilizados estão descritos na secção 3.3).

No servidor, é necessário usar um componente de reconciliação optimista que execute as operações por ordem total (neste caso, utiliza-se o componente baseado na verificação da estabilidade da ordem), de forma a garantir a convergência das várias réplicas e permitir a observação do efeito de todas as operações conhecidas (ainda que de forma provisória). Relativamente aos componentes de adaptação e registo usam-se as implementações mais simples para o servidor.

No cliente, usam-se as implementações mais simples para o componente de reconciliação e de registo. Na execução normal, usa-se o componente de adaptação que executa as operações localmente. Como se detalha mais tarde, durante a execução síncrona usa-se um componente de adaptação especial (descrito na secção 4.4). Para permitir a substituição do componente de adaptação numa instância do *coobjecto*, é necessário indicar a validade da substituição numa operação definida na cápsula do *coobjecto* — para tal, criou-se uma nova cápsula derivada da cápsula simples a usar no *coobjecto* documento estruturado.

Relativamente à informação de *awareness*, usa-se o componente que mantém essa informação como uma lista das mensagens produzidas pela execução das operações. As operações definidas nos subobjectos produzem mensagens simples indicando quais as alterações executadas. O editor permite aos utilizadores consultar a lista de modificações de forma a tomarem conhecimento da evolução do documento[3].

Para criar um tipo de documento específico é necessário definir quais os tipos de elementos básicos permitidos e a sua possível configuração.

[2]Existe uma excepção no caso de se mover a referência para outro subobjecto e essa operação ser executada primeiro.

[3]Esta informação pode ser tratada pelo editor de forma diferente — por exemplo, pode usar cores diferentes para representar os elementos que foram modificados recentemente (a esta forma de apresentar a informação de *awareness* é usual chamar-se *consciência partilhada* (*shared feedback*) [43]).

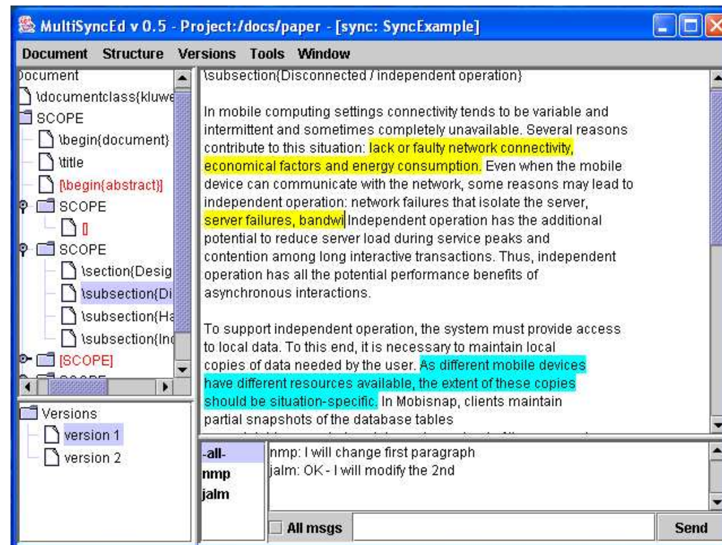

Figura 5.3: Edição de um documento LaTeX. As janelas da esquerda apresentam a estrutura do documento (em cima) e as versões do elemento seleccionado (em baixo). As janelas da direita apresentam o conteúdo da versão seleccionada (em cima) e a janela relativa à edição multi-síncrona.

Um tipo de elemento básico é criado derivando o subobjecto objecto com múltiplas versões de forma a incluir na interface o tipo de objectos armazenado. As novas operações definidas limitam-se a invocar as operações genéricas definidas no subobjecto base.

Na representação do documento estruturado como uma árvore de elementos, os nós da árvore definem a sua estrutura. Um nó da árvore é criado derivando o subobjecto lista de referências de forma a incluir na interface o tipo de elementos que podem ser adicionados à lista. As operações definidas, após verificarem a validade da operação no elemento da estrutura definido, invocam as operações genéricas definidas no subobjecto base.

Por exemplo, um documento de texto genérico pode ser definido de forma muito simples recorrendo a dois tipos de elementos. O único tipo de elemento básico armazena objectos que contêm texto. Os nós da árvore são listas de referências de outros nós e/ou de instâncias do único elemento básico definido. Neste caso, não se impõe nenhuma restrição à estrutura definida pelos utilizadores.

Na figura 5.3 pode observar-se a utilização deste documento de texto genérico para armazenar um documento LaTeX. Neste caso, por exemplo, cada secção é armazenada como uma lista de referências para elementos básicos que contêm o texto da secção e os textos de cada uma das subsecções (esta organização podia ser refinada definindo, por exemplo, uma nova lista para cada subsecção ou dividindo cada secção/subsecção numa sequência de parágrafos).

Após criar os subobjectos usados no documento que se pretende criar, o novo *coobjecto* é criado a partir do *coobjecto* documento estruturado base especificando qual o subobjecto que é a raiz do documento. Todos os outros componentes usados são iguais.

Para que o editor possa manipular todos os documento derivados do *coobjecto* documento estruturado base, definiram-se duas interfaces de introspecção que devem ser implementadas pelos subobjectos criados. Por exemplo, a interface relativa a uma lista de referências, indica quais os tipos de elementos que podem ser inseridos em cada posição.

5.1.2 Replicação secundária parcial

A definição de um documento estruturado como uma árvore de subobjectos permite explorar o mecanismo de replicação parcial de um *coobjecto*. Assim, cada cliente pode obter, para cada documento, uma cópia parcial que contenha apenas os elementos da subárvore nos quais o utilizador está interessado.

Como os elementos de um documento estruturado são alcançados por navegação a partir do subobjecto raiz, é importante que quando se obtém uma cópia de um subobjecto, se obtenha também a cópia de todas as listas que constituem o seu caminho desde a raiz. A estratégia de replicação prévia (*pre-fetching*) deve garantir esta propriedade.

Para auxiliar a estratégia de replicação prévia, cada subobjecto mantém a lista dos subobjectos que o referenciam (no componente de atributos do sistema associados ao subobjecto). As operações definidas no subobjecto lista de referências procedem a essa actualização em cada réplica. A estratégia de replicação prévia leva a que, ao ser replicado um subobjecto, sejam igualmente replicados todos os subobjectos que o referenciam.

Uma organização que poderia ter sido usada em alternativa consiste em manter toda a árvore do documento, com excepção dos elementos básicos, num único subobjecto. Como a estrutura apenas contém listas de referências, a dimensão deste subobjecto é razoável. Assim, uma cópia parcial de um documento que inclua este subobjecto permite sempre alcançar todos os elementos básicos.

5.1.3 Invocação cega

O mecanismo de invocação cega permite aos utilizadores modificar elementos não incluídos na cópia parcial de um documento estruturado. Para que se possa observar o efeito esperado das operações executadas usam-se cópias de substituição. A cópia de substituição de um elemento básico não contém nenhuma versão. A cópia de substituição de uma lista de referências contém uma lista sem elementos.

Os utilizadores podem usar estas cópias de substituição para executar as operações normais. Num elemento básico, é possível criar novas versões e modificar ou remover as versões criadas durante a sessão de edição. Numa lista de referências, é possível inserir uma referência para um novo elemento e mover ou remover uma referência entretanto inserida.

Os elementos do documento estruturado que são cópias de substituição são mostrados aos utilizadores numa cor diferente. Por exemplo, na figura 5.4 pode observar-se que esses elementos são represen-

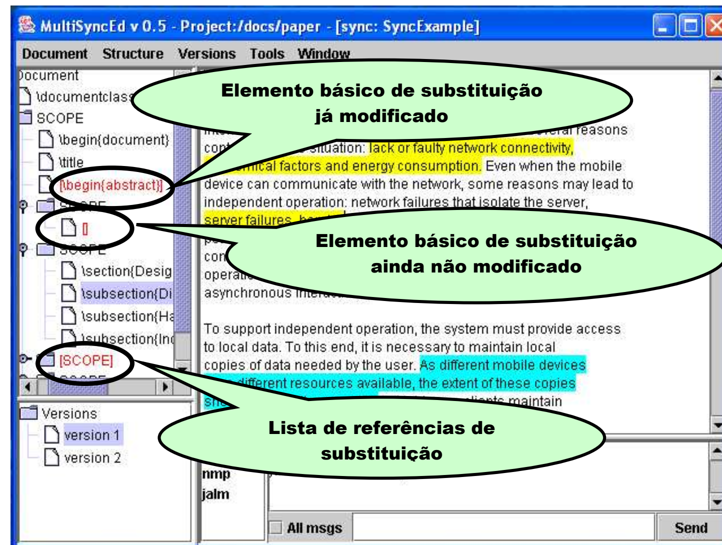

Figura 5.4: Cópias de substituição na edição de um documento estruturado.

tados em cor vermelha na janela que apresenta a estrutura do documento. Desta forma, os utilizadores sabem que não estão a observar uma versão completa do elemento.

5.1.4 Edição assíncrona

O editor multi-síncrono permite a edição assíncrona de um documento. Assim, cada utilizador pode modificar um documento estruturado de forma independente. O *coobjecto* documento estruturado base define a estratégia de reconciliação que se executa para tratar as modificações executadas concorrentemente — como se descreveu, esta estratégia combina a utilização do componente de reconciliação com o código das operações definidas nos subobjectos.

5.1.5 Edição síncrona

O editor multi-síncrono permite, ainda, que um grupo de utilizadores participe numa sessão síncrona para modificar um documento estruturado. Para tal, utiliza-se a estratégia detalhada na secção 4.4.

Assim, qualquer utilizador pode iniciar uma sessão síncrona para edição de um documento, usando a sua cópia privada como estado inicial. Novos participantes podem, posteriormente, juntar-se à sessão obtendo uma cópia do *coobjecto*.

Para manter o estado das várias cópias do *coobjecto* sincronamente sincronizadas usa-se o componente de adaptação síncrono. Como se disse, este componente usa uma aproximação pessimista na propagação das invocações efectuadas: apenas entrega as invocações para execução local após elas terem sido ordenadas pelo sistema de comunicação em grupo. Assim, para garantir a consistência das réplica, usa-se o componente de reconciliação simples, o qual se limita a executar as operações assim

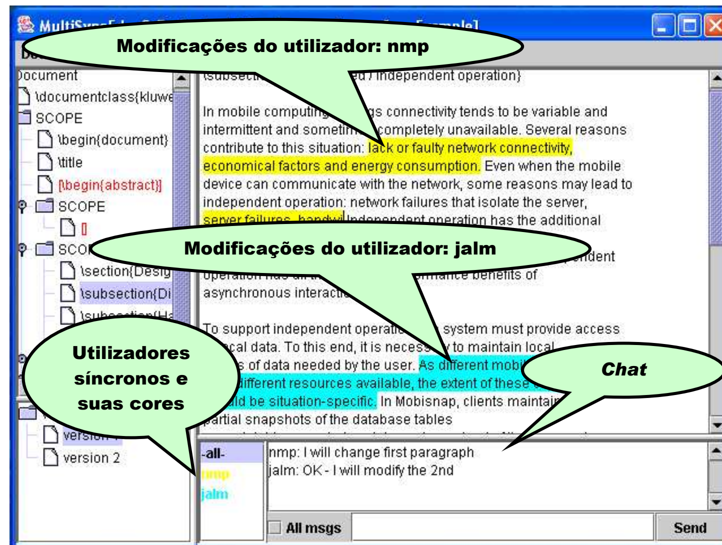

Figura 5.5: Elementos de edição síncrona no editor multi-síncrono.

que elas são conhecidas[4].

Para permitir a edição síncrona de uma versão de um elemento básico adopta-se a alternativa de manipular o seu conteúdo de forma exterior ao *coobjecto* (este problema foi discutido na secção 4.4.1). Assim, cada versão é modificada no âmbito de uma subsessão síncrona. Quando todos os participantes de uma subsessão concluem as suas modificações, submete-se no elemento básico respectivo a operação de modificação que substitui o conteúdo da antiga versão pelo resultado da edição síncrona.

Na figura 5.5 pode observar-se a edição síncrona de uma versão. Para que um utilizador tenha consciência das modificações produzidas por cada participante na sessão, as modificações são marcadas com cores diferentes. Adicionalmente, o editor apresenta a informação de quais os participantes e fornece uma pequena ferramenta de troca interactiva de mensagens (*chat*) que permite a comunicação entre os vários utilizadores.

5.2 Agenda partilhada

Nesta secção descreve-se uma aplicação que permite manipular uma agenda partilhada por vários utilizadores. Esta agenda pode ser usada para manter as reservas efectuadas para um recurso partilhado (por exemplo, uma sala de reuniões) ou como uma agenda pessoal acedida por mais do que uma pessoa (por exemplo, o próprio e a sua secretária).

Nesta aplicação, múltiplos utilizadores podem, independentemente, solicitar a introdução de uma

[4]Para garantir que as intenções dos utilizadores são respeitadas, quando se executam concorrentemente duas operações sobre uma mesma lista de referências, poderia ter sido usado um componente de reconciliação que executasse um algoritmo de transformação de operações.

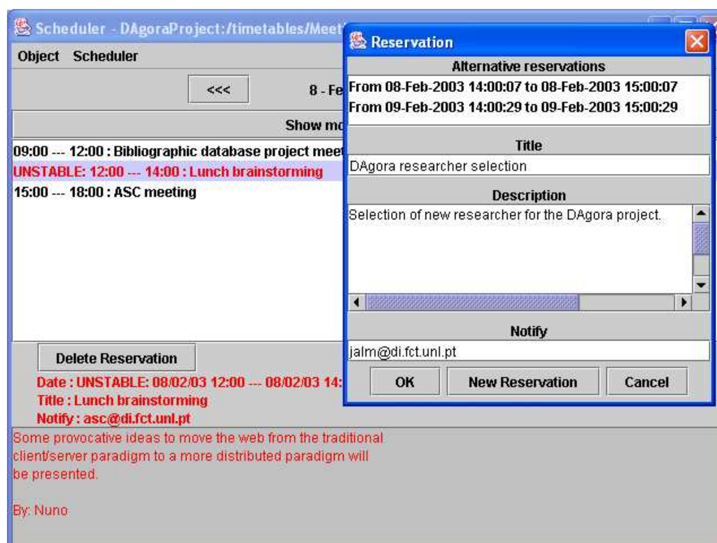

Figura 5.6: Aplicação de agenda partilhado.

nova marcação. No entanto, como é impossível introduzir duas marcações com horários sobrepostos é necessário definir um esquema que garanta a consistência final das várias réplicas da agenda. Este esquema deve garantir que existe um momento (tão breve quanto possível) a partir do qual o resultado definitivo de um pedido de nova marcação é conhecido. No entanto, para evitar que os utilizadores efectuem pedidos de marcações com horários sobrepostos a marcações com horário ainda não confirmado, a agenda deve incluir o resultado provisório das marcações não confirmadas.

Quando efectua um pedido de nova marcação um utilizador pode indicar vários períodos de tempo alternativos, assim reduzindo a probabilidade de ter o seu pedido recusado. Quando o resultado final de um pedido é determinado, deve notificar-se o utilizador do mesmo (caso este o pretenda). Na figura 5.6, observa-se o pedido de uma nova marcação, que inclui a descrição da marcação, a sequência de uma sequência de períodos de tempo alternativos e o endereço para o qual o resultado definitivo deve ser enviado (o transporte apropriado para o endereço indicado será usado).

Para implementar esta aplicação no sistema DOORS é necessário criar o *coobjecto* que mantém a agenda partilhada. A aplicação que manipula o *coobjecto* fornece a interface gráfico que permite aos utilizadores efectuarem novos pedidos e observarem as marcações efectuadas anteriormente.

5.2.1 *Coobjecto* agenda

A informação de uma agenda é armazenada nos seguintes subobjectos.

Primeiro, um subobjecto agenda semanal, que mantém as marcações de uma semana. Este subobjecto inclui duas operações de modificação para inserir e remover uma marcação num dado horário. Para que a intenção do utilizador seja respeitada, quando uma operação de remoção é executada (removendo a

marcação que o utilizador indicou e não outra marcação escalada para a mesma hora), cada marcação tem um identificador único.

Segundo, um subobjecto agenda global, que mantém referências para os subobjectos anteriores, um por cada semana que contém marcações. Este subobjecto inclui duas operações de modificação para inserir e remover uma marcação num dado horário[5]. A execução destas operações consiste em invocar a operação correspondente no subobjecto agenda semanal relativo à semana indicada. Em consequência da execução de uma operação de inserção pode ser criado um novo subobjecto agenda semanal. Em consequência da execução duma operação de remoção pode ser removida a referência para um subobjecto agenda semanal.

Terceiro, um subobjecto central de marcações, sem estado interno e que define as operações para inserir e remover uma marcação com múltiplos horários alternativos. A execução destas operações consiste em invocar, sucessivamente, para cada um dos horários alternativos indicados até que um possa ser executado com sucesso, a operação correspondente no subobjecto agenda global. Adicionalmente, estas operações produzem as mensagens de notificação correspondentes ao seu resultado.

O estado inicial de uma agenda inclui dois subobjectos raiz: um subobjecto agenda global sem nenhuma referência e um subobjecto central de marcações. O único gestor de subobjectos implementado é usado para cada uma das versões[6].

O *coobjecto* agenda mantém duas versões do seu estado — para tal, é baseado na cápsula dupla. A versão definitiva reflecte a execução de todos os pedidos de marcação cujo resultado se encontra garantido. A versão provisória reflecte, adicionalmente, a execução de todas as outras marcações conhecidas. A utilização de duas versões permite utilizar subobjectos que representem uma agenda normal, i.e., sem que cada marcação tenha associado o estado *definitivo* ou *provisório*.

No servidor, usam-se os seguintes componentes. Para garantir a consistência final das várias réplicas, as versões definitiva e provisória são actualizadas, respectivamente, pela versão pessimista e optimista do componente de reconciliação que executa as operações por uma ordem total definida por uma réplica sequenciadora. Esta aproximação, permite estabelecer a ordem de execução definitiva de uma operação (e consequentemente o seu resultado final) desde que a réplica principal esteja acessível.

Para propagar o resultado definitivo das operações, associa-se à versão definitiva a composição do componente de *awareness* que propaga as mensagens produzidas para os utilizadores com o componente

[5]Este subobjecto contém uma operação adicional que remove todas as marcações de uma semana. Esta operação é usada para remover marcações antigas.

[6]Como é de supor que a maior parte dos subobjectos tenham o mesmo estado em ambas as versões, uma optimização possível consiste em manter em disco, para os subobjecto iguais, apenas uma versão. Uma nova implementação do gestor de subobjectos pode fazer esta verificação de forma simples. A manutenção de apenas uma cópia em memória parece possível através da modificação do funcionamento dos representantes dos subobjectos, embora ainda seja necessário investigar todas as implicações de uma aproximação desse tipo.

de *awareness* que garante que apenas uma mensagem é enviada. As mensagens produzidas na versão provisória são descartadas pelo componente de *awareness* que descarta todas as mensagens recebidas.

No cliente, a versão provisória é actualizada pelo componente de reconciliação que executa imediatamente as operações submetidas. A versão definitiva não é modificada — usa-se um componente de reconciliação que não executa operações. Para ambas as versões usa-se o componente de *awareness* que descarta as mensagens recebidas.

Quer no cliente, quer no servidor, usa-se a implementação mais simples disponível para o componente de adaptação. Para o componente de registo e de atributos usam-se as implementações mais simples que permitam usar uma réplica sequenciadora.

5.2.2 Replicação secundária parcial

O agrupamento de todas as marcações relativas a uma semana num subobjecto permite que a cópia parcial de um cliente inclua apenas um pequeno subconjunto de todas as marcações — em geral, um utilizador apenas está interessado na semana actual e (possivelmente) em algumas das semanas seguintes. Qualquer cópia parcial deve incluir ainda o subobjecto agenda global, o qual permite aceder às agendas semanais. Para tal, a estratégia de replicação prévia deve garantir que a agenda global está incluída em qualquer cópia parcial (para tal, o identificador do subobjecto agenda global está incluído na lista de subobjectos relacionados em todos os subobjectos agenda semanal).

Os subobjectos definidos (com excepção do subobjecto central de reservas, explicado de seguida) surgem como elementos naturais na representação de uma agenda — outros elementos poderiam ter sido definidos, como, por exemplo, a agenda diária. Assim, a necessidade de agrupar os objectos que mantêm os dados de um *coobjecto* em subobjectos, para que o mecanismo de replicação secundária parcial possa ser utilizado, não constitui problema neste caso.

5.2.3 Invocação cega

O *coobjecto* agenda permite a invocação cega de operações, independentemente dos subobjectos localmente replicados. Como o subobjecto central de marcações é um subobjecto raiz, a sua referência está sempre disponível em qualquer *coobjecto*. Assim, é possível ao utilizador submeter novos pedidos, mesmo que os subobjectos que contêm as datas referidas não estejam disponíveis localmente (como é normal, o resultado de um pedido apenas é conhecido definitivamente quando a operação é executada num servidor, no qual todos os subobjectos estão disponíveis).

Para permitir que o utilizador observe o resultado esperado do seu pedido, usam-se cópias de substituição. Assim, o utilizador pode igualmente remover um pedido executado anteriormente durante a manipulação da agenda. Apesar de possível, a submissão de uma operação de remoção relativa a uma

marcação que o utilizador sabe existir na agenda é problemática na prática, pois requer o conhecimento do identificador único associado à marcação.

5.3 Outras aplicações

O sistema DOORS foi ainda usado como repositório de dados para um conjunto de outras aplicações (mormente efectuadas no âmbito de projectos de fim de licenciatura de alunos do curso de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa). Entre estes, merece realce a aplicação de base de dados discográfica cooperativa [167].

Esta aplicação permite a um conjunto de utilizadores manter informação partilhada sobre referências discográficas (incluindo o nome dos álbuns, nome das músicas, nome dos autores, apontadores na Internet, etc.). Cada utilizador pode atribuir uma classificação a cada referência. Adicionalmente, para cada referência existe um fórum de discussão.

Toda a informação mantida por esta aplicação é guardada em apenas um *coobjecto*. Este *coobjecto* armazena a informação numa base de dados relacional usando o subobjecto que implementa a interface de uma base de dados relacional. Para garantir a convergência final das várias réplicas do *coobjecto*, usa-se o componente de reconciliação ordem total optimista baseada num sequenciador e usando a técnica *desfazer-refazer*. Finalmente, usa-se o componente de *awareness* que permite notificar activamente os utilizadores (por exemplo, os utilizadores podem solicitar ser informados da adição de uma nova referência com um dado conjunto de características).

Esta aplicação exemplifica a utilização do sistema DOORS na replicação de uma base de dados relacional (convencional). As réplicas da base de dados são distribuídas por diferentes computadores e podem ser acedidas de forma independente (por exemplo, diferentes Intranets podem conter diferentes réplicas sincronizadas através de mensagens de correio electrónico). No entanto, o funcionamento dos *coobjectos* garante que as várias réplicas convergem para o mesmo estado.

Capítulo 6

Núcleo do sistema DOORS

O modelo do sistema DOORS, assim como as suas principais características e modo de utilização do mesmo modelo para suportar a gestão de dados partilhados, foi apresentado nos capítulos anteriores. O modelo descrito é suportado por um conjunto de serviços implementados pelo núcleo do sistema DOORS. Estes serviços executam as tarefas comuns indispensáveis ao funcionamento dos *coobjectos*. Neste capítulo, apresentam-se esses serviços e descreve-se a implementação efectuada no protótipo do sistema DOORS.

Na secção 6.1 descrevem-se os aspectos gerais relativos aos *coobjectos*, incluindo o modo como os mesmos são identificados. As secções seguintes detalham os serviços disponíveis nos servidores (secção 6.2) e nos clientes (secção 6.3).

6.1 Coobjectos

No sistema DOORS, cada *coobjecto* é identificado univocamente através de um identificador global único, $id_{coobj} \equiv (id_{volume}, id_{local})$, em que id_{volume} é o identificador único do volume no qual o *coobjecto* está armazenado e id_{local} é o identificador único do *coobjecto* no volume.

Cada subobjecto é identificado univocamente através de identificador global único, $id_{subobj} \equiv (id_{coobj}, id_{interno})$, em que id_{coob} é o identificador do *coobjecto* a que pertence e $id_{interno}$ é o identificador único do subobjecto no *coobjecto*.

O sistema DOORS inclui ainda um sistema de nomes que permite aos utilizadores usarem nomes simbólicos para designar os *coobjectos*. Para tal, uma camada de designação, descrita na secção 6.1.6, faz a conversão entre os nomes simbólicos e os identificadores internos utilizados no sistema.

```
// Lê um coobjecto existente
public CoObject getCoObject( CoobjId id, int flags);

// "Grava" as modificações efectuadas no coobjecto
public void putCoObject( CoObject proxy);

// Armazena o coobjecto dado no sistema com o nome "id"
public void putCoObject( CoObject proxy, CoobjId id);

// Remove o coobjecto "id"
public void deleteCoObject( CoobjId id);
```

Figura 6.1: API do sistema DOORS para manipulação dos *coobjectos*.

6.1.1 Criação de um *coobjecto*

A criação de um *coobjecto* é efectuada da seguinte forma. Primeiro, uma aplicação cria, de forma exterior ao sistema DOORS, uma cópia do *coobjecto* em memória[1]. Esta cópia é composta por instâncias dos vários componentes do *coobjecto*, incluindo um conjunto de subobjectos com o estado inicial do *coobjecto*.

Segundo, a aplicação grava o *coobjecto* no sistema DOORS invocando a função da API do sistema respectiva (a API do sistema relativa à manipulação de *coobjectos* é apresentada na figura 6.1). Neste momento, é atribuído um identificador único, id_{coobj}, ao *coobjecto* que passa a ser gerido pelo sistema DOORS.

O estado do *coobjecto* no momento da sua primeira gravação (incluindo todos os seus subobjectos) representa o seu estado inicial. O estado inicial do *coobjecto* é armazenado nos recursos disponibilizados pelo sistema usando as funções definidas no próprio *coobjecto*. Quando um *coobjecto* é inicialmente criado num cliente, o seu estado inicial é propagado para o servidor, onde se cria uma réplica do *coobjecto*. Esta réplica é criada usando um método da fábrica do *coobjecto* a partir do estado inicial gravado no cliente.

A partir deste momento, todas as modificações executadas no *coobjecto* são propagadas através de sequências de operações registadas automaticamente pelo *coobjecto*.

6.1.2 Criação de um subobjecto

Um subobjecto é sempre criado no âmbito de um *coobjecto*. Assim, a criação de um subobjecto corresponde a criar o(s) objecto(s) que representa(m) o subobjecto e a associá-lo com um *coobjecto* — estas operações são executadas automaticamente pelo método respectivo definido na fábrica do subobjecto. Durante a associação de um subobjecto com um *coobjecto*, atribui-se ao subobjecto o seu identificador único e regista-se, de forma transparente, a operação de criação do subobjecto (com o seu estado inicial). Esta operação, ao ser executada em cada uma das réplicas, cria a cópia inicial do subobjecto com

[1] A fábrica do *coobjecto*, criada pelo pré-processador, inclui métodos para executar esta operação no cliente e no servidor.

o mesmo estado inicial em todas as réplicas.

Após a sua criação, os subobjectos são modificados através da execução das operações definidas na sua interface. Estas operações são registadas e propagadas para todas as réplicas, como se explicou anteriormente. As operações de todos os subobjectos de um *coobjecto* são tratadas conjuntamente.

6.1.3 Remoção de um *coobjecto*

Um *coobjecto* é removido executando a função de remoção definida na API do sistema. Esta função limita-se a invocar a operação de remoção definida no componente de atributos do *coobjecto*. Assim, a informação sobre se um *coobjecto* foi removido ou não é mantida pelo próprio.

Quando a operação de remoção é executada numa réplica, o *coobjecto* passa ao estado de removido e é impedido qualquer acesso por parte dos clientes. No entanto, a réplica do *coobjecto* não é imediatamente eliminada, de forma a permitir a propagação da operação de remoção para as outras réplicas. O servidor apenas liberta os recursos relativos a um *coobjecto* removido depois de obter informação da propagação da operação de remoção para todos os outros servidores. O servidor mantém, durante um período de tempo longo, definido pelo administrador do sistema, para cada *coobjecto* eliminado, o seu identificador e o sumário das operações conhecidas no momento da eliminação. Esta informação é usada, durante as sessões de sincronização, para informar os outros servidores que o *coobjecto* pode ser eliminado caso não sejam conhecidas novas operações nesses servidores.

Se, antes de um *coobjecto* ser eliminado, for recebida uma operação executada concorrentemente com a operação de remoção, a remoção é interrompida e o *coobjecto* passa ao estado de "não removido", podendo ser acedido de forma normal. A propagação dessa operação leva a que os outros servidores passem igualmente ao estado de "não removido". Caso o *coobjecto* já tenha sido eliminado em algum servidor, esse servidor obterá uma nova cópia a partir de um dos outros servidores (de forma idêntica à obtenção de um cópia de um novo *coobjecto*)[2].

6.1.4 Remoção de um subobjecto

A remoção dos subobjectos depende do gestor de subobjectos utilizado. Em geral, podem-se utilizar técnicas de reciclagem automática ou remoção explícita. Este problema foi abordado na secção 3.3.8.

6.1.5 Versão dos *coobjectos*

Como se descreveu na secção 4.1.1, cada sequência de operações é identificada através do par (srv_{view}, n_seq), em que srv_{view} identifica univocamente um servidor (como se detalha na secção 6.2.1) e

[2]Este processo pode causar a não execução dos blocos *apenas uma vez* de algumas operações.

n_seq identifica univocamente a sequência de operações no conjunto das sequências de operações recebidas nesse servidor. Com base nestes identificadores, cada *coobjecto* mantém um sumário (vector versão) das operações conhecidas localmente, um sumário (vector versão) das operações executadas localmente, e um sumário (vector versão ou matriz) das operações que se sabe serem conhecidas nas outras réplicas. Estes sumários são actualizados durante as sessões de sincronização e aquando da recepção e execução das operações.

O sumário das operações executadas localmente (conjuntamente com o identificador da vista no qual ele é válido) chama-se *versão do coobjecto*.

6.1.6 Sistema de nomes

A camada de designação simbólica permite às aplicações usar nomes simbólicos para se referirem aos volumes e aos *coobjectos*. Assim, esta camada define um conjunto de funções que permitem manipular os volumes e os *coobjectos* usando nomes simbólicos. Estas funções convertem os nomes simbólicos nos identificadores únicos usados pelo sistema e invocam as operações respectivas na API do sistema. O nome simbólico de um *coobjecto* tem a seguinte forma: $[//servidor][/volume:]nome$, com *servidor* o nome do servidor em que o nome deve ser resolvido, *volume* o nome do volume e *nome* o nome do *coobjecto* que se pretende designar.

Cada volume possui um identificador único gerado automaticamente aquando da sua criação e um nome simbólico especificado pelo utilizador. Quando se usa um nome simbólico para identificar um volume, o nome é resolvido no servidor indicado. Caso o servidor não seja indicado, o nome é resolvido localmente. Ocorre um erro caso não se conheça nenhum volume com o nome indicado ou se existir mais do que um volume com esse nome. Nestes casos, a aplicação deve usar uma função do sistema (*listVolumeIds*) para obter a lista de identificadores de volumes com um dado nome — o resultado desta operação não é necessariamente completo (a execução da operação recorre ao serviço de descoberta que se apresenta na secção 6.2.3). Os identificadores dos volumes são expostos como cadeias de caracteres compostas pelo nome simbólico e por uma representação do identificador único. Estes identificadores podem ser usados para designar um volume de forma única.

Cada volume define um espaço hierárquico de nomes (semelhante ao espaço de nomes definido num sistema de ficheiros), em que cada nome é associado a um identificador único de um *coobjecto*. Esta associação é mantida no *coobjecto* directório global, idêntico ao descrito na secção 3.3.10[3] Este *coobjecto* tem um identificador único bem-definido no volume (igual para todos os volumes).

Quando um *coobjecto* é gravado pela primeira vez, a camada de designação adiciona ao directório

[3] Relativamente à descrição da secção 3.3.10, este *coobjecto* permite a definição adicional de ligações simbólicas (*symbolic links*) através de um subobjecto criado para o efeito.

global a associação entre o nome atribuído pela aplicação e o identificador único do *coobjecto*. A interpretação de um nome simbólico é efectuada obtendo, no *coobjecto* directório global, o identificador único associado a esse nome.

Quando se executa a função da API do sistema (definida na camada de designação) para remover um *coobjecto*, remove-se adicionalmente a associação entre o nome e o identificador desse *coobjecto* (submetendo a operação de remoção respectiva no directório global). Quando um *coobjecto* removido é "ressuscitado" pela existência de operações concorrentes com a operação de remoção, é criada uma nova associação entre um nome pertencente a um directório especial e o identificador único do *coobjecto*. A adição do novo nome é submetida apenas no servidor que recebeu a operação de remoção do *coobjecto*. Esta responsabilidade é delegada (no patrocinador) no caso de o servidor cessar a replicação do volume.

6.2 Servidores

Os servidores replicam volumes de *coobjectos* usando uma estratégia optimista. Esta aproximação permite fornecer aos clientes um elevada disponibilidade dos dados para operações de leitura e escrita, permitindo mascarar (algumas) falhas nos servidores e no sistema de comunicações.

Um servidor é responsável por gerir a cópia local de cada volume, incluindo uma cópia de cada *coobjecto* contido no volume replicado. Os servidores comunicam entre si para sincronizar o estado das réplicas locais. Adicionalmente, os servidores fornecem uma interface para administrar o servidor e aceder aos *coobjectos*. Assim, os utilizadores (administradores do sistema) podem criar novos volumes e gerir o conjunto de servidores que replicam cada volume. Para tal, os utilizadores submetem as operações que desejam efectuar num cliente que as propaga para o(s) servidor(es) apropriado(s). Os clientes podem ainda obter cópias de *coobjectos* e submeter operações de modificação (em consequência das acções executadas pelos utilizadores). Nesta secção descreve-se o funcionamento do servidor, incluindo os vários protocolos executados no protótipo do sistema DOORS.

6.2.1 Volumes e filiação

Um volume é um contentor de *coobjectos* replicado por um grupo variável de servidores. Um volume é criado num servidor em resultado de uma operação (*createVolume*) submetida por um utilizador (administrador do sistema). Um servidor inicia ou cessa a replicação de um volume em resultado de operações (*replicateVolume* e *unreplicateVolume* respectivamente) submetidas pelos utilizadores. Estas operações levam à execução de protocolos de inserção/remoção de um servidor no grupo de replicadores de um volume. Estes protocolos são iniciados pelo servidor que inicia/cessa a replicação do volume e têm a participação de apenas um outro servidor, que se designa de patrocinador.

De seguida, descreve-se o funcionamento do sistema de filiação implementado no sistema DOORS. Nesta descrição, indicam-se as acções a executar pelos servidores no âmbito da gestão de filiação de um dado volume, sendo que a gestão da filiação de cada volume é executada de forma independente.

***Coobjecto* de filiação** Em cada volume, a informação sobre o grupo de servidores que replica o volume é mantida num *coobjecto* especial manipulado pelo sistema, o *coobjecto* de filiação. Este *coobjecto* define três operações de modificação: inserção de um novo servidor; remoção de um servidor; e eliminação das referências relativas a um servidor anteriormente removido.

O estado do *coobjecto* de filiação define, em cada momento, uma vista (*view*) do grupo de replicadores do volume que inclui: um identificador da vista; a identificação do conjunto de replicadores; e uma função bijectiva que atribui a cada replicador do volume um número inteiro, srv_{view}, que o identifica na vista. Este identificador (srv_{view}) é usado na identificação das operações, como se explicou na secção 4.1.1.

Como se detalha no apêndice A.1.1, o funcionamento deste *coobjecto* garante que duas cópias que tenham conhecimento do mesmo conjunto de operações têm o mesmo estado (i.e., estão na mesma vista). Adicionalmente, garante-se que o identificador da vista a identifica univocamente, i.e., a cada conjunto de operações corresponde um e um só identificador e a cada identificador corresponde um e um só conjunto de operações.

Protocolo local de mudança de vista Como se referiu anteriormente, as operações executadas em qualquer *coobjecto* de um volume são identificadas com um par que inclui o identificador do servidor na vista. Os sumários de operações mantidos são igualmente baseados nestes identificadores. Assim, é necessário actualizar estes identificadores sempre que se *instala* uma nova vista. Para tal, definiu-se o *protocolo local de mudança de vista*, detalhado no apêndice A.1.2.

Este protocolo actualiza, de forma atómica, todos os *coobjectos* de um volume. O modo como o *coobjecto* de filiação é actualizado leva a que apenas sejam necessárias actualizações quando se verificam duas ou mais inserções concorrentes (e em que aos novos servidores foram atribuídos, na execução optimista, o mesmo identificador em diferentes cópias do mesmo volume) ou uma inserção concorrente com uma eliminação (em que o novo servidor pode reutilizar o identificador do servidor eliminado)[4].

Com base no *coobjecto* de filiação e no protocolo local de mudança de vista, os protocolos de inserção e remoção voluntária de um servidor no grupo são bastante simples e envolvem a comunicação apenas entre o servidor alvo da acção e outro servidor que pertença ao grupo de replicadores do volume. Este

[4]Uma aproximação alternativa consiste em utilizar identificadores globais dos servidores. Assim, não é necessário actualizar os identificadores quando uma nova vista é instalada, mas o espaço ocupado por estes é consideravelmente maior e a implementação e comparação dos vectores versão mais complexa

segundo servidor designa-se por patrocinador.

Protocolo de entrada no grupo de replicadores Para um servidor iniciar a replicação de um volume é necessário actualizar a informação relativa ao grupo de replicadores do volume e transferir uma cópia do volume para o novo servidor.

O *protocolo de entrada no grupo de replicadores*, detalhado no apêndice A.1.3, é iniciado pelo servidor que pretende iniciar a replicação do volume. Este servidor contacta o patrocinador informando-o da sua intenção. O patrocinador executa a operação de inserção respectiva no *coobjecto* de filiação e actualiza a réplica do volume usando o protocolo local de mudança de vista. Finalmente, o patrocinador propaga para o novo servidor, no âmbito de uma sessão de sincronização epidémica, uma cópia do volume, incluindo todos os *coobjectos*.

Protocolo de saída do grupo de replicadores Quando um servidor cessa voluntariamente a replicação de um volume é necessário que não se perca nenhuma informação que se encontre apenas nesse servidor (i.e., operações relativas a *coobjectos* do volume, incluindo o *coobjecto* de filiação, que apenas sejam conhecidas nesse servidor). Adicionalmente, a filiação do grupo de replicadores deve ser actualizada e os recursos usados para replicar esse volume devem ser libertos.

O *protocolo de saída do grupo de replicadores* de um volume, detalhado no apêndice A.1.4, é iniciado pelo servidor i que pretende cessar a replicação do volume. Este servidor coloca-se no estado de pré-removido e não executa mais operações relativas a esse volume. Neste momento, todos os *coobjectos* do volume são notificados que o servidor cessará a replicação do volume[5]. O servidor i informa o patrocinador da sua intenção, o qual notifica a réplica local de todos os *coobjectos* do volume desse facto. De seguida, o patrocinador estabelece uma sessão de sincronização com o servidor i. No fim da sessão de sincronização, o patrocinador executa a operação de remoção respectiva no *coobjecto* de filiação e actualiza a réplica do volume usando o protocolo local de mudança de vista. Após executar todas as acções relativas à sessão de sincronização, o servidor i pode libertar os recursos relativos ao volume.

Eliminação dos identificadores dos servidores removidos Como os identificadores dos servidores nas vistas são usados na identificação das operações, não é possível eliminar imediatamente o identificador associado a um servidor removido. Este identificador apenas pode ser eliminado quando não for necessário no funcionamento dos *coobjectos* — ou seja, quando as operações executadas no servidor removido tiverem sido propagadas e definitivamente processadas em todos os servidores.

[5]Esta informação é importante em algumas soluções de gestão de dados, como ficou patente na discussão da secção 4.1, relativa ao componente de reconciliação.

Para garantir estas propriedades, adicionou-se ao *coobjecto* de filiação uma operação de *disseminação e execução*, detalhada no apêndice A.1.5. Esta operação permite executar uma acção dada como parâmetro apenas após a réplica do volume num servidor reflectir todas as operações conhecidas num dado conjunto de servidores no momento da submissão da operação de *disseminação e execução*.

O *protocolo de eliminação de um identificador*, detalhado no apêndice A.1.6, é iniciado pelo patrocinador da remoção e consiste nos seguintes passos. Primeiro, o patrocinador usa a operação de *disseminação e execução* para garantir que nos vários servidores, apenas se inicia o processo de eliminação do servidor após terem sido propagadas todas as operações de todos os *coobjectos* recebidas do servidor removido. Segundo, cada servidor notifica os outros quando verifica que nenhum *coobjecto* necessita do identificador do servidor removido. Terceiro, quando num servidor se verifica que nenhum servidor activo necessita do identificador de um servidor, é submetida a operação de eliminação do identificador.

Protocolo de saída forçada do grupo de replicadores Em geral, um servidor apenas cessa a replicação de um volume por sua iniciativa. No entanto, existem circunstâncias em que pode ser necessário forçar a remoção de um servidor do grupo de replicadores — por exemplo, porque o servidor ficou danificado e a memória estável na qual estava armazenado o estado dos *coobjectos* é irrecuperável.

O *protocolo de saída forçada do grupo de replicadores*, detalhado no apêndice A.1.7, usa a seguinte aproximação. Primeiro, qualquer servidor pode sugerir a remoção de um (ou mais) servidor(es), mas o processo apenas prosseguirá se todos os outros servidores concordarem com a remoção. Segundo, após eleger um servidor como patrocinador da remoção, é necessário garantir que esse servidor recebe todas as operações conhecidas provenientes do servidor a remover. Terceiro, o servidor informa todos os *coobjectos* do volume da remoção do servidor e, de seguida, executa a operação de remoção no *coobjecto* de filiação no âmbito do protocolo local de mudança de vista.

Protocolo de sincronização de vistas No sistema DOORS, dois servidores apenas podem comunicar para sincronizar o estado dos *coobjectos* se se encontrarem na mesma vista. Assim, dois servidores, i e j, que se encontrem em vistas diferentes devem executar previamente um *protocolo de sincronização de vistas*. Este protocolo consiste na sincronização das operações do *coobjecto* de filiação conhecidas pelos dois servidores — quando cada um dos servidores recebe o conjunto de operações enviadas pelo parceiro efectua o protocolo local de mudança de vista. No final do protocolo, conhecidas as mesmas operações relativas ao *coobjecto* de filiação, os dois servidores encontram-se na mesma vista. Durante a sincronização das operações conhecidas, cada servidor envia para o parceiro apenas as operações que sabe que ele não conhece — usando o sumário das operações conhecidas no parceiro (ou à falta deste sumário) o identificador da vista, que é igualmente um sumário das operações de inserção e remoção que o parceiro conhece.

6.2.2 Sincronização epidémica dos servidores

Os servidores sincronizam as réplicas dos *coobjectos* de um volume durante sessões de sincronização epidémica [38] estabelecidas entre pares de servidores. Durante estas sessões, os servidores propagam entre si as operações de modificação que conhecem (independentemente do servidor no qual foram recebidas pela primeira vez). Desta forma, cada servidor recebe, para cada *coobjecto*, todas as modificações submetidas em todos os servidores, directa ou indirectamente. No sistema DOORS, considera-se que duas réplicas de um mesmo *coobjecto* estão sincronizadas quando conhecem o mesmo conjunto de operações (e os sumários das operações conhecidas e executadas localmente e das operações conhecidas nos outros servidores são idênticos).

No sistema DOORS, definiram-se dois protocolos de propagação epidémica. O primeiro protocolo é bilateral e durante a sua execução cada um dos servidores propaga as operações que conhece para o parceiro. O segundo protocolo é unilateral e apenas um dos servidores propaga as operações conhecidas para o parceiro.

Protocolo de propagação epidémica bilateral O *protocolo de propagação epidémica bilateral*, detalhado no apêndice A.2.2, consiste numa sequência de passos executados assincronamente — em cada um dos passos um servidor processa as mensagens recebidas e envia mensagens para o seu parceiro. Em cada um dos passos, e relativamente a cada um dos *coobjectos* a sincronizar, é possível: (1) solicitar o envio das operações não reflectidas num sumário; (2) enviar um conjunto de operações e solicitar o envio das operações não reflectidas num sumário; (3) solicitar o envio de uma cópia do *coobjecto*; (4) enviar uma cópia do *coobjecto*; (5) enviar a informação que o *coobjecto* foi eliminado.

O protocolo de sincronização bilateral é iniciado com o servidor i a pedir que o parceiro lhe envie, para cada um dos *coobjectos* a sincronizar, as operações não reflectidas no sumário das operações conhecidas em i (este processo é optimizado para que não seja trocada informação relativa aos *coobjectos* estáveis, i.e., que não foram modificados recentemente). O protocolo continua por mais três passos, em que cada um dos parceiros responde aos pedidos recebidos e solicita o envio da informação necessária à sincronização das réplicas dos *coobjectos*.

Protocolo de propagação epidémica unilateral No *protocolo de propagação epidémica unilateral*, detalhado no apêndice A.2.3, um servidor, i, envia para outro servidor, j, a informação que permite a j sincronizar o estado de j com o estado de i (i.e., o servidor j deve ficar a conhecer todas as operações conhecidas em i). Para tal, o servidor i envia para j todas as operações (e novos *coobjectos*) que, de acordo com o sumário das operações conhecidos nos outros servidores, não sabe serem conhecidas em j.

6.2.2.1 Comunicações entre os servidores

Como se descreveu, os protocolos de sincronização implementados no sistema DOORS são intrinsecamente assíncronos. Assim, é possível executar esses protocolos sobre transportes síncronos e assíncronos. Cada servidor pode receber comunicações num subconjunto não vazio de transportes. Para cada volume, o *coobjecto* de filiação mantém informação sobre os transportes implementados em cada servidor.

No protótipo do sistema DOORS foram implementados os seguintes transportes. O primeiro, síncrono ponto-a-ponto, usando directamente o protocolo de comunicações TCP. Neste caso, um servidor aceita continuamente conexões de outros servidores numa porta definida para o efeito.

O segundo, assíncrono ponto-a-ponto, usando correio electrónico. Neste caso, um servidor aceita comunicações de outros servidores através dum endereço de correio electrónico definido para o efeito. O servidor verifica periodicamente a existência de novas mensagens usando o protocolo POP3.

6.2.2.2 Política de sincronização

Cada volume implementa uma *política de sincronização* própria, que define os momentos em que cada sessão de sincronização deve ser estabelecida, quais os servidores envolvidos e quais os transportes a utilizar. O *coobjecto* de filiação de cada volume mantém a definição da política de sincronização como um objecto que implementa uma interface pré-definida. Esta política pode ser alterada pelo administrador do sistema. Com base nesta informação, os servidores são responsáveis por iniciar os protocolos de sincronização nos momentos adequados.

O estudo de políticas de sincronização óptimas para diferentes configurações está fora do âmbito desta dissertação — este assunto foi abordado em vários estudos [38, 57, 84]. No protótipo do sistema DOORS foram pré-definidas as seguintes estratégias simples. A primeira, aleatória, na qual cada servidor selecciona aleatoriamente o parceiro entre os servidores conhecidos. A segunda, utilizando uma configuração em forma de anel estabelecida a partir dos identificadores dos servidores. Em ambos os casos, define-se o momento da próxima sessão de sincronização utilizando uma variável aleatória de distribuição exponencial (com tempo médio definido pelo administrador do sistema).

Durante os períodos de mudança de vista (i.e., quando não se sabe que as operações de entrada e saída do grupo de replicadores foram propagadas para todos os outros servidores), existem servidores que têm uma visão desactualizada (e temporariamente divergente entre si) do conjunto dos replicadores de um volume. Assim, é necessário ter algum cuidado para garantir que as mudanças no grupo de replicadores sejam propagadas para todos os servidores. Nas estratégias pré-definidas, usa-se sempre uma estratégia aleatória com um tempo médio entre sessões muito curto durante os períodos de mudança de vista. Este problema é discutido com maior detalhe no apêndice A.2.4

6.2.3 Serviço de descoberta e disseminação de eventos

Os servidores do sistema DOORS estão interligados através do sistema de disseminação de eventos Deeds [45]. O sistema Deeds fornece um serviço de disseminação de eventos baseado em canais. Os clientes do serviço podem criar novos canais, subscrever canais já existentes e enviar eventos para um canal. Um evento enviado para um canal é propagado para os clientes que subscrevem o canal. As garantias de entrega de um evento a um cliente de um canal dependem da semântica do canal.

Todos os servidores do sistema DOORS subscrevem um canal de disseminação de eventos usado para descobrir quais os servidores que replicam um dado volume. Este canal de disseminação propaga os eventos para todos os servidores com a semântica de propagação "melhor esforço". Quando um cliente necessita de conhecer um servidor que replica um volume até ao momento desconhecido, o cliente submete um evento de descoberta neste canal.

Para cada volume é criado um canal de disseminação usado para propagar de forma expedita as novas operações submetidas no âmbito de um volume, como se detalha no apêndice A.2.5. Quando uma operação é recebida e identificada num servidor, ela é imediatamente propagada para os outros servidores através deste canal de disseminação. Este canal usa uma semântica de propagação "melhor esforço", pelo que não garante a entrega da operação a todos os servidores que subscrevem o canal. A subscrição do canal por parte de um servidor que replica um volume é opcional — a sincronização epidémica, descrita na secção 6.2.2, é o mecanismo base que garante a sincronização das várias réplicas.

6.2.4 Interacção com os clientes

Os servidores fornecem uma interface aos clientes que permite administrar o servidor e aceder aos *co-objectos* — ver figura 6.2. As operações de administração do servidor permitem criar novos volumes e iniciar/terminar a replicação de um volume num dado servidor. A execução destas operações leva à execução dos protocolos descritos na secção 6.2.1. Adicionalmente, é possível forçar a execução de sessões de sincronização (completas ou parciais) entre dois servidores (relativas a um dado volume). A execução destas operações leva à execução dos protocolos descritos na secção 6.2.2.

As operações definidas para manipular *coobjectos* permitem aos clientes aceder ao estado actual dos *coobjectos* e submeter modificações. De seguida descrevem-se as operações definidas para manipular *coobjectos* e o modo como elas são implementadas.

getCoObject Permite, ao cliente, obter uma cópia de um *coobjecto* e de um conjunto de subobjectos pertencentes a um mesmo *coobjecto*. No caso de o cliente já possuir uma cópia do *coobjecto* (indicando a versão que conhece), o servidor pode enviar a sequência de operações necessárias para actualizar a réplica do cliente. Caso contrário, é necessário propagar uma cópia do *coob-*

```
//------------------ UTILIDADES ----------------------------------------------------------
// Gera semente para gerador de identificadores únicos
UIDSeed genUIDSeed();

//------------------ GESTÃO DE VOLUMES : ADMINISTRADORES ---------------------------------
// Cria novo volume
void createVolume( String volumeName);

// Inicia/cessa replicação do volume indicado
void replicateVolume( String volumeName, MbrshipElement fromServer);
void unreplicateVolume( String volumeName);

// Força sincronização do volume ou coobjecto indicado
void synchronizeVolume( String volumeName, MbrshipElement fromServer);
void synchronizeObject( GlobalOID goid, MbrshipElement fromServer)

// Lista volumes replicados no volume dado
String[] listVolumes();

//------------------ ACESSO AOS COOBJECTOS / SUB-OBJECTOS : UTILIZADORES -----------------
// Obtém cópia/actualização do coobjecto/sub-objectos indicados para actualizar a versão indicada
ExtRepresentation getCoObject( Timevector state, Timevector view, GlobalOID[] goid, int falgs);
ExtRepresentation getCoObject(Timevector state,Timevector view, GlobalOID goid, CachePolicy policy, int flags);
ExtRepresentation getAddSubObject( Timevector state, Timevector view, GlobalOID goid, int flags);

// Submete sequência de operações executadas num coobjecto
OpId submitChanges( GlobalOID goid, Timevector viewID, OperationSeq ops);

// Grava novo coobjecto
void submitNewCoObject( GlobalOID goid, ObjectLLRepresentation objCore);

// Submete uma operação de interrogação/modificação para execução síncrona
byte[] submitSyncQuery( RemoteSession session, OperationSingle op);
byte[] submitSyncOperation( RemoteSession session, OperationSeq op);
```

Figura 6.2: Interface do servidor.

jecto/subobjecto. Do ponto de vista do sistema, o estado de um *coobjecto* (transmitido como resultado desta operação) inclui os atributos do sistema e uma sequência opaca de *bytes* obtida através dum método do *coobjecto*. Os atributos do sistema permitem obter a fábrica do *coobjecto*[6], a qual é usada para armazenar e criar uma cópia do *coobjecto* a partir da sequência de *bytes* recebida do servidor. Os subobjectos são transmitidos para os clientes de forma semelhante aos *coobjectos*, com excepção de a transmissão de um subobjecto (ou conjunto de subobjectos) ter de ser efectuada no âmbito do *coobjecto* em que o subobjecto está incluído (i.e., quando se propaga o estado de um subobjecto é necessário propagar o estado do *coobjecto* correspondente, a menos que a versão actual do *coobjecto* seja conhecida no cliente).

getAddSubObject Permite, ao cliente, obter uma cópia de um subobjecto consistente com a cópia parcial do *coobjecto* que o cliente possui (e cuja versão é indicada na operação). A execução desta operação é semelhante à anterior, com a seguinte diferença: o servidor deve enviar a versão do subobjecto indicada (e não uma versão mais recente) ou devolver um erro. Esta operação permite

[6] O cliente pode ter a necessidade de obter o código da fábrica para completar a recepção da cópia de um *coobjecto*/subobjecto. Para tal, deve obter uma cópia do *coobjecto* que armazena o código da fábrica.

ao cliente completar a sua cópia parcial, como se descreve na secção 6.3.1.

submitChanges Permite, ao cliente, enviar uma sequência de operações executadas num *coobjecto*. Uma sequência de operações contém o identificador da versão à qual as operações foram executadas de forma a permitir traçar as dependências causais das operações. Quando este identificador de versão corresponder a uma vista diferente da vista actualmente instalada, o identificador é actualizado recorrendo a uma função definida no *coobjecto* de filiação. O servidor entrega a sequência de operações recebida à cópia local do *coobjecto* através duma operação definida na interface do *coobjecto*.

submitNewCoObject Permite, ao cliente, enviar para o servidor o estado de um novo *coobjecto* criado no cliente. Como anteriormente, este estado contém os atributos do sistema e uma sequência opaca de *bytes* obtida no cliente através da interface do *coobjecto*. Um método definido na fábrica do *coobjecto* é responsável por criar, usando a sequência de *bytes* recebida do cliente, a cópia inicial do *coobjecto* (incluindo todos os subobjectos definidos inicialmente) no servidor. Para criar um *coobjecto* de um novo tipo, é necessário criar primeiro o *coobjecto* que armazena o código necessário para o instanciar.

submitSyncQuery Permite, a um cliente, enviar uma operação de leitura para execução imediata no servidor. O servidor executa imediatamente a operação na cópia local do *coobjecto* usando a interface do *coobjecto*. O servidor devolve ao cliente, não apenas o resultado da operação executada, mas também, a versão do *coobjecto* à qual foi executada. Esta operação permite especificar condições para a execução da operação — por exemplo, pode indicar-se que apenas se pretende executar a operação numa dada versão do *coobjecto* (ver detalhes na secção 6.3.3). Se o servidor não se encontra nas condições indicadas, a operação não é executada, devolvendo um erro ao cliente.

submitSyncOperation Permite, a um cliente, submeter uma sequência de operações de modificação para execução imediata. Como anteriormente, o servidor entrega imediatamente a operação à cópia local do *coobjecto* para execução. O servidor devolve ao cliente a versão da cópia local do *coobjecto* após as operações terem sido entregues (a execução imediata destas operações na cópia local do servidor não é garantida, pois depende da estratégia de reconciliação usada no *coobjecto*). De forma semelhante à operação anterior, é possível especificar condições para a execução da operação.

Como se observa pelas descrições anteriores, o núcleo do sistema delega nos *coobjectos* a maioria das acções que devem ser executadas. Esta aproximação permite manter o núcleo do sistema simples, ao mesmo tempo que permite a implementação de diferentes soluções de gestão de dados partilhados.

6.2.5 Recursos associados aos *coobjectos*

Em cada servidor, o núcleo do sistema gere um conjunto de recursos associados a cada *coobjecto*. Estes recursos consistem num conjunto de ficheiros e bases de dados usados por cada *coobjecto* para armazenar o seu estado actual. Cada subobjecto pertencente a um *coobjecto* tem igualmente associado um conjunto de recursos geridos pelo sistema.

O núcleo do sistema é responsável por criar, remover e gerir todos os recursos usados pelos *coobjectos*/subobjectos de acordo com os seus pedidos. Para tal, o núcleo do sistema fornece uma interface que permite manipular estes recursos (por exemplo, ler/gravar uma sequência de bytes num ficheiro, questionar e modificar uma base de dados).

O estado de um *coobjecto*/subobjecto é transmitido entre servidores ou para um cliente como uma sequência de *bytes* (opaca para o sistema). Esta sequência é criada por uma função do *coobjecto* que codifica o seu estado actual, incluindo todos os recursos a ele associados.

6.2.6 Suporte para múltiplas bases de dados

Para permitir que os *coobjectos*/subobjectos armazenem o seu estado interno numa base de dados, cada servidor e cada cliente têm associado um sistema de gestão de bases de dados. O núcleo do sistema encarrega-se de criar/destruir/gerir as bases de dados no sistema de gestão de base de dados de acordo com as necessidades dos *coobjectos* (os quais apenas têm acesso a uma referência que lhes permite aceder à base de dados para leitura e escrita).

O protótipo do sistema DOORS permite a utilização de três sistema de gestão de bases de dados: Hypersonic SQL [71] em qualquer plataforma (e usado por omissão); Microsoft Access [104] em plataformas Windows; e Oracle 8 [113] em qualquer plataforma.

Para permitir que em vários servidores sejam usadas diferentes bases de dados de forma transparente, criou-se, para cada base de dados, uma camada de uniformização responsável por fornecer uma visão uniforme do serviço. Esta camada é responsável por fornecer um método uniforme de criar e remover bases de dados. Relativamente às instruções de acesso à base de dados, esta camada devia limitar-se a filtrar as operações permitidas, restringido-as a um subconjunto de instruções universalmente suportadas (aproximadamente as instruções do SQL92 [76]). Na prática, verificou-se que é necessário efectuar pequenas modificações nas instruções submetidas à base de dados, de forma a garantir a utilização transparente de diferentes bases de dados. Entre estas modificações destacam-se:

- Introdução/remoção de um carácter terminador nas instruções de leitura/escrita;
- Conversão dos nomes dos tipos de dados nas instruções de criação de novas tabelas;
- Conversão na forma como se definem restrições na criação de novas tabelas;

- Emulação do tipo de dados inteiro auto-incrementado na base de dados Oracle (combinando a utilização de sequências e gatilhos (*triggers*)).

A ordem pela qual são devolvidos os resultados de uma instrução de leitura (*select*) não se encontra definida. Assim, para que se tenha uma visão uniforme do serviço de base de dados é necessário ordenar o resultado de todas as instruções de leitura (por exemplo, transformando uma instrução *select* numa instrução *select...order by*). No entanto, como a execução de leituras ordenadas pode ser ineficiente e a ordem dos resultados não é importante em muitas situações, decidiu-se não integrar esta funcionalidade na camada de uniformização. Assim, os programadores são responsáveis por efectuar leituras ordenadas sempre que a ordem é importante para o determinismo de uma operação.

6.2.7 Serviço de *awareness*

Cada servidor do sistema DOORS tem associado um serviço de *awareness*. Este serviço é responsável por entregar as mensagens produzidas pelos *coobjectos* aos serviços que fazem a sua entrega final aos utilizadores. Para tal, este serviço executa todos os passos necessários ao contacto fiável com os serviços finais, incluindo o tratamento de falhas temporárias.

O serviço de *awareness* fornece uma interface normalizada para a propagação de mensagens através de diferentes transportes. Adicionalmente, ao propagar as mensagens criadas durante a execução de uma operação de forma assíncrona, permite que a execução das operações nos *coobjectos* seja rápida.

O modo como cada serviço final deve ser contactado é definido em cada servidor. Para tal, define-se o código (*plug-in*) usado para contactar o serviço e os seus parâmetros de configuração específica. Por exemplo, no protótipo do sistema DOORS é possível usar os seguintes tipos de transporte (configuráveis em cada servidor):

Correio electrónico As mensagens são entregues ao servidor de SMTP especificado como parâmetro da configuração.

Mensagens SMS As mensagens são entregues através de um intermediário (*gateway*) acessível por HTTP. O endereço do intermediário é configurável.

Mensagens Deeds As mensagens são entregues ao servidor Deeds local.

Quando um *coobjecto* solicita a propagação de uma mensagem usando um mecanismo desconhecido no servidor, o serviço de *awareness* informa o *coobjecto* que não pode confirmar o processamento da mensagem. O *coobjecto* pode, posteriormente, questionar o serviço de *awareness* sobre a propagação da mensagem.

```
//------------------ GESTÃO DE VOLUMES : ADMINISTRADORES --------------------------------
// Cria novo volume
void createVolume( String inServer, String volumeName);

// Inicia/cassa replicação do volume indicado
void replicateVolume( String inServer, String volumeName, MbrshipElement fromServer);
void unreplicateVolume( String inServer, String volumeName);

// Força sincronização do volume/coobjecto indicado
void synchronizeVolume( String inServer, String volumeName, String fromServer);
void synchronizeObject( String inServer, CoobjId id, String fromServer)

// Lista volumes replicados num servidor
String[] listVolumes( String server);

// Lista identificadores de volumes com um dado nome
String[] listVolumeIds( String volumeName, int wait);

//------------------ ACESSO AOS COOBJECTOS : UTILIZADORES --------------------------------
// Lê um coobjecto existente
public CoObject getCoObject( CoobjId id, int flags);
public CoObject getCoObject( CoobjId id, CachePolicy policy, int flags);

// "Grava" as modificações efectuadas no coobjecto
public void putCoObject( CoObject proxy);

// Armazena o coobjecto dado no sistema com o nome "id"
public void putCoObject( CoObject proxy, CoobjId id);

// Remove o coobjecto "id"
public void deleteCoObject( CoobjId id);

// Lê/grava atributos de um coobjecto existente
public CoObjectAttrib getCoObjectAttrib( CoobjId id, int flags);
void putCoObjectAttrib( CoObjectAttrib attrib); //sistema
```

Figura 6.3: API do sistema DOORS.

Assincronamente, o serviço de *awareness* usa o serviço de descoberta para procurar outros servidores que saibam propagar a mensagem usando o método solicitado pelo *coobjecto*. Se possível, o servidor importa o código e as definições necessárias para processar localmente as mensagens. Caso tal não seja possível, o servidor propaga as mensagens para serem processadas noutro servidor.

6.3 Clientes

Os clientes do sistema DOORS fornecem às aplicações uma interface que lhes permite aceder aos *coobjectos* e administrar os servidores — ver figura 6.3. Para fornecerem estes serviços, os clientes acedem às operações disponibilizadas pelos servidores descritas na secção 6.2.4. Adicionalmente, para permitir o acesso aos dados em situações de desconexão, os clientes mantêm cópias parciais dos *coobjectos*. O modo de funcionamento do cliente é detalhado nesta secção.

6.3.1 Replicação secundária parcial

Os clientes mantêm cópias parciais dos *coobjectos*. Uma cópia parcial de um *coobjecto* consiste na parte comum do *coobjecto* e num subconjunto dos subobjectos contidos no *coobjecto*. A parte comum

do *coobjecto* consiste nos componentes necessários (cápsula, atributos do sistema, etc.) à criação de uma cópia do *coobjecto*. De forma semelhante ao servidor, para cada *coobjecto*/subobjecto replicado localmente, o cliente mantém um conjunto de recursos no qual está armazenado o seu estado actual.

Os clientes obtém as cópias parciais a partir de um qualquer servidor. Com cada cópia de um *coobjecto* (e respectivos subobjectos), o cliente mantém o identificador da vista no servidor quando a cópia foi obtida. Este identificador da vista é usado em todas as interacções com os servidores para que estes possam interpretar correctamente os identificadores das operações (e vectores versão).

Actualização da cópia parcial de um *coobjecto* Uma cópia parcial de um *coobjecto* pode ser actualizada a partir de qualquer servidor. Se o servidor tem instalada uma vista diferente da vista usada na cópia do cliente, o cliente deve contactar um novo servidor ou obter uma cópia nova a partir deste servidor.

Para determinar se a cópia de um cliente necessita de ser actualizada, o servidor compara a versão do *coobjecto* da sua cópia local, $o.v$, e a versão do *coobjecto* que o cliente possui, rv. Se os dois vectores forem iguais ($o.v[i] = rv[i], \forall i$), a cópia do cliente está actualizada. Se o vector do servidor for *superior* ao do cliente, i.e., $o.v[i] \geq rv[i], \forall i$, o servidor tem uma versão mais recente e a versão do cliente pode ser actualizada. Se o vector do servidor for *inferior* ao do cliente, i.e., $o.v[i] \leq rv[i], \forall i$, o servidor tem uma versão mais antiga do que a versão do cliente (obtida a partir de outro servidor) — neste caso, o cliente tem a opção de obter a cópia mais antiga (o comportamento desejado é especificado como parâmetro da operação $getCoObject$). Se nenhuma das situações anteriores se verificar, o cliente e o servidor possuem versões concorrentes do *coobjecto* (cada uma reflecte operações que a outra não reflecte) — neste caso, a versão do cliente não pode ser actualizada, mas o cliente pode obter uma nova cópia.

Quando a cópia parcial de um cliente é actualizada, o servidor pode enviar para o cliente uma nova cópia do estado do *coobjecto* (incluindo todos os subobjectos modificados nos quais o cliente está interessado) ou uma sequência de operações a executar no cliente para actualizar a cópia local. Quando se usa esta segunda opção, o cliente executa a sequência de operações recebidas pela ordem indicada. Se ocorrer algum erro durante a execução das operações no cliente (por exemplo, devido ao facto de não existir uma cópia local de um subobjecto usado), o cliente obtém uma nova cópia do estado do *coobjecto*.

A actualização dum *coobjecto* deve incluir a actualização de todos os subobjectos contidos na cópia parcial do cliente – caso contrário, a cópia local ficaria inconsistente. Quando a actualização da cópia de um *coobjecto* no cliente é efectuada através da propagação do estado do *coobjecto*, o servidor verifica a necessidade de enviar uma nova cópia de um subobjecto. Esta verificação é efectuada determinando se alguma operação que ainda não está reflectida na cópia do cliente modificou o subobjecto considerado, i.e, se $\exists i : so.v[i] > rv[i]$, com $so.v$ o sumário das operações que modificaram o subobjecto e rv a versão do *coobjecto* no cliente (note-se que podem ter sido executadas novas operações no servidor que não tenham modificado o subobjecto). Neste caso, o servidor envia uma nova cópia do subobjecto.

“Aumento” da cópia parcial de um *coobjecto* Por vezes, a cópia parcial presente no cliente não é suficiente para satisfazer as necessidades de uma aplicação que está a manipular um *coobjecto*, i.e., o utilizador pretende aceder a um subobjecto do qual o cliente não possui uma cópia local. Neste caso, o cliente necessita de “aumentar” a sua cópia local, obtendo uma cópia do subobjecto que seja consistente com a cópia do *coobjecto* que possui.

Uma cópia de um subobjecto é consistente com a cópia de um *coobjecto*, se o estado do subobjecto reflectir a execução de todas as operações reflectidas no estado do *coobjecto* e apenas essas. Um servidor verifica se a cópia local de um subobjecto é consistente com a cópia de um *coobjecto* de um cliente, sendo rv a versão do *coobjecto*, verificando se: (1) a cópia local do *coobjecto* reflecte todas as operações reflectidas no cliente, i.e, $o.v[i] \geq rv[i], \forall i$, com $o.v$ a versão do *coobjecto* no servidor; (2) o subobjecto não foi alterado por operações não reflectidas na cópia do cliente, i.e., $so.v[i] \leq rv[i], \forall i$ em que $so.v$ é o sumário das operações que modificaram o subobjecto no servidor. Se a cópia no servidor for consistente com a do cliente, o servidor pode enviar o subobjecto para o cliente. Caso contrário, o cliente não pode “aumentar” a sua cópia parcial a partir desse servidor.

6.3.2 Detalhes do funcionamento dos *coobjectos*

O funcionamento geral dos *coobjectos* nos clientes foi descrito na secção 3.2.2.2. Nesta secção descrevem-se pormenores da implementação do modelo descrito no protótipo do sistema DOORS.

Como se descreveu na secção 6.1, as aplicações manipulam cópias privadas dos *coobjectos* criadas pelo núcleo do sistema. Cada cópia inclui uma referência no sistema (*handle*) que permite, quando necessário, criar cópias de outros subobjectos consistentes com a versão actual. Esta referência permite ainda: aceder aos recursos associados ao *coobjecto* e aos subobjectos nele contidos; e submeter operações para execução imediata nos servidores.

Quando uma aplicação obtém uma cópia de um *coobjecto* pode indicar um conjunto de propriedade a satisfazer pela cópia obtida. Primeiro, pode obter-se uma cópia local, sem que o cliente contacte o servidor. Neste caso, a aplicação pode optar pela cópia obtida do servidor ou por uma cópia provisória, resultado da última gravação efectuada localmente por uma aplicação. Segundo, a aplicação pode solicitar que o cliente obtenha uma versão a partir de um servidor. Terceiro, a aplicação pode solicitar que o cliente devolva uma versão que respeite uma das seguintes propriedades:

Versão sucessiva A versão do *coobjecto* obtida é posterior às versões obtidas anteriormente.

Lê modificações A versão do *coobjecto* obtida reflecte as modificações executadas anteriormente pelo utilizador.

Para garantir estas propriedades, o cliente mantém, para cada utilizador e para cada *coobjecto*, a

seguinte informação (independentemente de, no momento, manter uma cópia desse *coobjecto*).

Primeiro, um vector versão, r, que reflecte as operações observadas nas versões lidas anteriormente. Este vector é actualizado sempre que uma nova versão do *coobjecto* é lida pelo utilizador, fazendo $r[i] = max(r[i], o.v[i])$, com $o.v$ a versão do *coobjecto* lida. A primeira propriedade é garantida se a versão do *coobjecto* a devolver, $o.v$, for *superior* ao vector r, i.e., $o.v[i] \geq r[i], \forall i$.

Segundo, o cliente mantém um vector temporal, w, que reflecte as operações escritas pelo utilizador. Este vector é actualizado sempre que uma operação submetida pelo utilizador é propagada para o servidor, fazendo $w[id.srv_{view}] = id.n_seq$, com id o identificador da operação submetida (este identificador é devolvido pelo servidor). A segunda propriedade é garantida se a versão do objecto a devolver, $o.v$, for *superior* ao vector w, i.e., $o.v[i] \geq w[i], \forall i$.

Finalmente, o cliente mantém o identificador da vista em que os vectores temporais anteriores são válidos. Quando o cliente detecta uma mudança de vista, os vectores temporais anteriores são convertidos na nova vista usando a operação disponibilizada no *coobjecto* de filiação.

As propriedades referidas anteriormente apenas são garantidas para acessos executados a partir de um mesmo cliente e pelo período de tempo indicado na configuração do cliente. Se um utilizador não aceder a um *coobjecto* durante o período de tempo indicado, a informação do utilizador relativa a esse *coobjecto* é removida.

Quando uma aplicação grava as modificações efectuadas a um *coobjecto*, o cliente extrai as operações executadas e guarda-as em memória estável para posterior propagação para o servidor. Após as operações estarem guardadas em memória estável, o *coobjecto* é informado do facto e a aplicação pode continuar a manipular essa cópia do *coobjecto* (e executar novas gravações mais tarde).

Sempre que possível (i.e., desde que as primeiras sequências ainda não tenham sido propagadas), o cliente combina as sequências de operações que resultam de modificações à mesma cópia privada de um *coobjecto* numa única sequência, de forma a reflectir a causalidade existente na sua execução. Quando isso não é possível, o cliente garante que envia as operações para o mesmo servidor por ordem de gravação.

Além de extrair a sequência de operações executadas para propagação para um servidor, o cliente grava o estado actual do *coobjecto* como uma versão provisória. Desta forma, em sucessivos acessos a um *coobjecto* durante um período de desconexão, os utilizadores podem observar as modificações efectuadas anteriormente. Se as limitações de espaço disponível para as cópias locais o permitirem, o cliente mantém também a versão obtida directamente do servidor.

Quando uma aplicação grava um novo *coobjecto*, o cliente obtém uma cópia do estado actual do *coobjecto*, que é propagada para o servidor para servir como estado inicial do *coobjecto*.

6.3.3 Execução síncrona e garantias de sessão

Os clientes do sistema DOORS fornecem um serviço que permite aos *coobjectos* executar operações de leitura e escrita imediatamente num servidor. Este serviço está disponível a partir da referência de sistema (*handle*) associado a cada cópia de um *coobjecto*. O cliente é responsável pela propagação das operações para um servidor e obtenção dos respectivos resultados.

Como as réplicas de um *coobjecto* nos vários servidores são fracamente consistentes, seria possível que sucessivas operações síncronas observassem estados inconsistentes do *coobjecto* — por exemplo, uma leitura executada num servidor podia reflectir uma operação que não fosse reflectida numa leitura posterior executada noutro servidor. Para que esta situação não ocorra, é possível especificar as garantias de sessão [162] que se pretende que as sucessivas invocações síncronas respeitem. As seguintes propriedades podem ser especificadas:

Versão sucessiva A versão do *coobjecto* lida é posterior às versões lidas anteriormente.

Lê modificações A versão do *coobjecto* lida reflecte as modificações executadas anteriormente.

Servidor único Todas as operações são executadas no mesmo servidor.

Estas propriedades são garantidas de forma semelhante às propriedades fornecidas na obtenção de uma cópia de um *coobjecto* no cliente, mas mantendo os vectores temporais no âmbito da sessão remota.

Os *coobjectos* devem utilizar este serviço respeitando as intenções dos utilizadores. Em especial, é necessário ter em atenção que a propagação síncrona de uma operação de modificação equivale à sua gravação — o utilizador não pode, posteriormente, desfazer a sua execução (embora possa executar uma operação que anule os seus efeitos). Desta forma, os utilizadores devem estar conscientes das situações em que uma operação executada vai ser propagada para um servidor.

6.3.4 Serviços básicos

Um cliente do sistema DOORS possui dois subsistemas básicos: o subsistema de comunicações e o subsistema de replicação antecipada.

O subsistema de comunicações é usado para contactar os servidores e os outros clientes. Este serviço disponibiliza uma interface comum para a propagação de mensagens usando diferentes transportes (no protótipo do sistema DOORS usa-se RMI). Dois tipos de propagação estão disponíveis: síncrona e assíncrona. A propagação síncrona é usada pelo cliente quando precisa de uma resposta imediata às mensagens que está a enviar — por exemplo, durante a invocação síncrona de operações e quando é necessário obter uma cópia de um *coobjecto*/subobjecto para entregar imediatamente a uma aplicação. A semântica da propagação síncrona é *no máximo uma vez* (*at-most once*).

A propagação assíncrona é usada quando o cliente não necessita obter uma resposta imediata às mensagens enviadas — por exemplo, quando propaga as modificações executadas por um utilizador a um *coobjecto*. A semântica da propagação assíncrona é *exactamente uma vez* (*exactly-once*) (desde que o cliente não fique definitivamente desconectado do servidor). Para tal, o subsistema de comunicações mantém em memória estável a informação (conteúdo das mensagens, números de sequência enviados/recebidos/garantidos) sobre as mensagens a enviar para os os diferentes parceiros (servidores e outros clientes).

O subsistema de replicação antecipada (*pre-fetching*) é responsável por manter na *cache* do cliente as cópias dos *coobjectos* necessárias para que os utilizadores continuem o seu trabalho em situações de desconexão. Este subsistema tem acesso à informação de quais os *coobjectos*/subobjectos acedidos por cada utilizador. Como se referiu anteriormente, o problema de determinar, de forma eficiente, quais são os dados que devem ser obtidos para suportar a operação desconectada está fora do âmbito do trabalho apresentado nesta dissertação. A solução implementada no protótipo DOORS é bastante simples e baseia-se numa política de replicação *"menos recentemente utilizado"* (*least recently used*) e na actualização periódica dos *coobjectos*/subobjectos replicados no cliente.

6.3.5 Comunicação entre clientes

Por vezes, dois clientes que se encontram desconectados dos servidores conseguem comunicar directamente entre si (por exemplo, usando redes de comunicação *ad hoc* estabelecidas usando dispositivos de rede Bluetooth [19] ou *wireless ethernet* [73]). No sistema DOORS permite-se que os clientes explorem estas comunicações para actualização do estado das suas réplicas secundárias.

Cada cliente implementa uma interface que permite aos outros clientes obter cópias de *coobjectos*/subobjectos. Esta interface consiste nas operações *getCoObject* e *getAddSubObject*, definidas de forma idêntica à da interface do servidor (e descritas na secção 6.2.4). Quando um cliente, *i*, recebe uma operação *getCoobject* (ou *getAddSubObject*) de outro cliente, *j*, podem ocorrer as seguintes situações.

- O cliente *i* não tem nenhuma cópia do *coobjecto*/subobjecto pedido. Neste caso a operação aborta e o cliente *j* é informado do erro.

- O cliente *i* possui uma cópia inalterada do *coobjecto*/subobjecto pedido. Neste caso, o cliente *i* actua de forma semelhante a um servidor, enviando para *j* o estado actual do *coobjecto*/subobjecto.

- O cliente *i* possui uma cópia provisória do *coobjecto*/subobjecto pedido, reflectindo as modificações efectuadas em *i* após o *coobjecto*/subobjecto ter sido obtido a partir de um servidor. Neste caso, o cliente *i* responde com base na cópia obtida do servidor ou na versão provisória em função das opções especificadas pelo cliente *j*.

Ao permitir que os clientes propaguem entre si versões modificadas dos *coobjectos*/subobjectos, permite-se que os utilizadores observem imediatamente as operações executadas provisoriamente por outros utilizadores. Apesar de a execução dessas operações ser provisória, os utilizadores podem (devem) evitar operações que entrem em conflito com as operações executadas pelos outros utilizadores. Adicionalmente, é necessário ter em atenção que, em geral, para as operações executadas posteriormente num cliente não é registada a dependência relativa às operações provisórias executadas nos outros clientes (embora seja possível manter essa informação). Este problema foi discutido na secção 4.1.

6.3.6 Serviço de *awareness*

De forma semelhante ao que acontece no servidor, cada cliente do sistema DOORS tem associado um serviço de *awareness*. O objectivo deste serviço é permitir aos utilizadores terem conhecimento das actividades que os outros utilizadores estão a executar concorrentemente. No cliente, este serviço apenas permite a propagação de informação utilizando o sistema de disseminação de eventos Deeds [45].

6.4 Sumário

Neste capítulo descreveu-se o núcleo do sistema DOORS e a sua implementação no protótipo criado. O núcleo do sistema fornece os serviços básicos necessários para suportar o modelo do sistema DOORS. O protótipo do sistema comprova a exequibilidade do modelo proposto.

O modelo de funcionamento do sistema DOORS, incluindo as suas principais características e técnicas usadas para a gestão de dados partilhados num ambiente de computação móvel foram apresentadas nos capítulos 3 e 4. A utilização do modelo proposto para suportar aplicações cooperativas tipicamente assíncronas foi descrito no capítulo 5.

Nos próximos capítulos apresenta-se um sistema de gestão de bases de dados relacionais para ambientes de computação móvel. Apesar desse sistema partilhar com o sistema DOORS um conjunto de princípios básicos, as técnicas usadas para colocar esses princípios em prática são distintas.

Capítulo 7

Apresentação do sistema Mobisnap

Nos próximos capítulos descreve-se um sistema de gestão de bases de dados para ambientes de computação móvel: o sistema Mobisnap [126, 33, 128]. Este sistema tem por objectivo permitir que múltiplos utilizadores acedam e modifiquem uma base de dados relacional, a partir de computadores móveis, mesmo durante os períodos de desconexão. Na secção 2.2 foram apresentadas algumas das aplicações típicas que se pretendem suportar com este sistema: um sistema de suporte a uma força de vendas móvel, um sistema de reserva de lugares e uma agenda partilhada.

O sistema Mobisnap é construído como uma camada intermédia de sistema (*middleware*) e baseia-se numa arquitectura *cliente estendido/servidor* [78]. O servidor mantém a cópia principal dos dados. Os clientes mantêm cópias parciais para fornecer uma elevada disponibilidade de acesso. As aplicações modificam o estado da base de dados através da submissão de pequenos programas escritos na linguagem PL/SQL [112]. Este programas, designados transacções móveis, são executados de forma provisória no cliente e, mais tarde, de forma definitiva, no servidor.

Para permitir garantir os resultados das transacções móveis nos clientes, os utilizadores podem obter reservas sobre os dados. Este mecanismo de reservas, detalhado no capítulo 8, permite garantir que não surgirá nenhum conflito quando uma transacção é executada no servidor. No capítulo 9 apresenta-se uma avaliação do modelo do sistema Mobisnap, incluindo o sistema de reservas. No capítulo 10 descreve-se um mecanismo de reconciliação que permite optimizar o conjunto de transacções móveis que podem ser executadas. Até ao final deste capítulo descreve-se o modelo geral, a arquitectura do sistema e detalham-se as transacções móveis.

7.1 Modelo geral

O sistema Mobisnap é um sistema de gestão de bases de dados para ambientes de computação móvel baseado numa arquitectura cliente estendido/servidor. O servidor mantém a cópia principal da base

```
1  -- ENCOMENDA DE PRODUTO: nome = "BLUE THING"; quantidade = 10; preço máximo = 50.00
2  -- alternativa: nome = "RED THING"; quantidade = 5; preço máximo = 40.00
3  DECLARE
4    prd_price FLOAT;
5    prd_stock INTEGER;
6  BEGIN               --- Primeira alternativa ---
7    SELECT price,stock INTO prd_price,prd_stock FROM products WHERE name='BLUE THING';
8    IF prd_price <= 50.00 AND prd_stock >= 10 THEN
9      UPDATE products SET stock = prd_stock - 10 WHERE name = 'BLUE THING';
10     INSERT INTO orders VALUES(newid,'Clt foo','BLUE THING',10,prd_price,'to ptocess');
11     NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Encomenda aceite...');
12     COMMIT ('BLUE THING',prd_price);      -- Conclui a transacção e devolve
13   END IF;                                 -- informação sobre a encomenda efecutada
14                     --- Segunda alternativa ---
15   SELECT price,stock INTO prd_price,prd_stock FROM products WHERE name='RED THING';
16   IF prd_price <= 40.00 AND prd_stock >= 5 THEN
17     UPDATE products SET stock = prd_stock - 5 WHERE name = 'RED THING';
18     INSERT INTO orders VALUES(newid,'Clt foo','RED THING',5,prd_price,'to ptocess');
19     NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Encomenda aceite...');
20     COMMIT ('RED THING',prd_price);       -- Conclui a transacção e devolve informação sobre a
21   END IF;                                 -- encomenda efecutada
22   ROLLBACK;                               -- Caso nenhuma alterantiva seja aceitável aborta a transacção
23 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Encomenda impossível...');
24 END;
```

Figura 7.1: Transacção móvel que introduz uma nova encomenda, com dois produtos alternativos, submetida por um vendedor.

de dados que representa o estado *oficial* dos dados. Os clientes mantêm cópias parciais, fracamente consistentes, dos dados. Uma cópia parcial contém um subconjunto de tabelas. Para cada tabela, apenas é mantido um subconjunto de colunas. Finalmente, apenas um subconjunto dos registos de cada tabela está contido na cópia parcial. A possibilidade de cada cliente manter apenas os dados que necessita (e pode armazenar) é importante para permitir a utilização do sistema em dispositivos móveis com recursos limitados (por exemplo, PDAs), como se referiu na secção 2.3.5.

As aplicações são executadas nos clientes. O sistema adopta uma política optimista de acesso aos dados, na qual as aplicações podem aceder e modificar a cópia mantida no cliente. A esta cópia, na qual se executam todas as transacções submetidas no cliente, chama-se versão *provisória* dos dados. Quando os recursos disponíveis no cliente o permitem, o cliente mantém adicionalmente uma versão *estável* dos dados, a qual mantém a cópia obtida a partir do servidor, modificada com as transacções executadas localmente cujo resultado pode ser garantido pelo sistema de reservas (detalhado no próximo capítulo). As aplicações podem aceder a ambas as versões. Durante os períodos de *boa* conectividade, as aplicações podem aceder directamente ao servidor.

Transacções móveis As aplicações modificam o estado da base de dados submetendo pequenos programas escritos num subconjunto estendido da linguagem PL/SQL [112], que se denominam transacções móveis (ou simplesmente transacções, nas situações em que a sua utilização não possa levar a interpretações incorrectas). Na figura 7.1 apresenta-se um exemplo de uma transacções móvel para submeter uma

encomenda de um produto, propondo duas possíveis alternativas.

A execução de uma transacção móvel corresponde sempre à execução do seu programa numa cópia da base de dados. As transacções móveis são executadas imediatamente no cliente produzindo um resultado provisório (e modificando o estado da versão provisória dos dados). As transacções móveis são posteriormente propagadas para o servidor onde são reexecutadas. O resultado definitivo de uma transacção móvel (e o modo como afecta a cópia principal dos dados) resulta da sua execução no servidor.

O modelo de execução das transacções móveis, com o resultado definitivo a ser determinado pela execução do código da transacção no servidor, permite a definição de regras de detecção e resolução de conflitos específicas para cada operação. Estas regras devem garantir o respeito pelas intenções dos utilizadores explorando a informação semântica associada aos tipos de dados e operações definidas. A transacção da figura 7.1 exemplifica algumas das estratégias que podem ser utilizadas.

Primeiro, as pré-condições para a execução da operação são especificadas de forma exacta (condições das linhas 8 e 16). Assim, a transacção não é abortada se os valores observados no cliente e no servidor forem diferentes, mas apenas se as condições expressas forem violadas. Esta aproximação permite reduzir o número de falsos conflitos detectados, quando comparada com as técnicas que forçam a igualdade dos valores lidos no cliente e no servidor (usuais em sistemas de bases de dados). Segundo, a transacção especifica várias alternativas. Esta aproximação representa uma forma simples de resolver eventuais conflitos desde que uma das alternativas seja possível.

Uma aplicação pode submeter transacções móveis que modifiquem dados não replicados no cliente. Neste caso, apesar de não ser possível obter um resultado provisório, a transacção pode ser propagada para o servidor onde o resultado final é obtido executando o programa da transacção na base de dados do servidor. Esta aproximação permite aos utilizadores continuarem o seu trabalho mesmo quando não é possível obter uma cópia dos dados (como se referiu na secção 2.3.6).

Quando o resultado definitivo de uma transacção móvel é obtido no servidor, os utilizadores podem já não estar a utilizar o sistema. Assim, como se referiu na secção 2.3.4, é importante integrar um mecanismo que possibilite informar os utilizadores sobre os resultados definitivos das suas operações. No sistema Mobisnap, definiu-se uma função que pode ser usada nas transacções móveis para enviar mensagens aos utilizadores através de diferentes transportes. Esta função é processada de forma especial pelo sistema, apenas produzindo efeito na execução definitiva da transacção móvel. Adicionalmente, o sistema encarrega-se de garantir a propagação destas mensagens através do transporte indicado.

Reservas A possibilidade de garantir o resultado de uma transacção móvel de forma independente tem várias vantagens, como se discutiu na secção 2.3.7. Primeiro, permite suportar de forma mais adequada a operação em situações de desconexão. Segundo, mesmo quando é possível contactar o servidor, permite reduzir o tempo de execução das operações e gerir de forma mais eficiente as comunicações entre os

clientes e o servidor, além de permitir gerir a execução dessas operações no servidor de acordo com a carga do sistema. O Mobisnap integra um sistema de reservas, detalhado no capítulo 8, que permite aos clientes obter garantias sobre os valores da base de dados. Caso o cliente disponha de reservas suficiente, o cliente pode garantir que não surgirá nenhum conflito quando a transacção móvel é executada no servidor, permitindo garantir o resultado da transacção de forma independente.

Alguns dos tipos de reservas definidos no sistema Mobisnap já foram utilizados noutros sistemas [111]. No entanto, a sua integração num sistema unificado como o apresentado é fundamental para a sua utilidade. Considere-se o exemplo da figura 7.1. Usando técnicas de divisão (*escrow*) [1] é possível garantir a existência de unidades suficientes de um produto para satisfazer o pedido. No entanto, para que o sistema possa garantir o resultado da transacção é igualmente necessário que possa garantir o valor do preço. Assim, seria necessário usar outro tipo de garantia. Neste caso, poder-se-ia usar um trinco (*lock*) apesar de essa técnica ser desnecessariamente restritiva.

Uma propriedade importante do sistema de reservas é a verificação transparente das reservas que podem ser utilizadas para garantir uma transacção móvel. Esta verificação é efectuada a partir do código da transacção, sem que seja necessário especificar quaisquer instruções especiais. Assim, qualquer transacção pode usar todas as reservas existentes e não apenas aquelas que a transacção está preparada para utilizar.

O sistema faz respeitar as garantias relativas às reservas concedidas, não ficando as propriedades associadas dependentes de convenções a cumprir pelas transacções. Assim, impede-se qualquer transacção executada no sistema Mobisnap ou directamente na base de dados central de modificar a base de dados de forma a violar as garantias relativas a qualquer reserva. Como as reservas são temporárias (*leased*) [58], garante-se que as restrições impostas às outras transacções não duram indefinidamente, mesmo que o cliente móvel, a quem a reserva foi concedida, seja destruído ou fique permanentemente desconectado. No entanto, para que a garantia dada quanto ao resultado de uma transacção seja válida é necessário que a transacção seja propagada para o servidor antes das reservas usadas expirarem.

Reconciliação de múltiplas transacções O sistema Mobisnap inclui ainda um mecanismo de reconciliação que permite criar um plano de execução quase óptimo como resultado da reconciliação de um conjunto de transacções móveis. A solução proposta é uma extensão do modelo do sistema IceCube [85], e utiliza informação semântica extraída automaticamente das transacções móveis. Este mecanismo de reconciliação é genérico e pode ser usado independentemente em qualquer sistema de bases de dados para reconciliar conjuntos de transacções executadas concorrentemente. Assim, este mecanismo é apre-

[1] As técnicas de divisão (*escrow*) permitem dividir um recurso fungível por várias réplicas — por exemplo, a existência (*stock*) de um produto pode ser dividido por vários vendedores. Cada réplica pode decidir sobre a sua parte de forma independente.

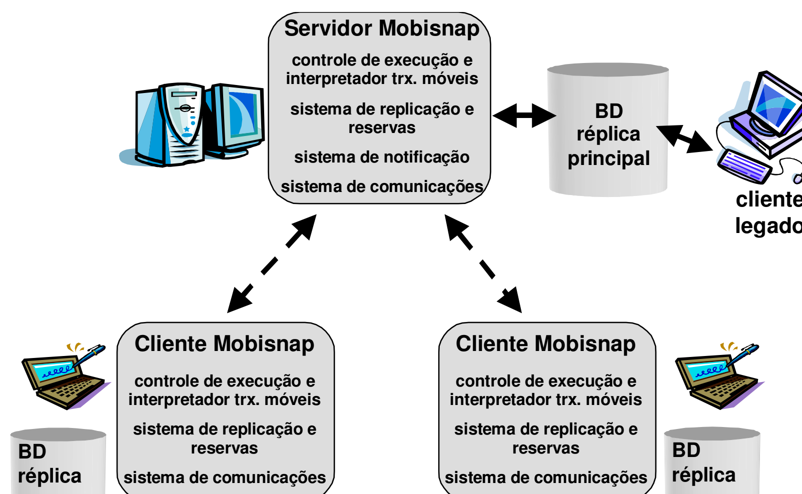

Figura 7.2: Arquitectura do sistema Mobisnap.

sentado de forma independente no capítulo 10.

Aproximação evolutiva O modelo do sistema Mobisnap é evolutivo, pois permite que aplicações que não usem o sistema Mobisnap continuem a aceder directamente à base de dados central. O sistema garante o funcionamento correcto dos mecanismos propostos no sistema Mobisnap (transacções móveis, reservas e reconciliação de múltiplas transacções) independentemente de qualquer modificação concorrente executada na base de dados central.

Adicionalmente, o sistema baseia-se na utilização de linguagens normalizadas (SQL e PL/SQL). Esta característica, ao permitir que os programadores utilizem as funcionalidades do sistema através de ferramentas que lhe são familiares permite facilitar o desenvolvimento de novas aplicações que utilizem as novas funcionalidades (como se discutiu na secção 2.3.8).

7.2 Arquitectura

A arquitectura do sistema Mobisnap é baseada no modelo cliente estendido/servidor [78], como se representa na figura 7.2. O servidor deve executar numa máquina com boa conectividade (em geral, um computador fixo). O servidor mantém o estado "oficial" dos dados, ou seja, a réplica principal (e completa) da base de dados. Os clientes podem ser computadores fixos ou móveis capazes de comunicar com o servidor. Os clientes mantêm cópias parciais dos dados.

O Mobisnap é desenhado (e implementado) como uma camada de sistema intermédia (*middleware*). Assim, o Mobisnap utiliza um sistema de gestão de bases de dados para gerir os dados no cliente e no servidor. As funcionalidades do sistema Mobisnap são implementadas pelos componentes do sistema recorrendo aos serviços básicos fornecidos pelo sistema de gestão de bases de dados.

O sistema Mobisnap apresenta uma aproximação evolutiva, permitindo aos clientes legados (*legacy*) continuarem a aceder directamente à base de dados central. Como é óbvio, os clientes que acedam directamente à base de dados central não podem utilizar as funcionalidades do sistema Mobisnap.

7.2.1 Servidor

O servidor Mobisnap é composto por um conjunto de subsistemas que interagem para executar as funcionalidades do sistema Mobisnap.

O sistema de gestão de bases de dados mantém o valor da base de dados manipulada pelas aplicações. Adicionalmente, armazena, de forma estável, toda a informação interna dos vários componentes do servidor.

Qualquer sistema de gestão de bases de dados que implemente as seguintes funcionalidades pode ser usado num servidor Mobisnap. Primeiro, deve fornecer uma interface que utilize a linguagem SQL'92 [76]. Segundo, deve implementar um mecanismo de gatilhos (*triggers*), usado pelo subsistema de reservas. Terceiro, deve implementar um mecanismo de transacções encaixadas (*nested transactions*) ou gravação e reposição total do estado (*checkpoint*), usado pelo mecanismo de reconciliação. No protótipo do sistema Mobisnap usou-se o sistema de gestão de bases de dados Oracle 8 [113].

O subsistema de reservas efectua a gestão completa das reservas, como se detalha no próximo capítulo. No funcionamento deste subsistema podem ser identificadas três actividades complementares. Primeiro, o processamento dos pedidos de obtenção de reservas. Segundo, o processamento das utilizações das reservas. Terceiro, a gestão da validade das reservas no tempo e dos recursos a elas associados.

O subsistema de replicação é responsável por gerir a interacção com os clientes no âmbito da gestão das réplicas dos clientes. No protótipo do sistema, o servidor limita-se a processar os pedidos de obtenção e actualização das cópias parciais secundárias submetidos pelos clientes. Estes pedidos são especificados através de uma instrução de interrogação baseada na instrução SQL *select*.

Numa versão mais completa da gestão da replicação secundária, este subsistema pode ser responsável pelas seguintes actividades complementares. Primeiro, a inferência de regras de acesso aos dados a serem utilizadas pelo mecanismo de replicação antecipada (várias soluções foram propostas na literatura [87, 91]). Segundo, o controlo da divergência entre a cópia principal e as cópias secundárias armazenadas nos clientes (várias soluções foram propostas na literatura [6, 146, 173]). Como se referiu anteriormente, estes problemas, relativos à gestão da replicação secundária, estão fora do âmbito desta dissertação.

O subsistema de controlo de execução é responsável por controlar a execução das transacções móveis recebidas do cliente. Este processo, descrito na secção 7.3, é efectuado utilizando um interpretador de PL/SQL em que os comandos SQL são executados directamente sobre a base de dados[2].

[2]Esta aproximação permite que a base de dados utilizada não necessite de executar programas PL/SQL.

O subsistema de notificações encarrega-se de entregar as mensagens produzidas durante a execução definitiva das transacções móveis aos serviços que efectuam a sua entrega final aos utilizadores. Este subsistema executa as mesmas funções que o serviço de notificações do sistema DOORS, descrito na secção 6.2.7.

O subsistema de comunicações controla a troca de mensagens entre os clientes e os servidores usando um modelo de propagação de mensagens síncrono ou assíncrono. Quando se usa um modelo síncrono, o subsistema executa uma semântica *no máximo uma vez* (*at most once*). Quando se usa um modelo assíncrono, o subsistema executa uma semântica *exactamente uma vez* (*exactly-once*).

No protótipo do sistema Mobisnap, são usados dois transportes na propagação de mensagens: TCP e SMTP. O sistema de propagação implementa um modelo de segurança utilizando técnicas de criptografia. Este modelo é baseado na definição de uma sessão segura de duração limitada que tem associada uma chave de criptografia simétrica. A sessão é estabelecida recorrendo a um modelo de criptografia assimétrica. O subsistema de comunicação usado no sistema Mobisnap é detalhado em [32, 33].

7.2.2 Cliente

O cliente Mobisnap é composto por um conjunto de subsistemas que interagem para executar as funcionalidades do sistema Mobisnap.

O sistema de gestão de bases de dados mantém as cópias parciais da base de dados manipulada pelas aplicações. Adicionalmente, armazena, de forma estável, a informação interna dos vários componentes do cliente. O cliente Mobisnap pode utilizar diferentes bases de dados — no protótipo do sistema Mobisnap usa-se o Hypersonic SQL [71], uma sistema de gestão de bases de dados de pequena dimensão (menor que 200 Kb) totalmente implementado em Java.

O subsistema de replicação controla a gestão das réplicas parciais mantidas pelo cliente. Como se referiu anteriormente, o cliente mantém duas cópias da base de dados, uma versão estável e uma versão provisória. Estas cópias são actualizadas a partir dos servidores e durante a execução local das transacções móveis. As aplicações podem executar operações de interrogação sobre cada uma das versões. No protótipo actual, é impossível saber se os resultados obtidos são incompletos devido à replicação parcial. No entanto, através da introdução de um mecanismo de replicação secundária semântica [35, 142] seria possível indicar a completude do resultado.

O subsistema de reservas efectua a gestão das reservas no cliente. Duas actividades são executadas. Primeiro, para cada utilizador, mantém as reservas que podem ser utilizadas localmente. Segundo, controla a utilização das reservas nas transacções móveis executadas localmente.

Os subsistemas anteriores são, igualmente, responsáveis por controlar a obtenção e actualização das réplicas locais e das reservas obtidas. No protótipo actual, as aplicações devem especificar quais os dados

e reservas a obter.

O subsistema de controlo de execução é responsável por controlar a execução das operações efectuadas pelas aplicações. As operações relativas à gestão da replicação e das reservas são propagadas para os subsistema relevantes. As operações de interrogação são executadas directamente sobre as cópias locais da base de dados. As transacções móveis são processadas como se descreve na secção 8.3, utilizando um interpretador de PL/SQL e a informação sobre as reservas existentes. Após serem executadas localmente, as transacções móveis são armazenadas de forma estável até serem propagadas para o servidor. O cliente usa o mecanismo de compressão apresentado na secção 10.4.2 para comprimir a sequência de operações a propagar.

O subsistema de comunicações foi explicado anteriormente. No entanto, nos clientes, o subsistema de comunicações é igualmente utilizado para as comunicações entre os clientes (efectuadas sempre de forma síncrona). Um cliente pode obter, a partir de outro cliente, uma cópia mais actualizada da base de dados. Adicionalmente, é possível, a um cliente, delegar noutro um conjunto de reservas. Para que o servidor possa verificar a validade da utilização das reservas, a delegação de uma reserva é sempre acompanhada de um endosso assinado pelo cliente que possuía a reserva — esta informação (cadeia de endossos) é posteriormente propagada para o servidor quando a reserva é utilizada.

7.2.3 Protótipo

Para validar a exequibilidade do modelo proposto, foi implementado um protótipo do sistema Mobsinap na linguagem Java. O protótipo implementa as funcionalidades propostas no sistema através de uma camada de sistema intermédia (*middleware*), organizada como descrito nesta secção.

Para permitir a utilização de diferentes bases de dados no cliente e no servidor usou-se a aproximação descrita na secção 6.2.6. O sistema de comunicações seguras foi criado recorrendo à API de segurança do Java. Inicialmente, recorreu-se à implementação desta API efectuada pela Cryptix [31]. Os interpretadores de PL/SQL foram criados usando o programa de criação de analisadores gramaticais JavaCC [169]. A implementação do mecanismo de execução de múltiplas transacções usa parcialmente o código do sistema IceCube [134].

O servidor do sistema Mobisnap foi testado em computadores PCs com o sistema operativo Windows 2000/XP e Linux. Os clientes foram testados em computadores PCs com várias versões do sistema operativo Windows e Linux e em computadores de bolso Compaq iPaq com o sistema operativo SavajeOS [150]. No entanto, como são completamente implementados em Java (incluindo o sistema de gestão de bases de dados usado), espera-se que corram em qualquer computador que possua uma implementação do ambiente “Java 2, standard edition”.

7.3 Transacções móveis

As aplicações modificam a base de dados através da submissão de transacções móveis. Uma transacção móvel é um pequeno programa especificado num subconjunto da linguagem PL/SQL[3] [112]. No apêndice B.1 detalha-se o subconjunto da linguagem implementado no protótipo do sistema Mobisnap. Relativamente à linguagem PL/SQL foram introduzidas as seguintes modificações: (1) as instruções *commit* e *rollback* terminam a execução de uma transacção móvel, i.e., actuam como uma instrução de terminação; (2) quando o resultado de uma instrução *select into* inclui mais do que um registo, é atribuído à variável o valor de um dos registos (em vez de lançar uma excepção); (3) foi introduzida a função *newid* cujo resultado é o mesmo identificador único quando executada no cliente e no servidor.

A execução de uma transacção móvel (no cliente ou no servidor) corresponde sempre à execução do seu programa usando um interpretador especial de PL/SQL. A execução das várias instruções de acesso à base de dados de uma transacção móvel são efectuadas no âmbito de uma transacção definida no sistema de gestão de bases de dados usado.

Em geral, uma transacção móvel corresponde a uma operação na aplicação que a submete — por exemplo, a reserva de um lugar num comboio ou a introdução de uma encomenda são exemplos típicos de operações que podem ser submetidas como transacções móveis. Para cada operação, a aplicação deve ter definido um esqueleto de transacção móvel. Este esqueleto é instanciado com os valores especificados pelos utilizadores, criando uma transacção móvel que será submetida para execução pelo sistema Mobisnap. De seguida descreve-se o processamento de uma transacção móvel.

7.3.1 Processamento

Quando uma aplicação submete uma transacção móvel, o seu processamento consiste, em geral, nos seguintes passos.

No cliente, a transacção é executada de forma provisória e o seu resultado devolvido à aplicação. Se o cliente possui reservas suficiente para garantir o resultado da transacção, este resultado pode ser considerado definitivo. Em consequência, ambas as versões da cópia local da base de dados são actualizadas. Caso contrário, o resultado é provisório. Neste caso, apenas a versão provisória é actualizada. O processamento de uma transacção móvel no cliente e a sua interacção com o sistema de reservas é detalhado na secção 8.3.

Após ter sido executada localmente, uma transacção móvel é armazenada de forma estável no cliente. Quando um cliente se conecta com o servidor, o cliente propaga as transacções móveis armazenadas para

[3] O PL/SQL é um linguagem imperativa definida pela Oracle que estende a linguagem SQL com um conjunto de estruturas de características comuns às linguagens de alto nível, como por exemplo a definição de tipos, constantes e variáveis; instruções de atribuição, de selecção (*if*) e de repetição (*loop*,*for*,*while*).

o servidor. Em ambientes fracamente conectados, a propagação das transacções pode ser efectuada de forma assíncrona e incremental com uma semântica *exactamente uma vez* (*exactly-once*).

Quando um servidor recebe uma transacção móvel de um cliente, o servidor executa o programa da transacção móvel na réplica principal dos dados para obter o seu resultado definitivo. Se a transacção foi garantida no cliente, o modelo das reservas garante que não surgirá nenhum conflito inesperado durante a execução da transacção no servidor, como se detalha na secção 8.4.

Por vezes, a existência de reservas concedidas a outros utilizadores pode levar a que uma transacção móvel falhe desnecessariamente. Por exemplo, caso um cliente tenha reservado um lugar num comboio (para garantir transacções de forma independente), o servidor garante que, pelo menos, um lugar permanece disponível. Assim, qualquer transacção de outro cliente que tente reservar o último lugar disponível é abortada. Caso o cliente não utilize o lugar reservado, pode ter existido uma transacção móvel que foi abortada sem necessidade.

Para lidar com estas situações, o Mobisnap inclui um mecanismo de reexecução. As transacções móveis abortadas no servidor podem ser reexecutadas após as reservas relevantes expirarem ou serem canceladas. Neste sentido, uma reserva pode ser vista como uma opção para modificar a base de dados em primeiro lugar. As aplicações devem indicar, quando submetem uma transacção móvel, se este mecanismo deve ser usado e qual o prazo para a sua reexecução final.

Existem duas situações em que é necessário executar um conjunto de transacções no servidor. Primeiro, quando o cliente propaga um conjunto de transacções para o servidor. Segundo, quando um conjunto de transacções aguarda a sua reexecução no servidor. Nestas situações, o servidor utiliza o mecanismo de reconciliação descrito no capítulo 10 para optimizar o conjunto de transacções não garantidas no cliente que pode ser executado com sucesso (as transacções garantidas no cliente são executadas imediatamente).

O sistema mantém informação sobre as transacções móveis que foram submetidas recentemente para o servidor numa tabela do sistema (periodicamente, os registos mais antigos são removidos). Esta informação consiste no resultado definitivo da transacção, no caso de este já ter sido determinado, ou na indicação que aguarda reexecução (nas situações explicadas anteriormente). Esta informação permite aos clientes obter o resultado das transacções submetidas em interacções anteriores.

7.3.2 Processamento alternativo

O modo de processamento típico de uma transacção móvel, descrito anteriormente, pode ser alterado através das seguintes opções especificadas aquando da submissão da transacção móvel:

Propaga sempre Uma transacção móvel é sempre propagada para o servidor, independentemente do seu resultado no cliente (por omissão as transacções que abortaram – i.e., tiveram o resultado *tentative*

abort – não são propagadas).

Garantido Se não for possível garantir o resultado de uma transacção móvel no cliente, a transacção é imediatamente propagada para o servidor para execução síncrona (não sendo executada provisoriamente no cliente). Se não for possível contactar o servidor, o processamento prossegue como habitualmente.

Servidor Se for possível contactar o servidor, a transacção é imediatamente propagada para o servidor para execução síncrona. Caso contrário, o processamento prossegue como habitualmente.

Imediato Esta opção altera o comportamento das anteriores. Se não for possível satisfazer imediatamente o processamento indicado, a transacção móvel é abortada e o seu processamento é concluído.

Reexecuta Explicita que se pretende usar o mecanismo de reexecução no servidor, caso a execução da transacção falhe. Um parâmetro adicional define o prazo limite para a execução definitiva da transacção (i.e., o último momento em que é possível reexecutar a transacção móvel).

Melhor caminho Explicita que, quando apenas é possível garantir um resultado não óptimo no cliente, se pretende que a execução da transacção no servidor siga o melhor caminho possível, mesmo que diferente do caminho executado no cliente (ver detalhes no próximo capítulo). No exemplo da figura 7.1, pode aceitar-se, no servidor, a alternativa relativa ao produto *BLUE THING* mesmo que no cliente apenas tenha sido possível garantir a alternativa relativa ao produto *RED THING*.

Caminho garantido Explicita que, se não for possível garantir o caminho de execução no cliente, não se deve tentar obter garantias sobre caminhos alternativos.

O processamento de uma transacção móvel no cliente e no servidor respeita as opções especificadas pela aplicação aquando da sua submissão no cliente.

Neste capítulo apresentou-se o modelo básico do sistema Mobisnap, descrevendo-se a sua arquitectura e o modo de processamento das transacções móveis. No próximo capítulo detalha-se o sistema de reservas proposto para prevenir a ocorrência de conflitos.

Capítulo 8

Reservas

Neste capítulo descreve-se o sistema de reservas do sistema Mobisnap. O objectivo das reservas é permitir ao sistema garantir o resultado de uma transacção móvel de forma independente. Para tal, é necessário garantir que não surgirá nenhum conflito quando o programa da transacção é executado no servidor. Assim, é possível garantir que uma transacção móvel tem o mesmo resultado e produz as mesmas modificações no cliente e no servidor (ou modificações equivalentes no contexto da aplicação). O sistema de reservas do sistema Mobisnap é desenhado com o objectivo de alcançar estes objectivos enquanto as aplicações continuam a usar as instruções usuais no PL/SQL. No cliente, o sistema verifica automaticamente a possibilidade de garantir o resultado de uma transacção a partir do programa da transacção móvel.

8.1 Tipos de reservas

Uma reserva pode fornecer dois tipos de garantias para a execução de uma transacção móvel no servidor. Primeiro, uma garantia sobre o valor da base de dados. Assim, é possível garantir que o estado da base de dados respeita as pré-condições para a execução da transacção móvel. Segundo, uma garantia sobre a exequibilidade de uma operação de modificação. Assim, garante-se que é possível executar as operações da transacção móvel. De seguida, apresentam-se as reservas definidas no sistema Mobisnap[1]. Na secção 8.5 apresentam-se exemplos que mostram a necessidade de combinar diferentes tipos de reservas para garantir uma transacção móvel.

8.1.1 Reservas *value-change* e *slot*

Uma reserva *value-change* fornece o direito exclusivo de alterar um subconjunto de colunas num registo já existente. Por exemplo, um utilizador pode reservar o direito de modificar a descrição do ocupante

[1] Para nomear as reservas, usam-se os termos ingleses propostos em [126, 128].

de um lugar num comboio. Esta reserva é semelhante a um trinco de escrita (*write lock*) de pequena granularidade e influencia as instruções SQL *select* e *update*.

Uma reserva *slot* fornece o direito exclusivo de inserir, remover ou modificar registos com valores pré-determinados para algumas colunas. Por exemplo, um utilizador pode reservar o direito de alterar uma agenda durante um dado período de tempo. Esta reserva é semelhante a um trinco de predicado (*predicate lock*) [60], incluindo registos existentes e não existentes. Esta reserva influencia as instruções SQL *select*, *insert*, *delete* e *update*.

Este tipo de reservas permite, não só, garantir a possibilidade de executar alterações, mas também, garantir que o estado da base de dados no servidor será idêntico ao observado no cliente. Assim, pode ser visto como um mecanismo que permite mover a cópia principal de um subconjunto dos dados para um computador móvel.

O uso deste tipo de reservas deve ser efectuado cuidadosamente, porque elas impedem outros utilizadores de modificar os dados reservados. No entanto, se a granularidade das reservas obtidas pelos utilizadores for adequada, estas reservas fornecem um mecanismo eficaz que contribui para o funcionamento correcto do sistema garantindo o resultado das transacções independentemente.

8.1.2 Reserva *value-lock*

Uma reserva *value-lock* garante que o valor de um subconjunto de colunas de um registo não será modificado concorrentemente. Por exemplo, pode garantir-se que o identificador de um produto não é modificado. Esta reserva é semelhante a um trinco de leitura (*read lock*) de pequena granularidade e influencia a instrução SQL *select*.

Este tipo de reservas apenas permite garantir que o estado da base de dados no servidor será idêntico ao observado no cliente.

8.1.3 Reservas *value-use*

Uma reserva *value-use* permite a uma transacção móvel usar um dado valor para um dado elemento de dados, independentemente do seu valor actual. A necessidade deste tipo de garantias é comum na vida real — por exemplo, um vendedor ambulante pode garantir o preço de uma encomenda (igual ao preço que ele conhece) independentemente de qualquer actualização que tenha sido efectuada concorrentemente.

Uma transacção que use esta reserva observa o valor reservado no cliente e no servidor (as instruções de leitura devolvem o valor reservado em vez do valor actual da base de dados).

8.1.4 Reservas *escrow*

Uma reserva *escrow* fornece o direito exclusivo de usar uma parte de um recurso divisível representado por um elemento de dados numérico. Por exemplo, a existência de um produto (*stock*) pode ser dividido entre vários vendedores móveis. Esta reserva é baseada no modelo de divisão (*escrow*) [111].

O modelo de divisão é aplicado a elementos de dados numéricos que representam recursos idênticos[2] e divisíveis — por exemplo, o número de CDs existentes num armazém, x. Para estes dados é comum que as seguintes propriedades sejam válidas. Primeiro, eles são actualizados através da adição e subtracção de constantes — por exemplo, quando k CDs são vendidos, faz-se $x \leftarrow x - k$. Segundo, é necessário manter restrições quanto aos valores que esses dados podem tomar — por exemplo, a existência mínima de CDs é min, ou seja $x \geq min$.

Para estes elementos de dados (chamados elementos de dados fungíveis) é possível dividir os recursos disponíveis por várias réplicas — por exemplo, pode dividir-se x em n partes x_i, $1 \leq i \leq n$, tal que $x = \sum x_i$. Cada réplica tem associada uma restrição local que garante a validade da restrição global, $x_i \geq min_i$: $min = \sum min_i$. Cada réplica pode garantir independentemente o resultado das transacções que respeitam as restrições locais, i.e., é possível garantir o resultado de transacções que subtraiam até $x_i - min_i$ (valor agregado relativo a todas as transacções garantidas na réplica i) a x_i.

Exemplo: Seja dez o valor actual da existência de CDs ($x = 10$) e dois o seu valor mínimo ($x \geq 2$). Usando o modelo de divisão (*escrow*) é possível dividir a existência actual por duas réplicas — por exemplo, a primeira réplica pode ficar com seis CDs ($x_1 = 6$) e a segunda réplica com quatro CDs ($x_2 = 4$). A restrição global deve também ser dividida entre as duas réplicas — por exemplo, a existência mínima em cada réplica pode ser igual a um ($x_1 \geq 1 \wedge x_2 \geq 1$). A primeira (respectivamente segunda) réplica pode garantir transacções que subtraiam até cinco (respectivamente três) CDs à existência de CDs ($x_1 - min_1 = 6 - 1 = 5$ e $x_2 - min_2 = 4 - 1 = 3$). ■

Discute-se agora a implementação do modelo de divisão no sistema Mobisnap. Para reconhecer os aspectos envolvidos, considere-se a transacção móvel apresentada na figura 8.1, na qual se insere uma nova encomenda submetida por um vendedor móvel. Nesta transacção é possível usar as técnicas de divisão para garantir a disponibilidade da existência do produto. A transacção executa os seguintes passos para actualizar a existência do produto: (1) o valor actual é lido e a validade da operação verificada (linhas 3-4); (2) o valor do elemento de dados que representa a existência é actualizado para reflectir a operação executada (linha 5). Estes passos representam um padrão na actualização de elementos de dados divisíveis.

O primeiro aspecto a considerar consiste na garantia da validade da operação (passo 1). Para tal, é

[2] Na secção 8.5 mostra-se como é possível solucionar problemas com recursos não idênticos usando o sistema de reservas — por exemplo, a marcação de lugares num comboio.

```
1  -- ENCOMENDA DE PRODUTO: nome = "BLUE THING"; quantidade = 10; preço máximo = 50.00
2  BEGIN
3    SELECT price,stock INTO prd_price,prd_stock FROM products WHERE name='BLUE THING'
4    IF prd_price <= 50.00 AND prd_stock >= 10 THEN           -- Verifica a validade da operação
5      UPDATE products SET stock = prd_stock - 10 WHERE name = 'BLUE THING';
6      INSERT INTO orders VALUES (newid,'Clt foo','BLUE THING',10,prd_price,'to process');
7      NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Encomenda aceite...');
8      COMMIT prd_price;                     -- Conclui a transacção e devolve informação sobre
9    END IF;                                  -- a encomenda efectuada
10   ROLLBACK;
11 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Encomenda impossível...');
12 END;
```

Figura 8.1: Transacção móvel que especifica uma nova encomenda submetida por um vendedor (o bloco de declaração de variáveis é omitido).

comum usar uma instrução *if* que verifique se a actualização viola a restrição local. Para usar a mesma condição sem nenhuma função especial, é necessário que a mesma restrição seja imposta no cliente e no servidor. Assim, o modelo de divisão, como definido anteriormente [90], não pode ser implementado imediatamente porque usa diferentes restrições em cada uma das réplicas.

No sistema Mobsinap, introduzimos a seguinte alteração ao modelo de divisão apresentado anteriormente: $x : x \geq min$ é dividido em n partes $x_i : x_i \geq min \wedge x - min = \sum(x_i - min)$. Como anteriormente, cada réplica pode garantir independentemente o resultado das transacções que respeitem as restrições locais. No entanto, usa-se a mesma restrição em todas as réplicas. Esta propriedade permite verificar a validade de uma operação usando a mesma restrição global (ou, mesmo, usar condições mais restritivas).

Exemplo: Para obter as mesmas garantias que no exemplo anterior, usam-se os seguintes valores: $x_1 = 7, x_1 \geq 2$ e $x_2 = 5, x_2 \geq 2$. A primeira (respectivamente segunda) réplica pode garantir transacções que subtraiam até cinco (respectivamente três) CDs da sua existência ($x_1 - min = 7 - 2 = 5$ e $x_2 - min = 5 - 2 = 3$).

Seja x_1 a réplica no servidor e x_2 a réplica num cliente móvel. $x_2 = 5, x_2 \geq 2$ significa que o cliente móvel reservou o direito de subtrair três ao valor de x (i.e., pode garantir encomendas para três CDs). Neste caso, diz-se que o cliente móvel obteve uma reserva *escrow* para três unidades de x. No servidor, o valor de x é actualizado ($x_1 = 10 - 3 = 7$) de forma a reflectir as garantias obtidas pelo cliente. Assim, nenhuma transacção de outros clientes pode usar os elementos reservados.

Em alternativa, o valor de x_2 pode ser visto como o menor valor de x quando a transacção móvel é executada no servidor (i.e., $x \geq x_2$). O valor de x_2 apenas pode ser usado para garantir condições que usem um operador relacional semelhante ao da restrição. Por exemplo, a condição $x \leq 10$ não pode ser garantida pela reserva *escrow* $x_2 = 5, x_2 \geq 2$ porque esta reserva não garante o valor máximo de x. O valor de x_1 pode ser visto como a quantidade do recurso divisível não reservado por nenhum cliente móvel (e portanto disponível para ser utilizado pelos outros clientes, incluindo os clientes legados). ■

O segundo aspecto a considerar consiste na actualização dos elementos de dados fungíveis. Em geral,

uma transacção móvel actualiza o seu valor através de uma instrução *update*. No sistema Mobisnap, o sistema infere automaticamente a quantidade usada do recurso reservado a partir da instrução de *update*. Como esperado, os recursos usados são consumidos da reserva *escrow*. As próximas transacções apenas podem ser garantidas usando o remanescente dos recursos reservados.

Como se descreveu, esta aproximação é completamente transparente para os programadores. Os programadores escrevem as transacções móveis sem nenhuma referência às reservas. Se o cliente possui alguma reserva, o sistema usa-a automaticamente para tentar garantir o resultado das transacções. O sistema mantém igualmente registo das reservas usadas de forma automática.

O protótipo do sistema Mobisnap tem uma limitação: apenas é possível impor uma única restrição sobre cada elemento de dados fungível (ou $x \geq min$ ou $x \leq max$) — antecipa-se que este seja o cenário mais comum. Para impor uma restrição $min \leq x \leq max$ seria necessário usar diferentes valores para x_i dependendo do tipo de operação executada.

8.1.5 Reservas *shared value-change* e *shared slot*

Uma reserva *shared value-change* garante o direito partilhado de modificar o valor de um registo (ou subconjunto de colunas de um registo). Por exemplo, esta reserva pode ser usada para garantir uma operação de incremento sobre um contador partilhado. Esta reserva permite obter a garantia que uma instrução *update* não será bloqueada por uma reserva *value-change*, *slot* ou *value-lock*.

Uma reserva *shared slot* garante o direito partilhado de inserir, remover e modificar registos com valores pré-definidos. Por exemplo, esta reserva pode ser usada para garantir operações sobre tabelas a que apenas é possível adicionar registos (*append-only tables*). Esta reserva garante que uma instrução de *insert*, *delete* ou *update* não será bloqueada por uma reserva *value-change*, *slot* ou *value-lock*.

Estas reservas não fornecem qualquer garantia sobre o estado futuro da base de dados. No entanto, ao garantirem que nenhum cliente obtém uma reserva exclusiva que impossibilite a execução de uma operação, elas são importantes para garantir de forma completa uma transacção, como se exemplifica na secção 8.5.

8.2 Concessão e garantia de respeito pelas reservas

Nesta secção descreve-se o modo como as reservas são concedidas e como se garante o respeito pelas reservas concedidas, i.e., como se garante que nenhuma transacção viola as garantias fornecidas por uma reserva (incluindo as transacções executadas directamente na base de dados central pelos clientes legados). Nas próximas secções descreve-se o processamento das transacções móveis no cliente e no servidor quando se usam reservas.

	vc	vl	s	vu	e	shvc	shs
value-change (vc)	não	não	não	sim	não	não	não
value-lock (vl)	não	sim	não	sim	não	não	não
slot (s)	não	não	não	sim	não	não	não
value-use (vu)	sim	sim	sim	sim	sim	sim	sim
escrow (e)	não	não	não	sim	sim*	não	não
shared value-change (shvc)	não	não	não	sim	não	sim	sim
shared slot (shs)	não	não	não	sim	não	sim	sim

* Sim, enquanto as reservas agregadas não violarem a restrição global.

Tabela 8.1: Tabela de compatibilidade das reservas.

Um cliente móvel requisita uma conjunto de reservas do servidor antes de se desconectar. Este conjunto deve ser definido com base nas operações que se espera serem efectuadas pelo utilizador até à próxima interacção com o servidor. A dedução de bons valores para este problema pode ser vista como uma extensão do problema da replicação antecipada [91, 87] (onde os clientes devem obter cópias dos dados que os utilizadores vão aceder). Como se referiu anteriormente, este problema está fora do âmbito desta dissertação e pode ser atacado usando técnicas de previsão [53].

Nas experiências apresentadas na secção 9.2 usou-se uma estratégia simples que adapta a previsão do comportamento futuro do cliente com base no comportamento verificado. Esta estratégia permite obter bons resultado numa aplicação de suporte a uma força de vendas móvel. A interface gráfica da aplicação desenvolvida permite, adicionalmente, especificar as reservas que devem ser obtidas de forma simples. Internamente as reservas são obtidas através de variantes da instrução *select*. Por exemplo, para requisitar uma reserva *escrow* de três unidades relativa ao produto com o identificador 3, o cliente submete a instrução *get escrow reservation stock instances 3 from products where id='5'*.

Quando um cliente solicita uma reserva, o servidor verifica se é possível conceder a reserva. Primeiro, o servidor verifica se existe alguma reserva concedida que impossibilite a concessão da nova reserva. Para tal, usa-se a tabela de compatibilidade apresentada na tabela 8.1, em que sim significa que é possível conceder uma reserva de um dado tipo mesmo que já tenha sido concedida um reserva do outro tipo que actue sobre o mesmo elemento de dados. Se os conjuntos de elementos de dados afectados por duas reservas não se intersectarem, ambas podem ser concedidas.

Segundo, o servidor verifica se o utilizador do cliente móvel pode obter a reserva requisitada. Esta verificação é efectuada com base em regras de autorização especificadas pelo administrador do sistema. Estas regras especificam que reservas cada utilizador pode obter e por quanto tempo podem ser concedidas. A composição destas regras [33] pode ser simples e estática ou complexa recorrendo a funções PL/SQL.

Quando uma reserva é concedida, o sistema Mobisnap tem de garantir que a promessa que lhe está

associada não é quebrada por nenhuma outra transacção. Para fazer cumprir estas promessas, ao mesmo tempo que permite aos clientes legados aceder directamente à base de dados central, as seguintes acções têm de ser executadas na base de dados (note-se que estas acções são desfeitas para permitir a execução de uma transacção que use uma dada reserva):

- Para cada reserva *value-change*, cria-se um gatilho (*trigger*) que impede a modificação do elemento de dados reservado. O gatilho impede a modificação lançando uma excepção. Neste caso, diz-se que a modificação foi bloqueada.
- Para cada reserva *value-lock*, cria-se um gatilho que impede a modificação do elemento de dados reservado por qualquer transacção, incluindo as transacção submetidas por um cliente que possua essa reserva. Para tal, ao contrário do que sucede com as outras reservas, esta acção não é desfeita para permitir a execução de uma transacção recebida de um cliente que possui esta reserva.
- Para cada reserva *slot*, cria-se um gatilho que impede qualquer transacção de inserir, remover ou modificar registos com os valores referidos na reserva.
- Para cada reserva *escrow*, actualiza-se o valor dos elementos de dados fungíveis como explicado na secção anterior.
- Para as reservas *value-use*, *shared slot* e *shared value-change* não é necessária qualquer acção na base de dados.

Para as reservas *value-change* e *slot* existe um opção adicional que pode ser usada: o pedido de reserva pode especificar uma actualização temporária que reflicta a impossibilidade de modificar o elemento de dados reservado enquanto a reserva permanecer válida. Por exemplo, considere-se um registo que representa um lugar num comboio. Quando um cliente móvel obtém uma reserva *value-change* sobre esse registo, o pedido de reserva pode especificar que se modifique para verdadeiro o valor da coluna que indica se o lugar está ocupado. Esta modificação indica que as transacções não garantidas por esta reserva não devem utilizar este lugar. Se apesar desta indicação, uma transacção tentar modificar o valor do registo, a modificação será bloqueada pelo gatilho criado aquando da concessão da reserva.

O sistema Mobisnap mantém registo de todas as reservas concedidas para verificar a possibilidade de conceder uma nova reserva (face às reservas já existentes) e para verificar a validade da utilização de uma reserva para garantir uma transacção móvel. Quando uma reserva expira ou é cancelada explicitamente pelo cliente, as acções executadas para garantir que a reserva é respeitada são desfeitas. As acções relativas a uma reserva são igualmente desfeitas (de forma temporária) para permitir a execução de uma transacção móvel que use essa reserva.

Como foi dito anteriormente, uma reserva é temporária (*leased*) [58], i.e., apenas é válida durante um período limitado de tempo. Esta propriedade garante que as restrições impostas às outras transacções, para fazer cumprir a promessa associada a um reserva, não duram indefinidamente, mesmo que o cliente móvel, a quem a reserva foi concedida, seja destruído ou fique permanentemente desconectado. Outra consequência da duração limitada das reservas consiste no facto da garantia fornecida num cliente móvel apenas ser válida se a transacção móvel for propagada para o servidor antes que as reservas usadas expirem.

8.3 Processamento das transacções móveis no cliente

Um cliente executa uma transacção móvel localmente em um ou dois passos, como se explica de seguida.

No primeiro passo, o cliente executa o programa da transacção para verificar se é possível garantir o seu resultado usando as reservas disponíveis. Este processo é descrito mais tarde. Se for possível garantir a transacção móvel, são actualizadas ambas as versões da réplica local dos dados. O processamento local da transacção termina, sendo a transacção móvel armazenada localmente até ser propagada para o servidor. Caso contrário, o sistema desfaz qualquer modificação executada à base de dados (abortando a transacção da base de dados na qual a transacção móvel foi executada) e prossegue para o segundo passo.

No segundo passo, o cliente executa o programa da transacção na versão provisória da réplica local da base de dados. O resultado desta execução é considerado provisório. Se o programa executa até uma instrução *commit* ou executa até ao fim do programa, o resultado é *tentative commit*. Neste caso, as modificações produzidas reflectem-se na réplica provisório dos dados. Se o programa executa até uma instrução *rollback*, o resultado é *tentative abort* e qualquer modificação produzida é desfeita.

Se, durante o processamento de uma transacção móvel (em ambos os passos), é necessário algum elemento de dados não replicado localmente para prosseguir a avaliação da transacção (por exemplo, a condição de uma instrução *if* usa o valor de uma coluna não replicada), a execução é abortada e o resultado da execução local é *unknown*.

8.3.1 Verificação da possibilidade de garantir uma transacção móvel

Nesta subsecção descreve-se o modo como o interpretador do cliente verifica se uma transacção móvel pode ser garantida. A ideia base consiste em executar o programa da transacção e verificar cada instrução do caminho de execução.

Garantir instruções Durante a execução, o interpretador mantém, para cada variável, além do seu valor actual, a informação sobre se esse valor é garantido e quais as reservas que o garantem (se alguma).

O *valor de uma variável é garantido* se foi atribuído numa instrução SQL de leitura garantida ou numa instrução de atribuição que não incluía nenhuma variável não garantida.

Uma instrução SQL pode ser garantida se: (1) todas as variáveis usadas na instrução SQL (como valores de entrada) são garantidas; (2) existe uma reserva que inclua os elementos de dados lidos ou escritos, i.e., que tenha uma condição compatível com a condição expressa na instrução SQL e inclua a coluna indicada. Para verificar a segunda condição, é necessário comparar a descrição semântica da reserva e a condição expressa na instrução SQL[3].

As seguintes restrições adicionais são aplicadas. Uma reserva *value-use* ou *value-lock* não pode garantir uma instrução de escrita (*update*, *insert*, *remove*). Uma reserva partilhada (*shared value-change* e *shared slot*) não pode garantir uma instrução de leitura (*select*).

Nas instruções de leitura é, por vezes, apenas possível obter uma garantia parcial do valor lido, i.e., a garantia que quando a transacção é executada no servidor o valor lido se encontra num dado intervalo. Por exemplo, se uma transacção móvel executa uma instrução de leitura que conte o número de registos que satisfazem uma dada condição e o cliente detém apenas uma reserva *value-change* que satisfaça a condição indicada, é possível garantir que, quando a instrução de leitura é executada no servidor, o resultado obtido é maior ou igual a um. Esta garantia, semelhante à obtida nas reservas *escrow*, pode ser suficiente para garantir o resultado da transacção móvel. Durante o processamento da transacção móvel no cliente, a utilização de uma variável com este tipo de garantia está sujeita às mesmas restrições impostas às variáveis garantidas por reservas *escrow* (explicadas anteriormente).

Quando se executa uma instrução de escrita SQL, ambas as versões da base de dados são actualizadas. Uma instrução de leitura SQL garantida devolve o valor reservado. Uma instrução de leitura SQL não garantida devolve o valor da versão provisória da base de dados.

Uma *instrução if é garantida* se todas as variáveis envolvidas na condição expressa são garantidas e permitem garantir o resultado da condição. Como se explicou na secção 8.1, as variáveis garantidas por reservas *escrow* não podem ser usadas para garantir o valor exacto de uma variável, mas apenas que este valor se encontra num dado intervalo. Assim, apenas podem ser usadas para garantir o valor de condições que exprimam relações de ordem ($<$, $\leq$, $>$, $\geq$).

Se a condição não pode ser garantida, assume-se que o seu valor é falso, por omissão. Assim, é possível garantir uma modificação alternativa (numa sequência de modificações alternativas guardadas por instruções *if*) quando não é possível garantir a opção preferida. Por exemplo, na transacção da figura 7.1, o cliente pode apenas deter reservas sobre o produto "RED THING". Neste caso, é possível garantir a encomenda deste produto, mas é impossível garantir a opção preferida pelo utilizador (relativa ao produto "BLUE THING"). Quando a transacção móvel é executada no servidor, por omissão, executa-se o

[3]Aproximações semelhantes são usadas na replicação secundária semântica [35] e na optimização das interrogações [4]).

mesmo caminho de execução, mesmo que a alternativa preferida pelo utilizador seja possível.

Uma aplicação pode modificar o processamento descrito usando as opções descritas na secção 7.3.2. Primeiro, é possível abortar a execução da transacção se uma condição não puder ser garantida. Segundo, é possível solicitar a execução do melhor caminho possível no servidor, independentemente do caminho garantido no cliente.

A aproximação usada para garantir as instruções *if* pode ser igualmente usada para garantir o valor de outras condições, como por exemplo as condições especificadas nas instruções de ciclo.

Resultado da execução Se um programa executa até uma instrução *commit*, o seu resultado é *reservation commit*. Neste caso, diz-se que o resultado da transacção é garantido independentemente no cliente. No entanto, dependendo das instruções que foi possível garantir, é atribuído um dos seguintes níveis de garantias:

- *Full* (completo), se todas as instruções no caminho de execução são garantidas por reservas. Note-se que, mesmo neste caso, é incorrecto aplicar, no servidor, o conjunto de escrita (*write set*) obtido aquando da execução da transacção no cliente, porque os elementos de dados fungíveis devem ser actualizados usando instruções de adição e remoção.

- *Read* (leitura), se for possível garantir todas as instruções com excepção das instruções de escrita. Neste caso (e nos seguintes), quando a transacção móvel é executada no servidor, a execução das escritas não garantidas pode ser bloqueada por uma reserva *value-change*, *value-lock* ou *slot*.

- *Pre-condition* (pré-condição), se é possível garantir todas as condições que determinam o caminho de execução. Neste caso, o sistema penas garante que a execução do programa de uma transacção seguirá o mesmo caminho de execução no servidor.

- *Alternative pre-condition* (pré-condição alternativa), se, pelo menos, uma condição que determina o caminho de execução não foi garantida. Neste caso, a aplicação pode forçar a execução do mesmo caminho de execução ou não no servidor (como se explicou anteriormente).

Como se exemplifica na secção 8.5, uma aplicação deve utilizar informação adicional (do domínio da semântica da aplicação) para interpretar o significado dos diferentes níveis de garantias. Por exemplo, os níveis de garantia *completo* e *leitura* são equivalentes se se souber que é impossível obter uma reserva que impeça a execução das instruções de escrita. Os resultados obtidos devem ser apresentados aos utilizadores utilizando esta informação adicional conhecida.

Se um programa executa até uma instrução *rollback*, o sistema aborta a execução, desfazendo todas as modificações efectuadas. O processamento da transacção móvel prossegue no segundo passo descrito no início desta secção.

Quando o resultado de uma transacção é *reservation commit*, o cliente associa automaticamente a informação sobre a execução do programa (reservas usadas, caminho de execução e valores lidos nas instruções de leitura) com a transacção. O cliente propaga esta informação para o servidor, o qual a usa durante a execução da transacção.

8.4 Processamento das transacções móveis no servidor

A execução de uma transacção móvel no servidor consiste na execução do programa da transacção móvel na base de dados central.

Transacção não garantida Se o resultado da transacção não foi garantido no cliente, o código da transacção deve fazer respeitar as intenções do utilizador através da especificação de regras de detecção e tratamento de conflitos apropriadas.

Durante a execução de uma transacção móvel no servidor, as escritas bloqueadas por reservas podem ser tratadas pela transacção (independentemente da transacção ter sido garantida no cliente ou não). Para tal, o código da transacção móvel deve *apanhar* a excepção lançada pelo gatilho que protege os elementos de dados reservados. Caso a excepção lançada não seja tratada, a transacção é abortada e as aplicações podem posteriormente verificar a causa do insucesso. As transacções abortadas podem ser reexecutadas usando o mecanismo de reexecução mencionado na secção 7.3.1.

Transacção garantida Se o resultado da transacção foi garantido no cliente, o interpretador que executa o programa da transacção móvel deve executar o seguinte conjunto de acções adicionais (para que as garantias fornecidas sejam respeitadas).

Antes de executar a transacção, para cada reserva usada para garantir o seu resultado no cliente, é necessário desfazer temporariamente (até à conclusão da execução da transacção) as acções que garantem o respeito por essas reservas. Dois casos especiais existem. Para uma reserva *escrow* é adicionado ou subtraído ao valor actual dos dados a quantidade de recursos usados na execução da transacção no cliente. Como estas reservas são parcialmente consumidas com a execução das transacções, nenhuma acção é executada no fim do processamento da transacção. Para uma reserva *value-use*, o valor actual dos dados é substituído pelo valor da reserva — após a execução da transacção o valor inicial é reposto.

Durante a execução da transacção, para cada instrução de leitura garantida no cliente, o interpretador devolve o valor relativo ao mesmo registo lido no cliente. Assim, esta propriedade estabelece uma ordem no conjunto de resultados obtidos na execução de uma instrução de leitura. No caso de a leitura ter sido garantida por uma reserva *value-use*, *value-change*, *value-lock* ou *slot*, esta propriedade garante que o mesmo valor é obtido no cliente e no servidor.

Quando o nível de garantia de uma transacção no cliente foi *alternative pre-condition*, o interpretador deve garantir que o caminho de execução respeita a opção especificada pela aplicação. Por omissão, o interpretador executa o mesmo caminho independentemente do resultado das condições especificadas nas instruções *if*. Como se referiu anteriormente, a aplicação pode forçar a execução do melhor caminho possível.

A execução de uma transacção móvel consome parcialmente as reservas *escrow* usadas. Por exemplo, se o cliente tinha uma reserva que lhe permitia subtrair três unidades ao valor de uma variável fungível e a transacção subtrai uma unidade, o cliente fica apenas com o direito de subtrair duas unidades a essa variável. Todas as outras reservas permanecem válidas (até expirarem ou serem devolvidas).

Após executar a transacção, o servidor refaz as acções necessárias ao respeito das reservas que permanecem válidas.

8.5 Exemplos

Nesta secção apresentam-se exemplos da utilização de reservas para garantir o resultado de transacções móveis.

Recursos fungíveis anónimos A figura 8.1 representa uma operação típica executada num sistema de suporte a uma força de vendas móvel: um pedido de encomenda. O pedido de encomenda é satisfeito se a existência disponível permite satisfazer o pedido e o preço do produto não é superior ao preço máximo que o comprador está disposto a pagar.

Na tabela 8.2 apresentam-se as reservas obtidas pelo vendedor móvel. Para cada reserva, mantém-se a seguinte informação: o identificador único da reserva, *id*; o *tipo* da reserva; as *colunas* reservadas nos registos seleccionados na *tabela* pela *condição*; o *valor* do(s) elemento(s) reservado(s); *informação* adicional específica para cada tipo de reserva (por exemplo, o tipo de restrição das reservas *escrow*); informação de *ligação* com outra reserva; e data de expiração da reserva. A data de expiração permite ao sistema não usar reservas que expirem antes do próximo contacto esperado com o servidor. A ligação permite associar uma reserva a outra, de modo a que apenas seja possível usar a primeira se se usar também a segunda. Neste exemplo, a reserva *value-use* está ligada à reserva *escrow*, de modo a que o vendedor móvel apenas possa garantir o preço reservado (com a reserva *value-use*) para as unidades de produto reservadas (com a reserva *escrow*).

Quando a transacção é executada no cliente, é óbvio que as duas reservas são necessárias para garantir o resultado da condição expressa na instrução *if* (linha 4). A reserva *escrow* garante que $prd_stock \geq 10$ é verdadeiro quando a transacção é executada no servidor. Na verdade, a reserva é mais geral e permite garantir a subtracção de até quinze unidades ao valor de *prd_stock* com a restrição global $prd_stock_{clt} \geq$

id	tipo	tabela	coluna	condição	valor	informação	ligação
45-1	*escrow*	products	stock	name='BLUE THING'	15	≥ 0	-
45-2	*value-use*	products	price	name='BLUE THING'	44.99	-	yes - 45-1

Tabela 8.2: Reservas obtidas por um vendedor móvel para garantir encomendas (data de expiração omitida).

```
1  ------ COMPRA 1 BILHETE: comboio = "London-Paris 10:00"; dia = '18-FEB-2002'; preço máximo = 100.00
2  BEGIN
3    SELECT price, available INTO tkt_price, tkt_avail FROM trains
4                          WHERE train='London-Paris 10:00' AND day='18-FEB-2002';
5    IF tkt_price <= 100.00 AND tkt_avail >= 1 THEN  --Verifica disponibilidade de lugar e se preço é aceitável
6      UPDATE trains SET available = tkt_avail - 1 WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002';
7                                     -- Selecciona e reserva um lugar específico para o comprador
8      SELECT seat INTO tkt_seat FROM tickets
9                  WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND used = FALSE;
10     UPDATE tickets SET used = TRUE, passenger = 'Mr. John Smith'
11                    WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND seat = tkt_seat;
12     NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Bilhete reservado...');
13     COMMIT (tkt_seat,tkt_price);      -- Conclui a transacção e devolve a informação
14   END IF;                             -- sobre o lugar reservado
15   ROLLBACK -1;
16 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossível adquirir bilhete ...');
17 END;
```

Figura 8.2: Transacção móvel para reservar um bilhete de comboio num sistema de reserva de bilhetes (o bloco de declaração de variáveis é omitido).

0 (ou a promessa que quando a transacção é executada no servidor, se tem $prd_stock \geq 15$). A reserva *value-use* garante que o preço é aceitável. A instrução *update* que actualiza a existência do produto (linha 5) é garantida pela reserva *escrow*. Esta instrução é usada para inferir a quantidade de recursos fungíveis utilizados. Neste caso, são usadas dez unidades, pelo que, após a execução da transacção, o cliente apenas mantém reservadas cinco unidades — este será o novo valor da reserva *escrow*. A instrução *insert* não é garantida por nenhuma reserva. Assim, o resultado da transacção será o nível *read* de *reservation commit*. Se a aplicação sabe que a inserção na tabela *orders* não pode ser bloqueada, é possível ter a certeza que a transacção executará com sucesso no servidor.

Para garantir o sucesso da instrução *insert* seria necessário obter uma reserva adicional. Uma hipótese consiste em obter uma reserva *shared slot* para a tabela *orders* (as propriedades da função *newid* garantem a unicidade do identificador). Uma segunda hipótese consiste em adicionar uma coluna à tabela *orders* que especifique o vendedor responsável pela encomenda. Assim, obtendo uma reserva *slot* para as encomendas introduzidas pelo próprio vendedor, seria possível garantir a instrução *insert*.

Recursos fungíveis não anónimos A transacção móvel da figura 8.2 representa um exemplo típico da utilização de recursos não anónimos: a reserva de um lugar num comboio. Neste exemplo, cada lugar é único e tem um identificador associado. Assim, após verificar que existe, pelo menos, um lugar disponível (linha 5), é necessário obter o identificador do lugar (linha 8-9) e actualizar a informação do

tipo	tabela	coluna	condição	valor	informação
escrow	trains	available	train='London-Paris 10:00'	2	≥ 0
value-use	trains	price	AND	95.00	-
value-change	tickets	*	day='18-FEB-2002'	$(4A, false, \ldots)$	-
value-change	tickets	*		$(4B, false, \ldots)$	-

Tabela 8.3: Reservas obtidas pelo garantir reservas de bilhetes de comboio (identificador, ligação e data de expiração omitidos).

tipo	tabela	coluna	condição	valor
value-lock	trainsids	*	true	todos os registos
value-use	trains	price	train=23 AND day='18-FEB-2002'	95.00
value-change	tickets	*	train=23 AND day='18-FEB-2002'	$(4A, false, \ldots)$
value-change	tickets	*	train=23 AND day='18-FEB-2002'	$(4B, false, \ldots)$

Tabela 8.4: Reservas obtidas pelo garantir reservas de bilhetes de comboio: segunda alternativa (identificador, ligação, informação e data de expiração omitidos).

nome do passageiro associada a esse lugar (linhas 10-11).

Vamos assumir que o cliente obteve reservas sobre dois lugares do comboio: as reservas respectivas são apresentadas na tabela 8.3. O processamento da transacção móvel prossegue de forma similar ao exemplo anterior até ser necessário obter o identificador do lugar a usar. A instrução *select* (linhas 8-9), que obtém o identificador do lugar, devolve um dos lugares reservados porque as instruções de leitura devolvem preferencialmente valores reservados. Neste caso, poder-se-ia obter, por exemplo, o lugar "4A". A instrução *update* (linhas 10-11) actualiza a informação relativa ao lugar "4A" com o nome do passageiro e o preço do bilhete. Estas instruções são garantidas pela reserva *value-change* que o cliente detém sobre o registo que mantém a informação do lugar "4A". Quando a transacção móvel é executada no servidor, o sistema garante que a instrução *select* devolve o mesmo registo, i.e., o registo relativo ao lugar "4A". Assim, o resultado da transacção no cliente é o nível *full* de *reservation commit* (todas as instruções são garantidas).

Se as reservas *value-change* não estivessem disponíveis, seria impossível garantir as instruções de leitura e escrita relacionadas com o lugar (linhas 8–11). O resultado da instrução *select* seria obtido a partir da versão provisória dos dados. Assim, o resultado da execução local da transacção seria o nível *pre-condition* de *reservation commit*. Para esta transacção, este resultado significa que é possível garantir a disponibilidade e preço do lugar, mas é impossível garantir o lugar que será atribuído ao passageiro.

Na figura 8.3 apresenta-se uma versão alternativa para reservar um lugar num comboio. Neste exemplo, a base de dados mantém uma tabela que associa o nome do comboio com o seu identificador (um inteiro). Adicionalmente, não existe informação sobre o número de lugares disponíveis: este valor é obtido contando o número de lugares não ocupados.

Na tabela 8.4 apresentam-se as reservas obtidas para garantir dois lugares. Relativamente à versão

```
1  ------ COMPRA 1 BILHETE: comboio = "London-Paris 10:00"; dia = '18-FEB-2002'; preço máximo = 100.00
2  BEGIN
3    SELECT id INTO train_id FROM trainsIds WHERE name = 'London-Paris 10:00';
4    SELECT price INTO tkt_price FROM trains WHERE train = train_id AND day='18-FEB-2002';
5    SELECT count(*) INTO tkt_avail FROM tickets WHERE train = train_id AND day='18-FEB-2002' AND used=FALSE;
6    IF tkt_price <= 100.00 AND tkt_avail >= 1 THEN  --Verifica disponibilidade de lugar e se preço é aceitável
7                                       -- Selecciona e reserva um lugar específico para o comprador
8      SELECT seat INTO tkt_seat FROM tickets
9                  WHERE train = train_id AND day = '18-FEB-2002' AND used = FALSE;
10     UPDATE tickets SET used = TRUE, passenger = 'Mr. John Smith'
11                    WHERE train = train_id AND day = '18-FEB-2002' AND seat = tkt_seat;
12     NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Bilhete reservado...');
13     COMMIT (tkt_seat,tkt_price);      -- Conclui a transacção e devolve a informação
14   END IF;                              -- sobre o lugar reservado
15   ROLLBACK -1;
16 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossível adquirir bilhete ...');
17 END;
```

Figura 8.3: Transacção móvel para reservar um bilhete de comboio num sistema de reserva de bilhetes: segunda alternativa (o bloco de declaração de variáveis é omitido).

tipo	**tabela**	**coluna**	**condição**	**valor**
slot	datebook	*	day='17-FEB-2002' AND *hour* $\geq$ 8 AND *hour* $\leq$ 13 AND name = 'J. Smith'	todos os registos que satisfazem a condição

Tabela 8.5: Reservas obtidas sobre uma agenda partilhada para garantir novas marcações (identificador, ligação, informação e data de expiração omitidos).

anterior, obtém-se adicionalmente a reserva *value-lock* que permite garantir a conversão efectuada entre o nome do comboio e o seu identificador. Esta conversão é efectuada no início da transacção móvel (linha 3). Como o valor obtido é garantido pela reserva *value-lock*, a utilização do identificador do comboio não introduz nenhum problema para a possibilidade de garantir a transacção.

A segunda diferença em relação à versão anterior consiste na inexistência de um elemento de dados com o número de lugares disponíveis (sobre o qual se obtinha uma reserva *escrow*). Neste caso, a transacção móvel conta o número de lugares não ocupados (linha 5). As reservas *value-change* permitem garantir que, aquando da execução da transacção móvel no servidor, existem, pelo menos, dois lugares disponíveis (uma garantia semelhante à obtida anteriormente pela reserva *escrow*). Assim, é possível garantir o resultado da condição da instrução *if* (linha 7). O processamento da transacção móvel prossegue de forma idêntica à versão anterior, permitindo obter como resultado o nível *full* de *reservation commit*.

Agenda partilhada Neste exemplo, supõe-se que existe uma agenda que mantém o conjunto de marcações de demonstrações a efectuar pelos vendedores de uma empresa. Esta agenda pode ser modificado por vários utilizadores (por exemplo, os vendedores e as secretárias da empresa).

Na figura 8.4 apresenta-se uma operação típica deste tipo de aplicação: a introdução de uma nova marcação. Supõe-se adicionalmente que o vendedor obteve uma reserva *slot* que lhe permite introduzir marcações em seu nome para a manhã do dia 17 de Fevereiro, como descrito na tabela 8.5.

```
1  --- INSERE MARCAÇÃO: '16-FEB-2002' às 10:00 OU '17-FEB-2002' às 10:00, vendedor='J. Smith'
2  BEGIN
3   id = newid;                                       --- Primeira alternativa ------------------------------
4   SELECT count(*) INTO cnt FROM datebook WHERE day='16-FEB-2002' AND hour>=9 AND hour<12 AND user='J. Smith';
5   IF (cnt = 0) THEN                                 --- Verifica disponibilidade
6     INSERT INTO datebook VALUES( id, '16-FEB-2002', 10, 'J. Smith', 'Demonstração BLUE THING');
7     NOTIFY( 'SMS', '351927435456', 'Demonstração marcada para 16-FEB-2002, 10:00');
8     COMMIT ('16-FEB-2002',10);                      --- Conclui e devolve marcação
9   ENDIF;                                            --- Segunda alternativa -------------------------------
10  SELECT count(*) INTO cnt FROM datebook WHERE day='17-FEB-2002' AND hour>=9 AND hour<12 AND user='J. Smith';
11  IF (cnt = 0) THEN                                 --- Verifica disponibilidade
12    INSERT INTO datebook VALUES( id, '17-FEB-2002', 10, 'J. Smith', 'Demonstração BLUE THING');
13    NOTIFY( 'SMS', '351927435456', 'Demonstração marcada para 17-FEB-2002, 10:00');
14    COMMIT ('17-FEB-2002',10);                      --- Conclui e devolve marcação
15  END IF;
16  ROLLBACK;
17 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Marcação impossível... ');
18 END;
```

Figura 8.4: Transacção móvel que introduz uma nova marcação numa agenda partilhada (o bloco de declaração de variáveis é omitido).

A reserva que o cliente possui não lhe permite garantir a instrução *select* referente ao dia 16 (linha 4). Assim, o resultado da condição da instrução *if* (linha 5) não pode ser garantido. Como se explicou anteriormente, por omissão, durante a verificação da possibilidade de garantir o resultado de uma transacção, considera-se que o resultado da condição é falso. A execução da transacção continua na linha 10. Neste caso, o resultado da instrução *select* é garantido pela reserva *slot* que o cliente possui. Em consequência, é possível garantir o resultado da instrução *if* (linha 11). A instrução de inserção (linha 12) é igualmente garantida pela reserva *slot*. Assim, o resultado da transacção é o nível *alternative pre-condition* de *reservation commit*. Este resultado garante que a marcação será efectuada, pelo menos, para o dia 17.

Quando a transacção móvel é executada no servidor, caso a aplicação tenha especificado a opção *melhor caminho*, a marcação será efectuada para o dia especificado em primeiro lugar (dia 16), caso tal seja possível. Se a aplicação não especificar essa opção, a transacção executará o mesmo caminho, sendo introduzida a marcação garantida no cliente.

Os exemplos apresentados nesta secção mostram como as reservas são usadas para garantir o resultado das transacções e o significado dos diferentes níveis de garantias em face de diferentes transacções. Adicionalmente, estes exemplos mostram que um sistema que pretenda fornecer garantias sobre diferentes aplicações necessita de fornecer diferentes tipos de garantias e que a combinação de diferentes garantias é, em geral, fundamental para garantir o resultado das transacções móveis no cliente. No próximo capítulo avalia-se quantitativamente a utilização do sistema de reservas, assim como se avalia qualitativamente a utilização do sistema Mobisnap para suportar aplicações de bases de dados num ambiente de computação móvel.

Capítulo 9

Avaliação do modelo básico do sistema Mobisnap

O sistema Mobisnap apresenta dois mecanismos principais: as transacções móveis e as reservas. Para que o sistema possa suportar adequadamente o desenvolvimento de aplicações para ambientes de computação móvel, é necessário que os mecanismos propostos possam ser utilizados para responder aos requisitos de múltiplos tipos de aplicações. Este aspecto, de avaliação qualitativa do sistema, é abordado na secção 9.1.

O sistema de reservas apenas permite confirmar o resultado de uma transacção móvel independentemente se o cliente móvel tiver obtido as reservas necessárias antes de ficar desconectado do servidor. Na secção 9.2 apresenta-se um estudo sobre a eficácia do sistema de reservas para uma aplicação de suporte a uma força de vendas móvel. Este estudo mostra que, no cenário considerado, as reservas permitem garantir o resultado de uma elevada percentagem de transacções móveis independentemente.

O sistema Mobisnap integra ainda um mecanismo de reconciliação de múltiplas transacções. Devido ao seu carácter genérico, este mecanismo é apresentado e discutido no próximo capítulo.

9.1 Aplicações

De seguida, descreve-se o modo como várias aplicações podem ser implementadas no sistema Mobisnap, explorando os mecanismos existentes no sistema.

9.1.1 Suporte a uma força de vendas

Uma aplicação de suporte a uma força de vendas mantém informação sobre os produtos disponibilizados por uma empresa. Esta aplicação deve permitir, a um vendedor móvel, executar as operações necessárias para satisfazer os pedidos dos clientes enquanto se encontra desconectado. As operações mais comuns que a aplicação deve suportar são as seguintes:

```
1  ------ CANCELA ENCOMENDA: id = 3
2  BEGIN
3    SELECT status,product,qty INTO l_status,l_prod,l_qty FROM orders WHERE id = 3;
4    IF l_status = 'to process' THEN
5        UPDATE orders SET status = 'cancelled' WHERE id = 3;
6        UPDATE products SET stock = stock + l_qty WHERE id = l_prod;
7        COMMIT;
8    ELSE
9        ROLLBACK;
10   END IF;
11 END;
```

Figura 9.1: Transacção móvel que cancela uma encomenda introduzida anteriormente (o bloco de declaração de variáveis é omitido).

1. acesso à informação sobre os produtos e clientes;
2. submissão de uma nova encomenda;
3. cancelamento de uma encomenda submetida anteriormente.

Para permitir o acesso à informação sobre produtos e clientes, o vendedor móvel obtém uma cópia parcial da informação presente na base de dados central antes de se desconectar. Esta cópia parcial deve incluir, pelo menos, informação sobre o subconjunto de clientes que o vendedor se propõe visitar e os produtos que esses clientes estão interessados. Na aplicação desenvolvida [32], os clientes e os vendedores são agrupados em zonas geográficas. A cópia parcial de cada vendedor inclui os clientes da zona que ele visita e os produtos que eles encomendaram anteriormente. O vendedor tem ainda a possibilidade de seleccionar dados adicionais para serem mantidos na sua cópia local.

O vendedor executa as operações de submissão e cancelamento de uma encomenda usando a interface gráfica da aplicação. Na submissão de uma encomenda o vendedor especifica o produto, quantidade e preço máximo aceitável para a encomenda. Adicionalmente, pode especificar valores alternativos caso seja impossível satisfazer o pedido do cliente. No cancelamento de uma encomenda, o vendedor especifica a encomenda aceite pelo sistema que deve ser cancelada.

As operações executadas pelos vendedores são expressas como transacções móveis criadas pela aplicação. Para produtos fungíveis, as transacções móveis das figuras 7.1 e 9.1 exemplificam, respectivamente, as operações de submissão e cancelamento de uma encomenda. A transacção de cancelamento obtém a informação sobre a encomenda a cancelar e, caso a encomenda ainda não tenha sido processada, efectua as necessárias actualizações. Para produtos não fungíveis, a figura 8.2 exemplifica a operação de submissão de uma encomenda (o cancelamento de uma encomenda é semelhante à operação de cancelamento para produtos fungíveis). Estas transacções exprimem a semântica dos pedidos efectuados pelos clientes, verificando se é possível satisfazer as condições expressas. Adicionalmente, seria possível resolver eventuais conflitos indicando modificações alternativas no código das transacções móveis.

```
1   ------ REMOVE MARCAÇÃO: id=3476554
2   BEGIN
3     SELECT count(*) INTO cnt FROM datebook WHERE id = 3476554;
4     IF (cnt > 0) THEN
5       DELETE FROM datebook WHERE id = 3476554;
6       COMMIT;
7     END IF;
8     ROLLBACK;
9   END;
```

Figura 9.2: Transacção móvel que remove uma marcação numa agenda partilhada (o bloco de declaração de variáveis é omitido).

O resultado da submissão de uma nova encomenda pode ser garantida usando reservas *escrow*, *value-use* e *value-change* como se descreveu na secção 8.5. Assim, os vendedores podem obter as reservas necessárias para garantir as encomendas que esperam receber antes de se desconectarem (uma avaliação da eficiência desta aproximação é apresentada na secção 9.2). As reservas obtidas permitem garantir que a encomenda pode ser satisfeita com a existência do produto (*stock*) disponível.

Na aplicação desenvolvida, o resultado do cancelamento de uma encomenda nunca é garantida de forma independente. Este facto resulta da decisão de permitir que o processamento das encomendas possa ser iniciado em qualquer momento. Sem esta decisão, seria possível garantir o cancelamento de uma encomenda obtendo uma reserva *value-change* sobre o registo que descreve a encomenda. No entanto, esta reserva impede o início do processamento da encomenda no servidor porque não permite que se actualize o seu estado.

A aplicação desenvolvida é detalhada em [32]. A interface da aplicação foi desenvolvida usando *Java servlets* [157]. A aplicação executa nos clientes móveis usando o servidor de *servlets* Jetty [106], cuja reduzida dimensão se adequa às características dos dispositivos móveis.

9.1.2 Agenda partilhada

No âmbito da aplicação de suporte a uma força de vendas desenvolveu-se igualmente uma aplicação de agenda partilhada. Esta aplicação permite gerir as demonstrações de produtos executadas pelos vendedores. Assim, existe uma operação que introduz uma nova demonstração (figura 8.4) e uma operações que remove uma demonstração anteriormente marcada (figura 9.2). Estas operações podem ser executadas pelo vendedor ou pela secretaria da empresa.

É possível confirmar independentemente o resultado da introdução de novas marcações usando reservas, como se explica na secção 8.5. Um vendedor pode também garantir o cancelamento das suas marcações obtendo reservas *slot* sobre os registos das suas marcações.

9.1.3 Sistema de reserva de bilhetes de comboio

As operações disponibilizadas num sistema de reserva de bilhetes de comboio são semelhantes às da aplicação de suporte a uma força de vendas na qual os produtos transaccionados não são fungíveis. Assim, é possível usar a mesma aproximação para garantir operações de forma independente em vários computadores, i.e., sem acesso ao servidor central. Esta aproximação permite, não só, ultrapassar situações de desconexão com o servidor central, mas também, diminuir a latência da execução das operações e reduzir os picos de carga do servidor central, diferindo a execução das operações confirmadas independentemente para momentos de carga mais reduzida.

A transacção que reserva um lugar num comboio foi apresentada na figura 8.2. A transacção que cancela uma reserva pode ser definida de forma semelhante ao cancelamento de uma encomenda (com a adicional libertação do lugar reservado). É possível garantir o resultado dos pedidos de reserva independentemente usando reservas, como se explica na secção 8.5.

9.1.4 Metodologia de concepção das transacções móveis

As aplicações anteriores exemplificam o modo como os mecanismos incluídos no sistema Mobisnap podem ser usados para suportar a execução durante períodos de desconexão.

Uma transacção móvel deve incluir a verificação das pré-condições de execução da operação e, se possível, regras de resolução de eventuais conflitos. Assim, pretende-se garantir o respeito pelas intenções do utilizador e resolver os conflitos que surjam de forma apropriada. Apesar desta aproximação adicionar alguma complexidade à definição de uma transacção móvel, os exemplos apresentados relativos a várias aplicações mostram que é simples criar uma transacção móvel. Aproximações semelhantes foram adoptadas anteriormente em vários sistemas [161, 59], o que sugere que esta aproximação pode ser facilmente utilizada num grande número de aplicações.

As reservas permitem garantir independentemente o resultado das operações e não introduzem complexidade adicional na especificação das transacções móveis (porque a verificação da sua utilização é transparente). Os exemplos anteriores mostram que as reservas existente são suficientes para garantir o resultado de diferentes tipos de operações. Adicionalmente, a pequena granularidade das reservas permite obter reservas apenas sobre os elementos de dados necessários, não impondo restrições desnecessárias a operações executadas sobre outros dados.

9.2 Reservas

Nesta secção apresenta-se uma estudo da eficiência do sistema de reservas no suporte a uma força de vendas móvel. A principal medida de eficiência usada é a taxa de transacções confirmadas indepen-

dentemente em diferentes cenários de utilização do sistema e usando diferentes políticas de distribuição de reservas entre os clientes e os servidores. Os resultados obtidos mostram que é possível garantir o resultado de uma elevada taxa de transacções de forma independente. Nos piores cenários estudados, este valor é superior a 90% e 80%, quando, respectivamente, existe e não existe uma previsão fiável das encomendas recebidas, tomando como referência as transacções que podem ser aceites.

9.2.1 Modelo das experiências

As experiências apresentadas nesta secção simulam a utilização de uma aplicação de suporte a uma força de vendas móvel, semelhante à aplicação descrita na secção 9.1.1 ou a um sistema de reserva de lugares (secção 9.1.3).

Esta aplicação mantém informação sobre um conjunto de produtos, incluindo, para cada produto, a existência (*stock*) disponível e o preço. Os vendedores móveis usam um dispositivo portátil para submeter as encomendas recebidas dos seus clientes. Estes pedidos são submetidos como transacções móveis idênticas à transacção móvel apresentada na figura 8.1. Para garantir o resultado dos pedidos de encomenda independentemente, os clientes obtém reservas *escrow* e *value-use*.

Nas experiências realizadas assume-se, sem perda de generalidade, que apenas existe um produto disponível e que o seu preço é igual a um. Assim, o *valor de uma encomenda* é igual ao número de unidades encomendadas. Nestas experiências usa-se apenas um tipo de transacção móvel: a submissão de uma nova encomenda. Como se explicou na secção 9.1.1, para garantir o cancelamento de uma encomenda seria necessário restringir o seu processamento no servidor, o que não parece muito realista. Adicionalmente, para garantir o cancelamento de uma encomenda seria necessário que o cliente que recebe a operação tivesse uma reserva *value-change* sobre essa encomenda. Assumindo que cada vendedor obtinha este tipo de reservas sobre todas as encomendas que recebeu, seria possível garantir todos os pedidos de cancelamento recebidos pelo mesmo vendedor em que tinha sido efectuado o pedido de encomenda.

As experiências efectuadas simulam a execução do sistema Mobisnap, incluindo o servidor central, um conjunto de clientes e o sistema de comunicações. Os parâmetros das experiências são apresentados nas tabelas 9.1,9.2 e 9.3, que se explicam de seguida.

Cada experiência simula um período de doze horas. Os parâmetros de falha do servidor, descritos na tabela 9.1, levam a uma disponibilidade de $99,7\%$.

Um módulo de comunicações simula as comunicações entre os clientes e o servidor. Nas experiências apresentadas modelam-se dois cenários simples, descritos na tabela 9.1. O primeiro, *MOV*, modela um ambiente de computação móvel no qual os clientes permanecem desconectados durante longos períodos de tempo. Neste cenário, cada cliente fica impossibilitado de comunicar com o servidor durante

Nome	**móvel (*MOV*)**	**distribuído (*DIST*)**
Duração da experiência (t)	12 horas	
Tempo entre falhas no servidor	Exp(6 horas)	
Duração das falhas no servidor	Exp(1 min)	
Tempo entre falhas na comunicação cliente/servidor	Exp(2 horas)	
Duração das falhas nas comunicações cliente/servidor	Exp(36 min)	Exp(2 min)
Latência de propagação das mensagens	Exp(400 ms)	Exp(200 ms)

Tabela 9.1: Parâmetros das experiências: duração e falhas.

Nome	**pequeno (*PEQ*)**	**grande (*GRD*)**
Número de clientes	8	50
Existência inicial ($stock_{init}$)	300	30000
Quantidade em cada encomenda (valor da encomenda)	round(0.5 + Exp(1.5))	round(0.5 + Exp(19.5))

Tabela 9.2: Parâmetros das experiências: configuração do sistema simulado.

aproximadamente 30% do tempo. Esta impossibilidade é modelada através de falhas na comunicação entre o cliente e o servidor (injectando eventos de falha e recuperação na simulação). No entanto, podem existir várias causas para esta impossibilidade: desconexão voluntária, restrições de energia, etc. A latência das comunicações tem distribuição exponencial, modelando um tempo de comunicação variável. Uma mensagem com latência superior a um segundo é considerada perdida.

O segundo cenário, *DIST*, modela um ambiente de computação distribuída. Cada cliente está impossibilitado de comunicar com o servidor, em média, $1,6\%$ do tempo (o que está de acordo com estudos recentes de conectividade em redes distribuídas de larga escala [26, 120]).

Na tabela 9.2 descrevem-se as configurações do sistema estudadas. A primeira, *PEQ*, simula uma pequena força de vendas móvel. Nesta configuração, existem apenas 8 clientes e a existência inicial é igual a 300 unidades. A quantidade envolvida em cada encomenda é variável e baseada numa distribuição exponencial com valor médio igual a duas unidades. Esta distribuição leva à criação de encomendas geralmente pequenas com um pequeno número de encomendas envolvendo grandes quantidades. A segunda configuração, *GRD*, corresponde a um sistema com maior número de clientes, uma existência inicial e um valor médio por encomenda mais elevados. Esta configuração é usada para avaliar a influência da escala no comportamento do sistema.

Os pedidos de encomenda são gerados nos clientes, usando os parâmetros descritos na tabela 9.3, da

Nome	**boa previsão (*BOM*)**	**má previsão (*MAU*)**
Taxa esperada de utilização (e_u)	variável	
Fiabilidade da previsão ($sigma_{base}$)	0.04	0.64
Exactidão da previsão face ao observado	$\approx$ 10% em *PEQ* $\approx$ 5% em *GRD*	$\approx$ 55%

Tabela 9.3: Parâmetros das experiências: exactidão das previsões.

seguinte forma. Primeiro, para cada experiência, a taxa esperada de utilização, e_u, controla o valor esperado de todas as encomendas recebidas, exp_{total} : $exp_{total} = e_u \times stock_{init}$. Nas experiência apresentadas, e_u varia entre 55% e 175%, modelando desde uma procura reduzida até uma procura bastante forte.

Segundo, para cliente, gera-se um valor previsto de encomendas recebidas, exp_i. exp_i é criado aleatoriamente de forma a que $exp_{total} = \sum exp_i$. Na prática, podem-se usar técnicas de previsão [53] para obter este valor a partir da história passada dos clientes.

Terceiro, gera-se, a partir do parâmetro de fiabilidade da previsão, $sigma_{base}$, o valor real (esperado) das encomendas recebidas em cada cliente, dem_i. dem_i é obtido a partir de uma variável aleatória com distribuição normal (com $valor_medio = exp_i$ e $sigma = sigma_{base} \times exp_i$).

Finalmente, durante cada simulação, a submissão de cada pedido de encomenda é controlado por uma variável aleatória com distribuição exponencial, como é usual. O tempo médio entre a submissão de duas encomendas num cliente é igual a $t \times \frac{req_{avg}}{dem_i}$, com req_{avg} o valor médio de cada encomenda. O modo de criação dos pedidos leva a que, para cada cliente, o valor das encomendas recebidas possa diferir do valor real, dem_i.

Nas experiências realizadas, relativamente à qualidade das previsões, analisaram-se dois cenários. O primeiro, *BOM*, em que o valor previsto está muito próximo do valor observado. O segundo, *MAU*, em que as previsões são más. Na tabela 9.3, mostra-se, para cada cenário, a diferença média entre o valor das encomendas criadas e o valor previsto (exp_i).

Nas experiências realizadas avaliaram-se duas estratégias para obter reservas. Em ambas, os clientes obtém reservas que expiram apenas no fim da simulação. Na primeira estratégia, *simp X*, os clientes obtém reservas apenas uma vez, no início das experiência. *X* é a fracção da existência do produto que o servidor reserva para si próprio (i.e., que os clientes não podem reservar). O resto da existência é reservado pelos clientes (em conjunto com as reservas *value-use*), proporcionalmente ao valor de encomendas previsto (exp_i).

A segunda estratégia, *din X*, estende a primeira permitindo aos clientes obter reservas adicionais quando as reservas que possuem já não permitem garantir uma encomenda de valor médio. O servidor concede reservas adicionais proporcionalmente às reservas obtidas anteriormente (de forma a que todos os clientes tenham hipótese de obter reservas adicionais). Assim, um cliente *i* pode obter reservas adicionais até $(1 - X) \times (stock_{init} - \sum used_n) \times \frac{used_i}{\sum used_n}$ unidades da existência, com $used_i$ o número de unidades reservadas anteriormente pelo cliente *i*.

Nas experiências apresentadas, as reservas são sempre obtidas a partir do servidor. Esta aproximação difere de outros trabalhos [25] em que os algoritmos de redistribuição de recursos tangíveis envolvem múltiplos servidores. Este tipo de aproximação não parece apropriado aos ambientes de computação móvel a que se destina o sistema Mobisnap, nos quais os computadores que integram o sistema podem

permanecer desconectados durante longos períodos de tempo, mas antes, a ambientes de computação distribuída com boa conectividade entre os vários servidores.

Para avaliar a eficiência do sistema de reserva obtiveram-se os seguintes resultados em cada experiência:

- Fracção de encomendas (em valor) confirmadas independentemente nos clientes.
- Fracção de encomendas (em valor) confirmadas independentemente nos clientes ou imediatamente no servidor, i.e., encomendas que podem ser confirmadas com sucesso no servidor, caso exista conectividade, usando a existência do produto não reservada.

Como elemento de comparação, determinou-se, para cada experiência, o valor das encomendas que podem ser executadas com sucesso num sistema cliente-servidor, em que o cliente executa imediatamente todas os pedidos de encomenda no servidor. Quando não é possível contactar o servidor após três retransmissões, o cliente desiste de propagar a encomenda. Estes resultados são apresentados com a legenda *clt/srv*.

Como o valor máximo das transacções que podem ser executadas com sucesso depende da taxa de utilização, obteve-se igualmente o valor máximo das transacções que podem ser executadas num sistema distribuído perfeito em que nem os servidores nem as comunicações falham e com latência nula. Note-se que, usando o mecanismo de reavaliação das transacções, é sempre possível obter este valor máximo quando as reservas expiram (i.e., no fim da simulação).

Cada experiência simula o comportamento do sistema durante um período de doze horas. Todos os resultados apresentados são a média de dez simulações. Quando se comparam diferentes aproximações, usam-se os mesmos eventos de geração de encomendas.

9.2.2 Previsão fiável

Neste primeiro conjunto de experiências, avalia-se o cenário em que as previsões disponíveis nos clientes são boas. A diferença entre a previsão de encomendas recebidas e o valor real observado é, em média, 10% no cenário pequeno, *PEQ*, e 5% no cenário grande, *GRD*.

Por omissão, os resultados apresentados correspondem à configuração com pequeno número de cliente, *PEQ*, no ambiente computação móvel, *MOV*.

Confirmação independente dos resultados A figura 9.3 apresenta o valor das transacções que podem ser confirmadas com sucesso nos clientes de forma independente em função da taxa de utilização. Os valores apresentados são em percentagem do valor total de encomendas submetidas (gráfico da esquerda)

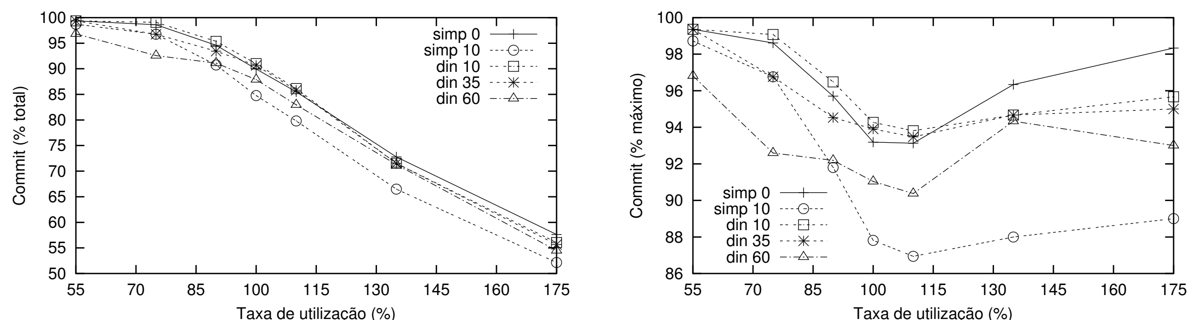

Figura 9.3: Transacções aceites localmente (cenário *MOV:PEQ:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

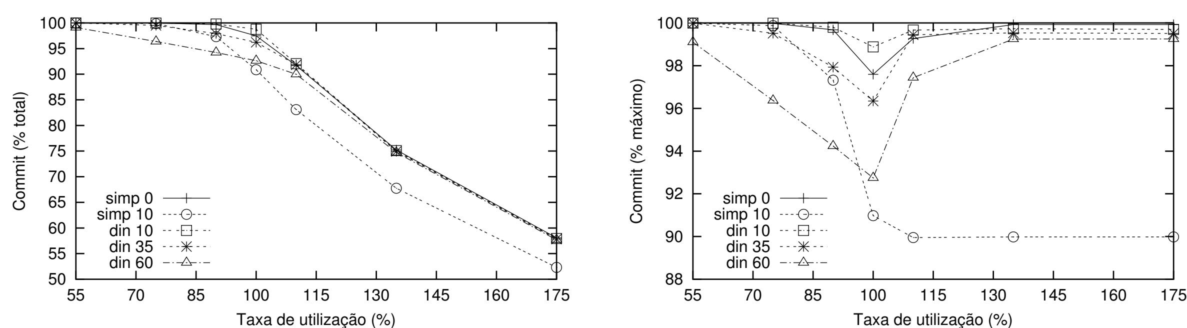

Figura 9.4: Transacções aceites localmente (cenário *MOV:GRD:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

e do valor máximo de encomendas que podem ser aceites num sistema cliente/servidor sem falhas (gráfico da direita). Os resultados mostram que é possível confirmar localmente, de forma independente, o sucesso de mais de 85% do máximo de encomendas que podem ser aceites, quer o cliente obtenha ou não reservas adicionais durante a simulação.

Como se esperava, este valor diminui quando a taxa de utilização aumenta e se aproxima de 100%. Quando a taxa de utilização é inferior a 100%, cada cliente obtém reservas que excedem as necessidades previstas. Estas reservas em excesso são usadas para confirmar encomendas não previstas.

Quando a taxa de utilização é superior a 100%, cada cliente apenas obtém reservas para um parte das encomendas previstas. Assim, mesmo que as encomendas recebidas sejam inferiores às previstas, as encomendas tendem a consumir todas as reservas que o cliente obtém. No entanto, ao contrário do que seria de esperar, os resultados da figura 9.3 não mostram nenhuma melhoria significativa no valor das transacções confirmadas localmente quando a taxa de utilização é superior a 100% (com excepção de "simp 0"). O estudo desta situação permitiu concluir que este facto se deve ao modo como os servidores satisfazem os pedidos de novas reservas. Assim, ao reservarem para si, em cada momento, uma fracção da existência disponível, levam a que os clientes não consigam obter novas reservas (em número suficiente) quando a existência disponível é escassa. Como esta situação apenas ocorre quando a existência

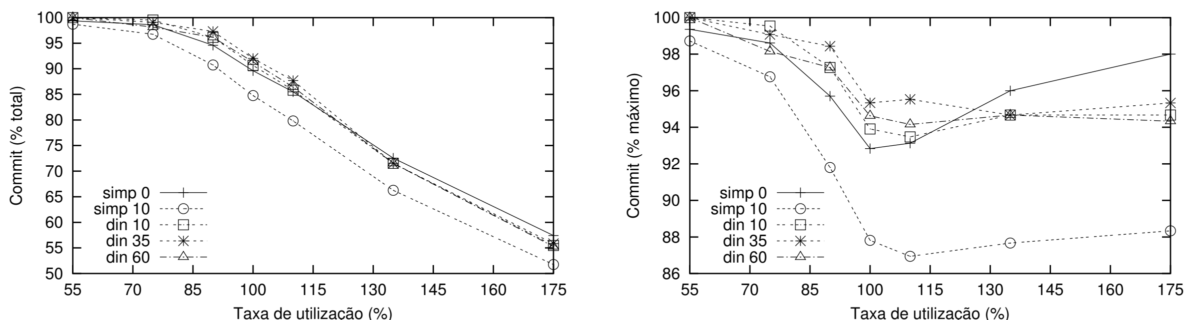

Figura 9.5: Transacções aceites localmente (cenário *DIS:PEQ:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

disponível se aproxima de zero, ela afecta um número semelhante de encomendas independentemente da existência inicial. Quando o valor total das encomendas aumenta, o valor relativo deste efeito torna-se desprezável, como se pode observar na figura 9.4, relativa ao cenário *GRD*. Os resultados apresentados mostram que, neste caso, quando a taxa de utilização se afasta de 100%, o valor das transacções confirmadas localmente aumenta para valores superiores a 99% do máximo possível.

Quando os clientes apenas obtém reservas no início da simulação, a existência reservada pelo servidor não pode ser usada para confirmar encomendas nos clientes. Este facto explica a diminuição da taxa de confirmação local para valores inferiores a 90% em "simp 10", quando a taxa de utilização é superior a 100%.

Os resultados das figuras 9.3 e 9.4 mostram que a taxa de sucesso local diminui com o aumento da fracção da existência reservada pelo servidor. Quando os clientes obtém reservas adicionais, este facto fica a dever-se a uma maior susceptibilidade dos clientes às falhas na rede, pela necessidade de contactarem mais frequentemente o servidor. Os resultados relativos ao cenário de computação distribuída, apresentados na figura 9.5, corroboram esta explicação.

Confirmação imediata dos resultados Os resultados anteriores mostram que é possível confirmar independentemente nos clientes mais de 85% das encomendas submetidas. Quando se esgotam as reservas que um cliente possui, ele pode confirmar imediatamente (mas não independentemente) o sucesso de uma encomenda contactando o servidor, caso o servidor tenha reservado para si uma fracção da existência disponível.

A figura 9.6 apresenta o valor das transacções que podem ser confirmadas com sucesso imediatamente nos clientes ou no servidor em função da taxa de utilização. Os resultados mostram que é possível confirmar imediatamente o sucesso de mais de 95% das encomendas efectuadas. Este valor aproxima-se de 100% quando a taxa de utilização se afasta de 100%. Adicionalmente, e ao contrário das reservas aceites localmente (figura 9.3), observa-se que este valor aumenta com o aumento da fracção

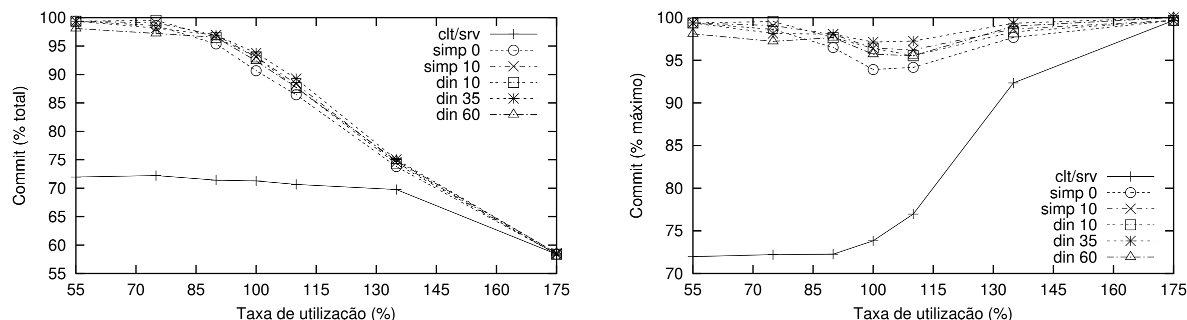

Figura 9.6: Transacções aceites imediatamente no cliente ou no servidor (cenário *MOV:PEQ:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

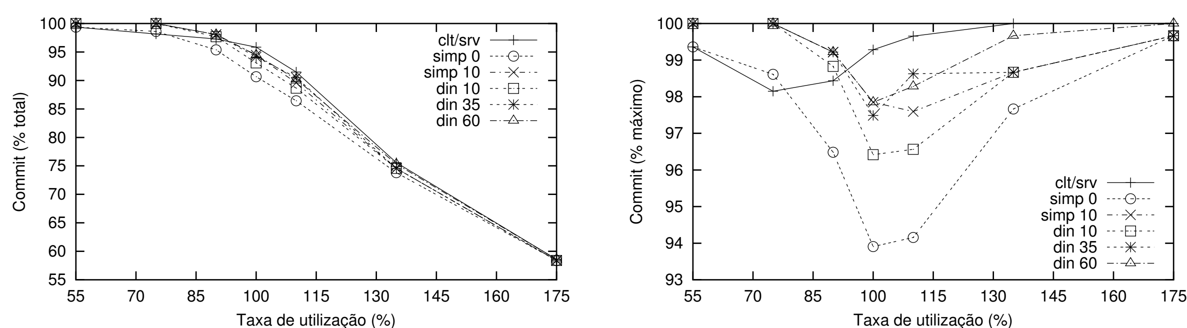

Figura 9.7: Transacções aceites imediatamente no cliente ou no servidor (cenário *DIS:PEQ:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

reservada pelo servidor (com excepção de “din 60”) — *simp 10* melhor que *simp 0* e *din 35* melhor que *din 10*.

O valor das encomendas que podem ser aceites imediatamente num sistema cliente/servidor é bastante inferior e está directamente relacionado com o tempo de desconexão. Assim, no ambiente de computação móvel *MOV*, em que o cliente permanece desconectado aproximadamente 30% do tempo, o valor das transacções aceites imediatamente é ligeiramente superior a 70% (este valor é superior a 70% porque o mecanismo de retransmissão de mensagens permite atenuar o efeito das falhas).

Ao contrário do que seria de esperar, o valor das transacções cujo sucesso pode ser determinado imediatamente não alcança 100%. Após análise das simulações executadas observou-se que este facto se devia a dois factores. Primeiro, quando a taxa de utilização é inferior a 100%, as falhas na rede levam à impossibilidade de contactar o servidor para confirmar algumas encomendas. Assim, quando as falhas na rede são mínimas, é possível confirmar imediatamente o sucesso do máximo de encomendas possíveis, como se pode observar na figura 9.7, relativo a um ambiente de computação distribuída.

Segundo, quando a taxa de utilização é superior a 100%, os clientes obtém, por vezes, reservas que não utilizam, quer porque não recebem mais encomendas, quer porque as reservas não são suficientes para garantir as encomendas recebidas. Este facto é consequência do reduzido número de encomendas

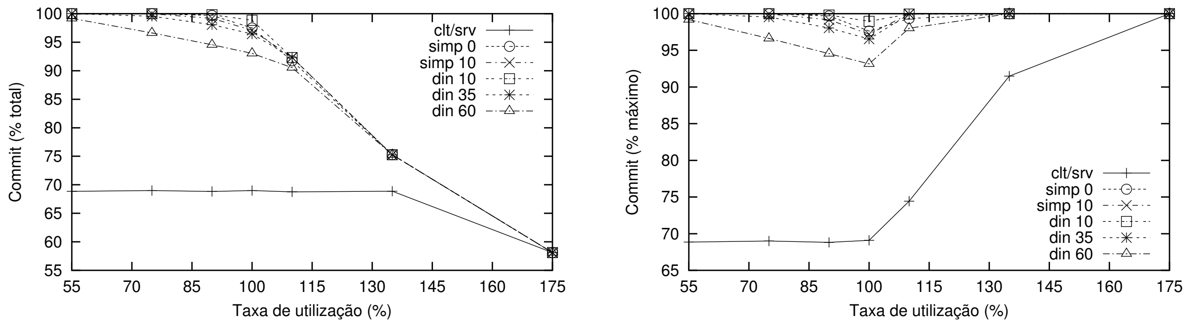

Figura 9.8: Transacções aceites imediatamente no cliente ou no servidor (cenário *MOV:GRD:BOM*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

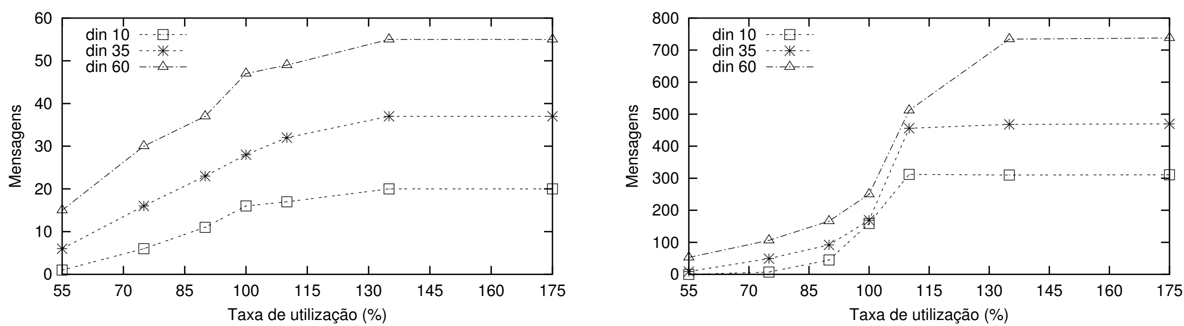

Figura 9.9: Mensagens enviadas na obtenção de novas reservas (cenário *MOV:PEQ:BOM* à esquerda e *MOV:GRD:BOM* à direita).

recebidas por alguns clientes. Quando cada cliente recebe, em média, um número mais elevado de encomendas, estas situações tendem a ser em menor número e a tornarem-se desprezáveis, como se observa na figura 9.8, relativo ao cenário *GRD* num ambiente de computação móvel.

Mensagens de obtenção de novas reservas Na figura 9.9 apresenta-se o número de mensagens propagadas na obtenção de novas reservas. Este valor inclui todas as tentativas de propagação, pelo que o envio de um pedido e a recepção da respectiva resposta é contabilizado como duas mensagens, caso não ocorra nenhum erro.

Os resultados mostram que o número de mensagens trocadas aumenta com a fracção da existência reservada pelo servidor. Este facto era esperado porque quanto maior o valor reservado pelo servidor, menor o valor das reservas concedidas, em cada momento, pelo servidor.

Adicionalmente, o número de mensagens enviadas aumenta com o aumento da taxa de utilização. Este aumento tende a ser mais significativo para valores próximos dos 100%. Estes factos são consequência da necessidade de obter mais reservas quando o número de encomendas aumenta e de o servidor conceder um menor número de reservas quando a existência disponível é reduzida (o que apenas acontece quando a taxa de utilização se aproxima dos 100%).

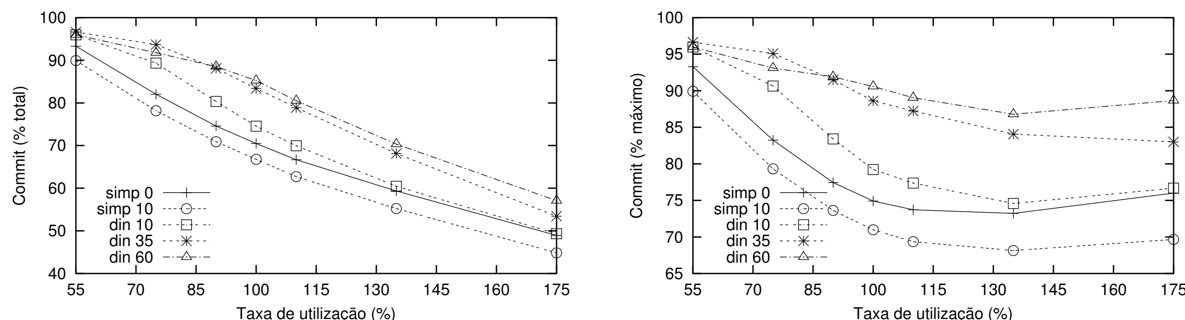

Figura 9.10: Transacções aceites localmente (cenário *MOV:PEQ:MAU*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

Quando o servidor informa os clientes que não pode conceder reservas adicionais, os clientes não pedem novas reservas. Contudo, os clientes verificam periodicamente (com um período longo dependente das encomendas recebidas) se a situação de indisponibilidade se mantém. Este facto justifica a quase estagnação do número de mensagens trocadas quando a taxa de utilização é superior a 110%.

Os resultados apresentados mostram ainda que número de mensagens por encomenda recebida é superior quando a existência é mais reduzida. Este facto é igualmente consequência do menor valor das reservas concedidas pelo servidor quando a existência disponível é mais reduzida. No entanto, mesmo no cenário *PEQ*, o número de mensagens trocadas é aceitável. Por exemplo, em *din 35*, para uma taxa de utilização de 110% são enviadas 32 mensagens, correspondentes a 16 pedidos de novas reservas. Este valor corresponde a uma média de 2 pedidos por cada cliente (incluindo os pedidos recusados e os que falham devido a falhas na rede).

9.2.3 Previsão não fiável

Neste segundo conjunto de experiências, estuda-se o cenário em que a previsão das encomendas recebidas se afasta muito da realidade observada. A diferença entre a previsão de encomendas recebidas e o valor real observado é, em média, 55%. Como anteriormente, os resultados apresentados correspondem à configuração com um pequeno número de clientes, *PEQ*, no ambiente de computação móvel, *MOV*, caso não exista outra informação.

Confirmação independente dos resultados obtendo reservas inicialmente Na figura 9.10 apresenta-se o valor das transacções que podem ser confirmadas com sucesso nos clientes. Como seria de esperar, os resultados são bastante piores do que quando a previsão das encomendas recebidas é fiável. Neste caso, os melhores resultados são obtidos quando os clientes obtém reservas adicionais durante a execução da simulação. Por exemplo, quando o servidor reserva inicialmente 60% da existência do produto, é possível garantir com sucesso mais de 85% do máximo de encomendas possíveis.

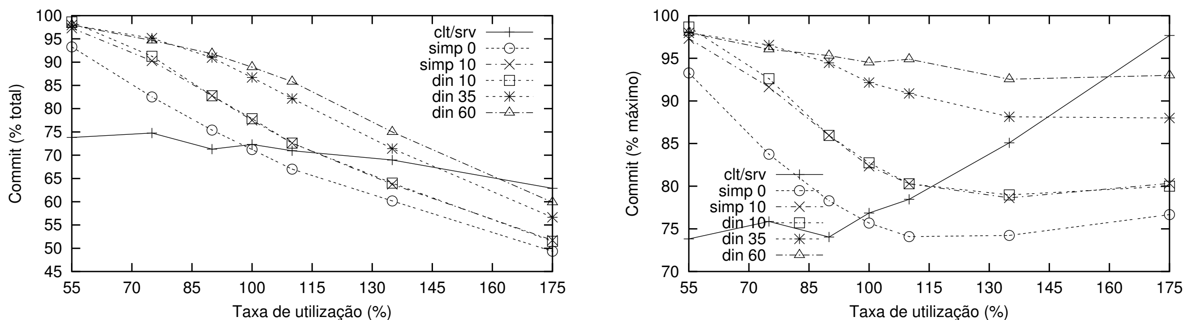

Figura 9.11: Transacções aceites imediatamente no cliente ou no servidor localmente (cenário *MOV:PEQ:MAU*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

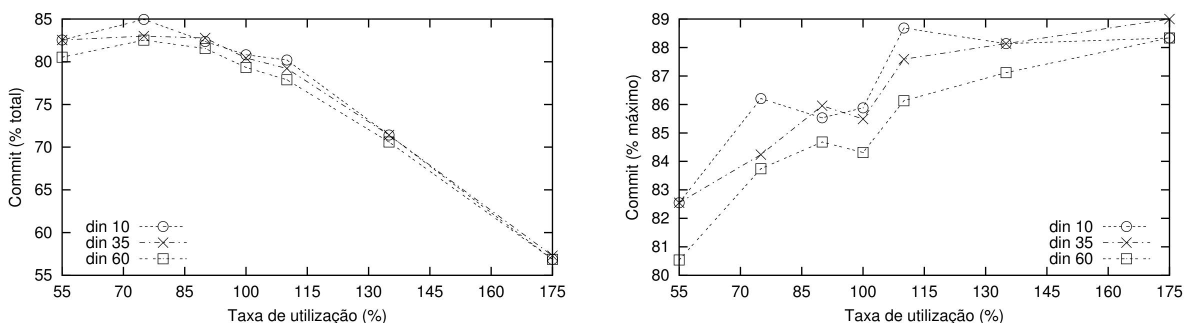

Figura 9.12: Transacções aceites localmente quando os clientes não obtém inicialmente nenhuma reserva (cenário *MOV:PEQ:MAU*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

Quando a taxa de utilização é pequena, a fracção das encomendas confirmadas localmente é muito elevada. Por exemplo, para uma taxa de utilização de 55%, é possível confirmar localmente mais de 95% das encomendas quando se obtém reservas adicionais. Este facto é consequência do excesso de oferta face à procura.

Confirmação imediata dos resultados obtendo reservas inicialmente Na figura 9.11 apresenta-se o valor das transacções que podem ser confirmadas imediatamente num cliente ou no servidor. Ao contrário do esperado, estes resultados mostram apenas uma ligeira melhoria relativamente ao resultado das transacções confirmadas localmente nos clientes. Analisando as experiências efectuadas foi possível descobrir que este facto se deve às reservas obtidas inicialmente, as quais são feitas com base em previsões que se verificam ser incorrectas.

Confirmação independente dos resultados não obtendo reservas inicialmente Para eliminar a influência negativa das previsões incorrectas no sistema de reservas, estudou-se o comportamento do sistema usando a estratégia de obtenção dinâmica de reservas, *din X*, não obtendo inicialmente nenhuma

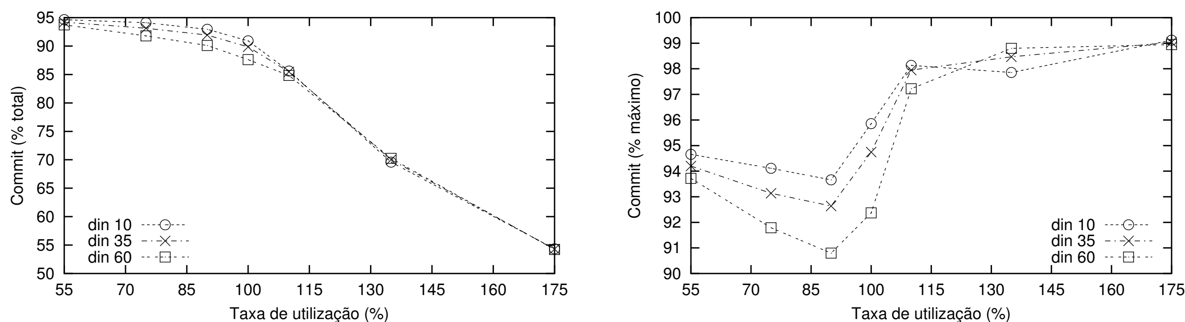

Figura 9.13: Transacções aceites localmente quando os clientes não obtém inicialmente nenhuma reserva (cenário *MOV:GRD:MAU*): valor relativo ao total de encomendas efectuadas (esquerda) e ao máximo possível (direita).

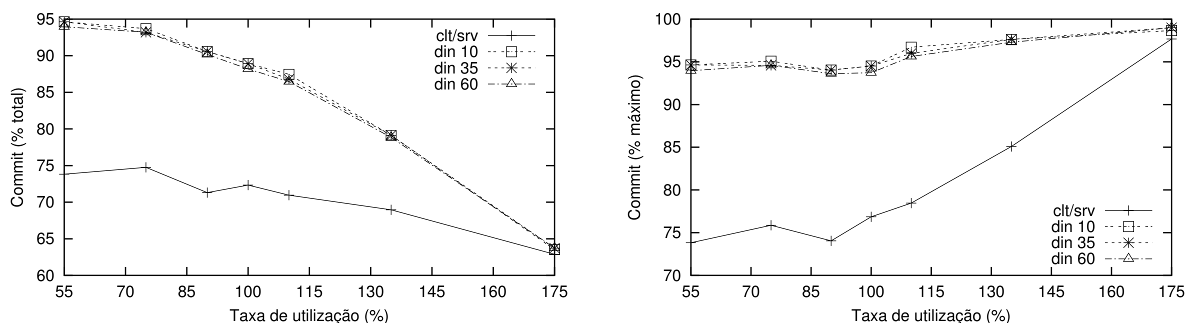

Figura 9.14: Transacções aceites imediatamente no cliente ou no servidor quando os clientes não obtém inicialmente nenhuma reserva (cenário *MOV:PEQ:MAU*): valor relativo ao máximo possível (direita).

reserva (esta situação pode ser usada igualmente nos cenários em que não existem previsões). Assim, um cliente apenas obtém reservas após receber a primeira encomenda.

A figura 9.12 apresenta o valor das transacções que podem ser confirmadas com sucesso nos clientes de forma independente em função da taxa de utilização, quando os clientes não obtém reservas no início da simulação. Os resultados mostram que, neste caso, é possível confirmar entre 80% e 90% das encomendas recebidas. Como o sistema se adapta à procura dinamicamente, é impossível confirmar algumas das primeiras encomendas recebidas em cada cliente. Este facto explica o aumento da fracção de transacções confirmadas localmente com o aumento da taxa de utilização (e, em consequência, do valor das transacções).

Os resultados obtidos quando a existência inicial é maior (cenário *GRD*), apresentados na figura 9.13, corroboram a explicação anterior. Adicionalmente, estes resultados mostram que a aproximação adoptada permite que o sistema de reservas se adapte às necessidades de cada cliente, permitindo confirmar localmente mais de 90% das transacções possíveis (para taxas de utilização superiores a 100%, este valor sobe para valores superiores a 97%).

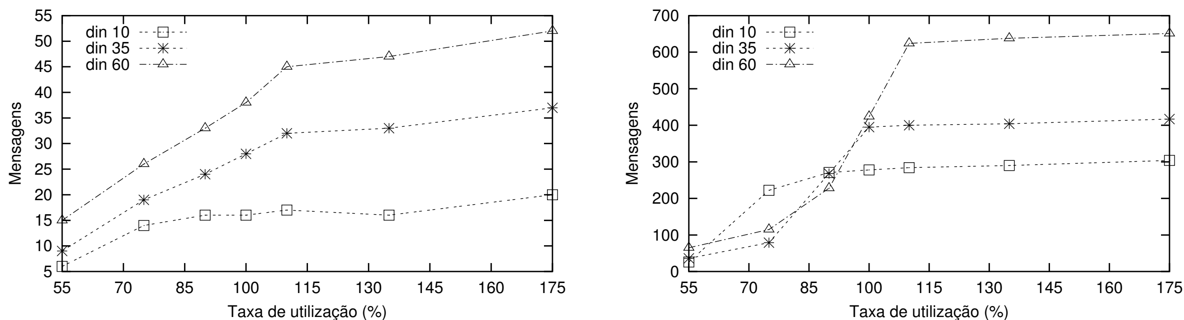

Figura 9.15: Mensagens enviadas na obtenção de novas reservas quando o cliente obtém inicialmente o máximo de reservas possíveis (cenário $MOV : PEQ : MAU$ à esquerda e $MOV : GRD : MAU$ à direita).

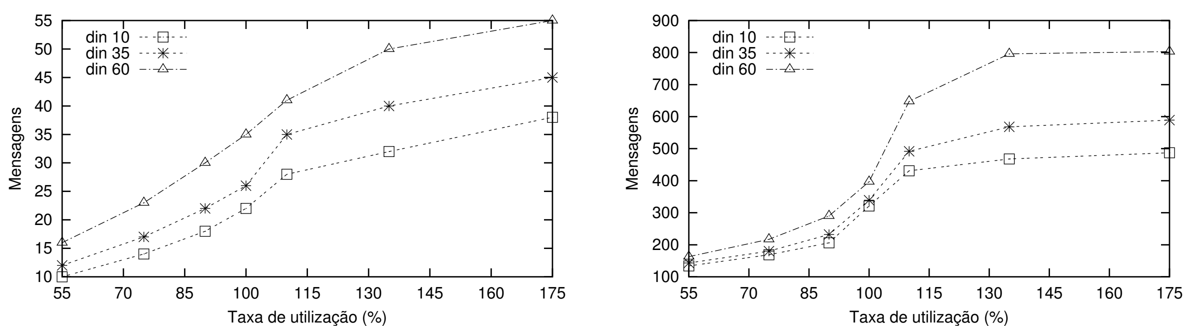

Figura 9.16: Mensagens enviadas na obtenção de novas reservas quando o cliente não obtém reservas no início da simulação (cenário $MOV : PEQ : MAU$ à esquerda e $MOV : GRD : MAU$ à direita).

Confirmação imediata dos resultados não obtendo reservas inicialmente A figura 9.14 mostra as transacções que podem ser imediatamente confirmadas no cliente ou no servidor. Como se esperava, os resultados obtidos são melhores do que quando o cliente obtém inicialmente reservas. Neste caso, é possível confirmar imediatamente o sucesso de mais de 95% das encomendas recebidas.

Mensagens de obtenção de novas reservas Na figura 9.15 apresenta-se o número de mensagens propagadas na obtenção de novas reservas quando o cliente obtém o máximo de reservas inicialmente. Como anteriormente, este valor inclui todas as tentativas de propagação, pelo que o envio de um pedido e a recepção da respectiva resposta é contabilizado como duas mensagens. Como se esperava, os resultados obtidos são muito semelhantes aos apresentados quando a previsão das reservas necessárias é fiável porque este valor está directamente relacionado com a existência disponível para a obtenção de novas reservas.

Na figura 9.16 apresenta-se o número de mensagens propagadas quando o cliente não obtém reservas inicialmente. Os resultados obtidos mostram que o número de mensagens enviadas tende a ser ligeiramente superior à situação anterior (em que os clientes obtém reservas no início da simulação com base na previsão das necessidades). No entanto, mesmo nesta situação, os custos de comunicações são pequenos.

Por exemplo, no cenário *PEQ*, com a estratégia de obtenção de reservas *din 60* (respectivamente *din 10*) e uma taxa de utilização de 110% (respectivamente 90%), os clientes obtém reservas a partir do servidor uma vez por cada $7,3$ (respectivamente $12,7$) encomendas recebidas no cliente. Estes valores são muito superiores no cenário *GRD*.

Os resultados apresentados mostram que o sistema de reservas permite suportar uma aplicação de suporte a uma força de vendas móvel quando é possível estimar a procura em cada cliente e mesmo quando não é possível fazer esta previsão. Neste caso, é necessário usar estratégias dinâmicas de obtenção de reservas. As estratégias usadas nestas experiências apresentam uma boa adaptação às necessidades dos clientes com custos de comunicação aceitáveis.

Capítulo 10

Sistema de reconciliação SqlIceCube

Neste capítulo descreve-se o sistema SqlIceCube [133]. O SqlIceCube é um sistema genérico de reconciliação para transacções móveis. Este sistema explora a semântica das operações para criar uma sequência de execução que permita maximizar o número (ou valor) de transacções móveis que podem ser executadas. A informação semântica necessária ao funcionamento do sistema é extraída automaticamente do código das transacções móveis.

No sistema Mobisnap, o SqlIceCube é usado para executar um conjunto de transacções móveis nas seguintes situações: (1) quando se executam as transacções móveis não garantidas recebidas de um cliente; (2) quando se executa o conjunto de transacções que aguardam reexecução após o cancelamento ou expiração de uma reserva.

O sistema SqlIceCube é um sistema genérico de reconciliação que estende o sistema IceCube apresentado em [134]. Como um sistema genérico, inclui algumas funcionalidades que não são exploradas no sistema Mobisnap. No entanto, de forma a apresentar o sistema de forma completa, descrevem-se todas as características do sistema SqlIceCube neste capítulo.

10.1 Modelo geral

O sistema SqlIceCube é um sistema de reconciliação que aborda o processo de reconciliação como um problema de optimização, aproximação da qual foi pioneiro o sistema IceCube [85]. Assim, este sistema tenta criar a sequência de execução que permite optimizar o conjunto de transacções móveis que podem ser executadas com sucesso combinando os conjuntos de transacções móveis executadas concorrentemente.

Considere-se o exemplo da figura 10.1, no qual dois utilizadores modificaram concorrentemente uma agenda partilhada. Um utilizador requisita a sala A às 9:00 e também a sala B ou C igualmente às 9:00. Outro utilizador requisita a sala A ou B às 9:00. Se as operações do primeiro utilizador fossem executadas

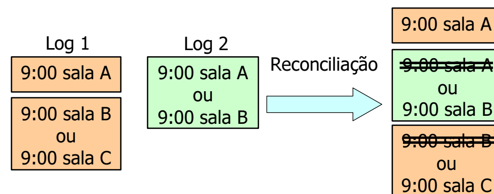

Figura 10.1: Ordenação de um conjunto de operações executadas concorrentemente.

antes da operação do segundo utilizador, a sala A e B seriam reservadas para o primeiro utilizador e a operação do segundo utilizador falharia (considerando que no caso de haver múltiplas alternativas se escolhia a primeira possível). Uma situação semelhante acontece se a operação do segundo utilizador for executada em primeiro lugar. A primeira operação do primeiro utilizador falha porque a sala A já está reservada. No entanto, a ordem apresentada na figura permite a execução de todas as operações.

O exemplo anterior mostra que, em algumas situações, é possível obter um melhor resultado de reconciliação através da reordenação das operações mesmo quando as operações apresentam regras de resolução de conflitos. Neste exemplo, não seria possível explorar as alternativas para obter o melhor resultado da reconciliação sem reordenar as operações. A generalidade dos sistemas de reconciliação genéricos (por exemplo, no sistema Bayou [161] e Deno [82]) não reordenam as operações.

Como, para qualquer conjunto não trivial de transacções, é impossível testar todas as sequências de execução possíveis, o sistema executa uma pesquisa heurística dirigida usando a informação semântica sobre as transacções a reconciliar. Esta informação é definida sobre a forma de relações que condicionam a ordenação e exequibilidade das operações. O sistema extrai esta informação automaticamente a partir do código das transacções móveis.

O SqlIceCube é uma extensão do sistema IceCube apresentado em [134]. Relativamente a esse sistema foram introduzidas duas mudanças fundamentais. Primeiro, ao contrário do sistema IceCube, a reconciliação é vista como um problema de planeamento em vez de um problema de resolução de restrições (binárias). Esta aproximação permite a execução de duas transacções incompatíveis entre si, desde que, depois de executar a primeira transacção, se execute uma nova transacção que torne possível a segunda transacção. Segundo, o sistema SqlIceCube inclui um mecanismo de extracção automática de relações entre as transacções móveis. Desta forma, os programadores não necessitam de expor esta informação explicitamente. Finalmente, foram introduzidas algumas alterações para adaptar os algoritmos de reconciliação ao modelo de execução das transacções (mais simples) e às (diferentes) relações semânticas extraídas.

10.2 Relações estáticas

O sistema SqlIceCube usa informação semântica expressa como relações entre pares de transacções móveis. As relações definidas têm uma natureza estática, sendo válidas independentemente do estado da base de dados. Estas relações podem codificar a semântica das aplicações (relações da aplicação) ou a semântica dos dados (relações dos dados).

10.2.1 Relações da aplicação

Uma *relação da aplicação* (*log contraints*) define uma dependência estática entre duas transacções executadas num mesmo fluxo de actividade, i.e., numa mesma sessão de uma aplicação. Estas relações são independentes da semântica de cada transacção móvel e exprimem as intenções dos utilizadores, codificando a semântica das macro-operações definidas por um conjunto de transacções móveis.

Foram definidas as seguintes relações da aplicação:

Predecessor/sucessor Estabelece que o sucessor apenas pode ser executado após a execução com sucesso do predecessor. Esta relação é útil quando o predecessor produz algum efeito usado pelo sucessor.

Predecessor/sucessor fraco Estabelece que, se as duas transacções forem executadas com sucesso, o predecessor deve ser executado antes do sucessor. Esta relação permite estabelecer a ordem de execução entre duas acções.

Alternativas Define um conjunto de transacções das quais apenas uma deve ser executada. A especificação de alternativas é um mecanismo simples de resolução de conflitos.

Grupo indivisível Define um conjunto de transacções, as quais devem ser todas ou nenhuma executada com sucesso.

Em geral, estas relações, por exprimirem as intenções dos utilizadores, são expressas explicitamente pela aplicação ao submeter as transacções móveis. No entanto, como se discute na secção 10.5.4.3, o sistema de extracção automática de relações consegue inferir algumas destas relações a partir do código das transacções. Por exemplo, uma transacção que especifique várias opções como uma sequência de instruções *if* pode ser dividida num conjunto ordenado de transacções alternativas.

No protótipo do sistema Mobisnap, apenas se usam as relações da aplicação inferidas automaticamente. No entanto, seria simples introduzir uma nova operação na API do sistema que permitisse às aplicações adicionar relações da aplicação entre as várias transacções móveis submetidas no cliente.

10.2.2 Relações dos dados

As *relações dos dados* (*data contraints*) exprimem relações, entre duas transacções, que reflectem a semântica das operações e dos dados, independentemente do estado da base de dados.

Foram definidas as seguintes relações:

Comutativas Indica que duas transacções são comutativas, i.e., que o resultado da sua execução é idêntico independentemente da ordem de execução (se executadas consecutivamente). Duas transacções t_1 e t_2 são comutativas sse $\forall s \in \mathcal{S}, t_1(t_2(s)) = t_2(t_1(s))$, com $\mathcal{S}$ o conjunto de todos os estados possíveis da base de dados, e $t(s)$ o estado da base de dados que resulta de executar a transacção t no estado s.

Impossibilita Indica que uma transacção torna impossível a execução (sucessiva) de outra transacção. t_1 impossibilita t_2 sse $\forall s \in \mathcal{S}, \neg valido(t_2, t_1(s))$, com $valido(t, s)$ verdadeiro sse é possível executar com sucesso a transacção t no estado s. Por exemplo, uma transacção que reserve um dado lugar num comboio impossibilita a execução com sucesso de outra transacção que reserve o mesmo lugar.

Torna possível Indica que uma transacção torna possível a execução (sucessiva) de outra transacção. t_1 torna possível t_2 sse $\forall s \in \mathcal{S}, valido(t_2, t_1(s))$. Por exemplo, uma transacção que cancele uma reserva de um dado lugar num comboio possibilita a execução com sucesso de outra transacção que reserve o mesmo lugar.

Desfavorece Indica que uma transacção desfavorece a execução (sucessiva) de outra transacção. t_1 desfavorece t_2 sse $\exists s \in \mathcal{S} : valido(t_2, s) \wedge \neg valido(t_2, t_1(s))$. Por exemplo, uma transacção que aumente o preço de um produto desfavorece a possibilidade de se executar uma transacção cujo preço a usar tenha um limite máximo.

Favorece Indica que uma transacção favorece a execução (sucessiva) de outra transacção. t_1 favorece t_2 sse $\exists s \in \mathcal{S}, \neg valido(t_2, s) \wedge valido(t_2, t_1(s))$. Por exemplo, uma transacção que reduza o preço de um produto favorece a possibilidade de se executar uma transacção cujo preço a usar tenha um limite máximo.

Estas relações são extraídas automaticamente a partir do código das transacções móveis como se explica na secção 10.5.

10.2.3 Transacções independentes

Com base nas relações anteriores, definiu-se a relação de independência entre duas transacções. Duas transacções dizem-se independentes se a execução de uma transacção não puder influenciar a execução

de outra transacção. Se duas transacções forem independentes, a ordem pela qual são executadas não influencia o resultado final. Face às relações definidas anteriormente, as transacções a e b são independentes sse:

$$\begin{aligned} independentes(a,b) = & \neg predecessor/sucessor(a,b) \wedge \neg predecessor/sucessor(b,a) \wedge \\ & \neg predecessor/sucessorfraco(a,b) \wedge \neg predecessor/sucessorfraco(b,a) \wedge \\ & \wedge comutativas(a,b) \end{aligned}$$

10.3 Algoritmo de reconciliação

O algoritmo de reconciliação define a ordem de execução das transacções na base de dados. O espaço de soluções possíveis é explorado por amostragem de forma heurística. Para tal, sucessivas soluções são criadas e executadas até que uma solução satisfatória seja encontrada. Cada solução gerada é independente das anteriores, i.e., não se usa uma aproximação de optimização local.

Uma solução inclui apenas as transacções executadas com sucesso. As restantes transacções podem ter sido excluídas da solução por não poderem ser executadas no estado final da base de dados ou por violarem uma relação estática (por exemplo, pertencer a um conjunto de alternativas, do qual foi executada uma transacção).

A cada transacção pode ser atribuído um valor numérico que representa a sua importância no contexto da aplicação (no sistema Mobisnap, assume-se que todas as transacções tem valor igual a um). A soma dos valores numéricos das transacções incluídas numa sequência representa o valor da solução — o sistema SqlIceCube tenta criar uma sequência de transacções que maximize este valor.

Para melhorar o desempenho do sistema de reconciliação decompõe-se a criação da sequência de execução em vários subproblemas. Cada subproblema consiste em criar uma parte da sequência a partir de um subconjunto de transacções independentes do resto das transacções. Nesta secção detalham-se os algoritmos usados no sistema de reconciliação.

10.3.1 Heurística

Uma solução (sequência de execução) é criada incrementalmente seleccionando em cada passo uma transacção para execução. Esta selecção é efectuada, aleatoriamente, entre as transacções com mérito mais elevado.

O mérito associado a uma transacção é estimado, de forma heurística, a partir das relações estáticas que se estabelecem entre as transacções. Nesta estimativa, apenas são consideradas as transacções que podem ser executadas. O mérito de uma transacção que não pode ser executada imediatamente por as relações da aplicação estabelecidas não estarem satisfeitas (por exemplo, uma transacção que é sucessora

de outra transacção ainda não executada) tem mérito nulo.

O mérito de uma transacção estima o benefício de adicionar essa transacção à solução parcialmente construída. Este mérito deve ser tanto menor quanto maior for o valor das transacções que não podem ser executadas no futuro devido à execução desta transacção. Mais precisamente, o mérito de uma transacção t é tanto maior quanto (por ordem decrescente de importância e considerando apenas as transacções que ainda podem ser executadas com sucesso):

1. Maior for o número de transacções já executadas que pertencem ao mesmo grupo indivisível (porque a execução de apenas um subconjunto das transacção de um grupo indivisível leva a uma sequência de execução inválida).

2. Menor for o valor das transacções que apenas podem ser executadas antes de t (porque todas as transacções predecessores têm de ser abortadas) – relação predecessor/sucessor fraco.

3. Menor for o valor das transacções que são alternativas a t (porque dados dois conjuntos de transacções alternativas entre si, é mais provável conseguir executar uma transacção do conjunto com mais alternativas) – relação alternativas.

4. Menor for o valor das transacções que t torna impossíveis (porque essas transacções não podem ser executadas, pelo menos, enquanto não for executada uma transacção que torne a sua execução possível outra vez) – relação impossibilita. Maior for o valor das transacções que t torna possíveis – relação torna possível.

5. Maior for o valor das transacções que são sucessoras (fracas ou fortes) de t (porque a execução do predecessor torna possível a execução do sucessor) – relação predecessor/sucessor fraco e predecessor/sucessor.

6. Menor for o valor das transacções que t desfavorece (porque menor será a probabilidade de executar essa transacções com sucesso) – relação favorece. Maior for o valor das transacções que t favorece – relação desfavorece.

Independentemente das regras anteriores, o mérito de uma transacção é infinitamente baixo se: (1) for sucessora (forte) de uma transacção ainda não executada (relação predecessor/sucessor); (2) for predecessora (fraca) de uma transacção já executada (relação predecessor/sucessor fraco); (3) for alternativa de uma transacção já executada.

10.3.2 Algoritmo básico

O algoritmo de reconciliação implementado é apresentado nas figuras 10.3 e 10.2. A função criaUmaSolução cria uma solução e a função reconcilia implementa o algoritmo de reconciliação completo.

```
reconcilia ( bd, trxs) =
  melhorValor := 0
  melhorSolução := []
  sumário := computaSumário( trxs)                  // Cria informação auxiliar
  DO
    bd.initTrx()                                    // Prepara criação de nova solução
    { solução, valor } := criaUmaSolução (bd, sumário, trxs)
    IF melhorSolução( bd, solução, valor) THEN      // Verifica se é a melhor solução
      melhorSolução := solução                      // Guarda a melhor solução
      melhorValor := valor
    END IF
    bd.rollbackTrx()
  UNTIL termina ( melhorSolução, melhorValor)
  bd.initTrx()                                      // Repõe estado da melhor solução
  bd.executa( melhorSolução)
  bd.commitTrx()
  RETURN { melhorSolução, melhorValor }
```

Figura 10.2: Algoritmo de reconciliação (sem partições)

Inicialmente, a função reconcilia calcula um sumário das relações definidas entre as transacções (computaSumário). Este sumário é usado para estimar o mérito de cada transacção eficientemente. O algoritmo de reconciliação prossegue com a criação de sucessivas soluções (criaUmaSolução) de forma provisória. A melhor solução é guardada. Os critério que definem se uma dada solução é a melhor até ao momento (melhorSolução) e se o algoritmo deve terminar (termina) podem ser definidos de forma específica em cada sistema. Por omissão, o valor da solução é usado para determinar se uma solução é a melhor até ao momento, e o algoritmo executa durante um período pré-definido de tempo (ou até todas as transacções poderem ser executadas com sucesso). Quando o algoritmo termina, a melhor solução encontrada é executada de forma definitiva na base de dados[1].

A função criaUmaSolução cria uma solução e executa-a na base de dados de forma incremental. Durante a execução do algoritmo, as transacções a reconciliar são agrupadas nos seguintes subconjuntos: (1) transacções já executadas com sucesso; (2) transacções impossíveis de executar devido à execução de uma transacção já incluída na solução (relações alternativas e predecessor/sucessor fraco); (3) transacções impossíveis de executar temporariamente (relação impossibilita e transacções que abortaram aquando da execução); (4) transacções candidatas à execução (todas as transacções não incluídas em nenhum dos outros conjuntos). Esta informação é mantida juntamente com o sumário das relações existentes entre as transacções.

O ciclo principal consiste na sucessiva selecção e execução de uma nova transacção [2]. Esta transacção é seleccionada de entre as transacções candidatas usando a heurística apresentada anteriormente.

---

[1] A seguinte optimização foi implementada: se o ciclo de pesquisa de novas soluções terminar imediatamente após ser criada a melhor solução, esta solução não chega a ser desfeita.

[2] Para permitir desfazer não só uma transacção, mas também uma sequência de transacções, é necessário que o sistema de bases de dados suporte um mecanismo de transacções encaixadas (*nested transactions*) ou de pontos de gravação intermédios no âmbito de uma transacção (*savepoints*), como o existente na base de dados Oracle 8 usada no servidor do sistema Mobisnap.

```
criaUmaSolução ( bd, sumário, trxs) =   // pseudo-código
    solução := []
    valor := 0
    REPEAT
      sumário.reinicia( trxs)
      WHILE sumário.algumaPossível() DO            // Cria solução incrementalmente
        próxTrx := seleccionaPorMérito ( sumário)   // Selecciona acção a executar
        dynOK := bd.executaInnerTrx( próxTrx)
        IF dynOK = FALSE THEN                       // Resultado: rollback
          sumário.excluiTmp( próxTrx)
        ELSE                                        // Resultado: commit
          solução := [solução|próxTrx]              // Actualiza solução
          valor := valor + próxTrx.valor
          sumário.executada( próxTrx)
          aExcluir := incompatíveis( próxTrx, trxs) // Actualiza informação sobre transacções
          sumário.exclui( aExcluir)
          aExcluirTmp := tornaImp( próxTrx)
          sumário.excluiTmp( aExcluirTmp)
          aIncluirTmp := tornaPosAjuda( próxTrx)
          sumário.incluiTmp( aIncluirTmp)
        END IF
      END WHILE
      gtrxs = gruposIncompletos(trxs,solução)       // Transacções pertencentes a grupos indivísiveis
      trxs = trxs \ gtrxs                           // parcialmente executados
      IF( NOT vazio(gtrxs)) THEN
        bd.rollbackTrx()
        bd.initTrx()
      END IF
    UNTIL vazio(gtrxs)
    RETURN { solução, valor}
```

Figura 10.3: Algoritmo de criação de uma única solução.

Se a execução da transacção seleccionada aborta, a transacção passa a ser considerada temporariamente impossível de executar. Se uma transacção é executada com sucesso, ela é adicionada à solução parcialmente construída. A informação sobre as transacções a executar é actualizada. As transacções que são incompatíveis com a execução dessa transacção (relações alternativas e predecessor/sucessor fraco) são adicionadas ao conjunto das transacções impossíveis (assim como todas as transacções que pertençam a um grupo indivisível com uma das transacções impossíveis). As transacções que são impossibilitadas (relação impossibilita) são adicionadas ao conjunto das transacções temporariamente impossíveis. As transacções que são beneficiadas (relações torna possível e favorece) são removidas do conjunto das transacções temporariamente impossíveis. Em ambos os casos, o ciclo itera enquanto existem transacções candidatas com mérito não nulo.

Quando se conclui a criação de uma solução é necessário verificar se a solução é válida face às relações estabelecidas[3]. Para tal, é necessário verificar se existe algum grupo indivisível parcialmente executado (todas as outras relações são necessariamente satisfeitas porque nunca se selecciona uma transacção cuja execução viole uma relação estática com uma transacção anteriormente executada). Em

[3]Uma optimização implementada no protótipo do sistema consiste na verificação, sempre que uma transacção é declarada impossível, da impossibilidade de executar completamente qualquer grupo indivisível com transacções já executadas. Neste caso, a solução que se está a criar é declarada imediatamente inválida.

```
reconciliaComPartições ( bd, trxs) =
   partições := particiona( trxs)           // Divide em partições
   solução := []
   valor := 0
   FOR EACH part IN partições               // Obtém solução para cada partição
      { soluçãoParcial, valorParcial } := reconcilia ( bd, part.trxs)
      solução := [ solução | soluçãoParcial ]
      valor := valor + valorParcial
   END FOR
   RETURN { solução, valor }
```

Figura 10.4: Algoritmo de reconciliação (com partições).

caso afirmativo, a solução é inválida e é necessário criar uma nova solução. Para evitar cair numa situação semelhante na próxima solução criada, o grupo de transacções parcialmente executado é excluído (temporariamente) do conjunto de transacções a reconciliar. Apesar desta aproximação poder levar a uma solução sub-óptima, garante que é sempre possível obter uma solução (porque se vão excluindo os grupos de transacções que causam problemas).

Durante a execução do algoritmo é criada uma explicação da razão pela qual cada transacção não foi incluída na solução final (omitido nos algoritmos apresentados). Esta informação pode ser usada pelas funções melhorSolução e termina ou para fornecer informação aos utilizadores.

10.3.3 Complexidade

A complexidade temporal esperada da função criaUmaSolução é $O(n^2)$, com n o número de transacções a reconciliar. Intuitivamente, o ciclo principal é executado $O(n)$ vezes ($O(n)$ transacções são seleccionadas para execução). A complexidade do ciclo principal é dominado pela função de selecção (seleccionaPorMérito) que analisa o mérito de todas as transacções candidatas. Esta função executa em $O(n)$ porque o mérito de cada transacção é avaliado em tempo constante usando o sumário das relações entre as transacções. O tempo esperado de actualização do sumário é constante (no caso esperado, a execução de uma transacção influencia apenas um pequeno número de transacções). No pior caso, estas actualizações executam igualmente em $O(n)$.

A complexidade de criação do sumário (computaSumário) é $O(n^2)$ porque cada par de transacções é comparado para obter as relações existentes entre elas. O ciclo de criação de soluções é igualmente executado em $O(n^2)$ (no caso esperado, apenas um número reduzido de soluções são criadas). Assim, a complexidade total (esperada) do algoritmo de reconciliação (reconcilia) é $O(n^2)$.

Através da utilização de uma estrutura de dados especialmente adaptada à manutenção do mérito de cada transacção (baseado na utilização de filas de prioridade - *priority queues*) é possível reduzir a complexidade da função criaUmaSolução para $O(n.logn)$. No entanto, a complexidade global mantém-se em $O(n^2)$ devido à função computaSumário.

10.4 Optimização da reconciliação

Nesta secção apresentam-se dois mecanismos que permitem melhorar o desempenho do sistema de reconciliação desenvolvido. Como a complexidade do algoritmo de reconciliação é $O(n^2)$, é possível melhorar o desempenho dividindo um problema de reconciliação numa sequência de subproblemas de menores dimensões (secção 10.4.1) e diminuindo o número de elementos envolvidos na reconciliação (secção 10.4.2). Estes mecanismos foram propostos anteriormente no âmbito do sistema IceCube [134].

10.4.1 Partições

Usando a noção de independência de transacções, definida anteriormente, é possível dividir o problema da reconciliação num conjunto de subproblemas. Para tal, divide-se o conjunto de transacções a reconciliar em subconjuntos disjuntos mutuamente independentes, a que se chamam partições. O resultado da reconciliação do conjunto original de transacções é equivalente ao resultado da reconciliação sucessiva de cada uma das partições.

Na figura 10.4 apresenta-se o algoritmo de reconciliação usando partições. A reconciliação é executada incrementalmente, invocando sequencialmente o algoritmo de reconciliação básico (reconcilia) para cada partição. A função particiona divide o conjunto de transacções original, $trxs$, em k partições de acordo com a seguinte relação:

$$trxs = p_1 \cup p_2 \cup \ldots \cup p_k : \forall i, j, i \neq j \Rightarrow p_i \cap p_j = \emptyset \wedge$$
$$\forall a \in p_i, b \in p_j, independentes(a,b) \wedge \neg alternativas(a,b) \wedge \neg grupo_indivisivel(a,b)$$

O ciclo que executa a reconciliação sucessiva das várias partições tem complexidade $O(|p_1|^2 + \ldots + |p_k|^2)$. Quando as partições têm dimensões idênticas, o tempo de execução do ciclo tende a crescer linearmente com o número de partições. Quando uma partição é muito maior do que as outras, o tempo de execução do ciclo tende a ser dominado pela maior partição.

No entanto, a complexidade do algoritmo de reconciliação é igualmente influenciada pela algoritmo de divisão em partições. Um algoritmo que use a relação anterior de forma imediata tem complexidade $O(n^2)$ porque uma transacção tem de ser comparada com todas as outras. Assim, para melhorar o desempenho, efectua-se inicialmente uma divisão imprecisa usando um algoritmo de complexidade esperada $O(n)$. Neste algoritmo, cada transacção define um conjunto arbitrário de identificadores (por exemplo, os nomes das tabela acedidas e/ou identificadores para as condições dos registos acedidos). As transacções com identificadores idênticos são incluídas na mesma partição. Para cada uma das partições é então aplicado o algoritmo de partição exacto. A complexidade global é então $O(n + |pi_1|^2 + \ldots + |pi_i|^2)$, com $pi_1, \ldots, pi_i$ as partições criadas pelo algoritmo de divisão imprecisa.

Note-se que, em muitos casos, é possível executar a divisão em partições de forma incremental e antes do momento da reconciliação (o que permite diminuir o tempo de execução do algoritmo de reconciliação). No sistema Mobisnap, a divisão em partições pode ser efectuada: nos clientes, para o conjunto de transacções não garantidas a propagar para o servidor; no servidor, para o conjunto de transacções a reexecutar quando uma reserva expira. Esta divisão pode ser efectuada incrementalmente quando uma nova transacção é adicionada aos conjuntos anteriores.

10.4.2 Compressão de uma sequência de transacções

Duas sequências de transacções dizem-se equivalentes se, quando executadas num qualquer estado da base de dados, produzem o mesmo efeito (i.e., $t_0, t_1, \ldots, t_n \equiv s_0, s_1, \ldots, s_p \Rightarrow \forall db, t_n(\ldots(t_1(t_0(db)))) \equiv s_p(\ldots(s_1(s_0(db)))))$. Um mecanismo de compressão de um registo de transacções transforma uma sequência de transacções numa sequência de transacções equivalente com menor número de elementos (eliminando as transacções cuja execução não produz efeitos no estado final da base de dados).

De seguida descreve-se o mecanismo genérico de compressão de uma sequência de transacções (ou operações). Este mecanismo é usado, no sistema SqlIceCube, para comprimir a sequência de transacções a reconciliar submetida por cada utilizador, e no sistema DOORS, para comprimir a sequência de invocações executadas num *coobjecto* por uma aplicação. Este mecanismo usa as seguintes propriedades, definidas entre pares de transacções, a e b:

- *b absorve a* sse a execução de *b esconde* os efeitos da execução de a, i.e., $\forall s \in \mathcal{S}, b(a(s)) = b(s)$. Neste caso, $[a,b] \equiv [b]$.
- *b compensa a* sse os efeitos da execução de a são compensados pela execução de b, i.e., $\forall s \in \mathcal{S}, b(a(s)) = s$. Neste caso, $[a,b] \equiv []$.
- *c comprime* $[a,b]$ sse a execução de c comprime os efeitos da execução sucessiva de $[a,b]$, i.e., $\forall s \in \mathcal{S}, b(a(s)) = c(s)$. Neste caso, $[a,b] \equiv [c]$ (esta propriedade não é usada no sistema SqlIceCube).

Estas propriedades podem ser usadas para comprimir apenas pares de transacções que possam ser executadas consecutivamente (logo, não se podem comprimir duas transacções alternativas entre si). Caso contrário, o resultado de uma transacção que deva ser executada entre ambas pode ser afectado. Verifica-se a possibilidade de executar duas transacções consecutivamente usando as seguintes regras de reordenação[4]:

[4] O mecanismo de compressão de sequências de operações implementado no sistema DOORS utiliza adicionalmente a função de transformação de operações [159], transforma definida pelos programadores, tal que, $transforma([a,b]) = [b^a, a^b] : \forall s, b(a(s)) \equiv a^b(b^a(s))$. Neste caso $[a,b] \equiv [b^a, a^b]$.

- $[a,t_0,t_1,\ldots,t_n,b] \equiv [a,b,t_0,t_1,\ldots,t_n]$, sse $independentes(t_0,b) \wedge \ldots \wedge independentes(t_n,b)$;
- $[a,t_0,t_1,\ldots,t_n,b] \equiv [t_0,t_1,\ldots,t_n,a,b]$, sse $independentes(a,t_0) \wedge \ldots \wedge independentes(a,t_n)$.

Adicionalmente, é necessário garantir que as intenções dos utilizadores expressas através de relações da aplicação estabelecidas entre as transacções são respeitadas. Para tal, aplicam-se as seguintes regras sempre que se comprime um par de transacções a e b usando qualquer das regras especificadas:

- Se a pertence a um conjunto de transacções alternativas $\mathcal{A}$ (relação alternativas), todas as transacções $\mathcal{A} \setminus \{a\}$ são removidas da sequência produzida. A mesma regra é aplicada a b.
- Se a pertence a um grupo indivisível, a transacção que resulta da compressão também deve pertencer a esse grupo indivisível (se o resultado for a sequência vazia, todas as transacções do grupo indivisível devem ser executadas com sucesso). A mesma regra é aplicada a b.
- Se existe uma relação predecessor/sucessor fraco ou predecessor/sucessor entre a e b, essa relação desaparece.
- Se a tem um predecessor (sucessor) t (relação predecessor/sucessor fraco ou predecessor/sucessor), a mesma relação deve ser estabelecida entre a transacção resultado da compressão e t. A mesma regra é aplicada a b.

A sequência de transacções a propagar para o servidor, armazenada num cliente, é comprimida incrementalmente. Assim, o cliente verifica imediatamente se pode comprimir cada nova transacção[5]

10.5 Extracção automática de relações

A extracção automática de relações entre as transacções móveis é executada a partir da análise estática dos seus programas. Este processo consiste em dois passos. Primeiro, para cada transacção, extrai-se a informação sobre os dados lidos, os dados escritos, e as pré-condições que levam ao sucesso da transacção. Segundo, as relações entre duas transacções são inferidas comparando a informação obtida anteriormente. Nesta secção detalha-se cada um destes processos.

A análise estática de programas [49] foi usada em vários sistemas com várias finalidades, entre as quais se destacam a verificação da correcção [48, 69, 10] e equivalência de programas [92]. No entanto, que o autor saiba, este trabalho é o primeiro a utilizar uma aproximação similar na reconciliação de dados. A análise estática executada no sistema SqlIceCube apresenta algumas características pouco comuns: é executada durante o funcionamento do sistema e a informação extraída dos programas é usada para inferir relações diferentes da equivalência entre dois programas.

[5]No âmbito do sistema Mobisnap, apenas as transacções não garantidas localmente são comprimidas.

10.5.1 Extracção de informação

A extracção de informação associada a uma transacção móvel é executada a partir do seu programa, analisando estaticamente os vários caminhos de execução possíveis. Para cada caminho de execução que termina numa instrução *commit*, extrai-se a seguinte informação:

Conjunto de escrita semântico Constituído pela descrição semântica de todas as modificações produzidas pela transacção.

Conjunto de leitura semântico Constituído pela descrição semântica de todos os elementos de dados lidos e usados pela transacção (os elementos de dados lidos e não usados são ignorados).

Conjunto de pré-condições Constituído por todas as condições usadas para determinar o caminho de execução.

Durante o processo de análise estática, as seguintes instruções são processadas de forma especial:

Instrução *select C into X from T where Cond* Atribui ao valor da variável *X* o valor especial *read(T,Cond,C)* que descreve semanticamente os dados lidos. Este valor não pode ser obtido de forma estática porque depende do estado da base de dados no momento em que a transacção móvel é executada.

Instrução *update T set C = Val where Cond* Adiciona ao conjunto de escrita a seguinte descrição semântica associada à instrução de actualização: *update(T,Cond,C,Val)*.

Instrução *insert into T[Cols] values Vals* Adiciona ao conjunto de escrita a seguinte descrição semântica associada à instrução de inserção: *insert(T,Cols,Vals)*.

Instrução *delete from T where Cond* Adiciona ao conjunto de escrita a seguinte descrição semântica associada à instrução de remoção: *delete(T,Cond)*.

Instrução *if(Cond) then ... else ...* No caso de o valor de *Cond* ser conhecido nenhuma acção especial é executada. No caso de o valor de *Cond* ser desconhecido serão analisados os dois possíveis caminhos de execução, assumindo num caso que a condição é verdadeira e adicionando o valor *precond(Cond)* ao conjunto de pré-condições e no outro que a condição é falsa e adicionando *precond(not Cond)* ao conjunto de pré-condições.

Durante a análise estática, qualquer expressão, *e*, cujo valor dependa de uma variável com um valor obtido numa instrução de leitura, tem igualmente um valor especial, *expr(e)*. Este valor pode ser atribuído a uma variável (numa instrução de atribuição) ou usado em qualquer das instruções de acesso à base de dados (apresentadas anteriormente).

O conjunto de leitura associado a um caminho de execução é obtido a partir dos elementos dos conjuntos de escrita e do conjunto de pré-condições, obtendo todos os valores especiais *read(...)* usados na descrição semântica desses elementos.

Associado a cada caminho de execução que termina com sucesso obtém-se um grupo de informação semântica composto pelo conjunto de escrita, de leitura e de pré-condições. Relativamente aos caminhos de execução que terminam numa instrução *rollback* não se obtém qualquer informação. Assim, associada a cada transacção podem existir vários grupos de informação semântica.

Exemplo: Considere-se o seguinte programa:

```
1  BEGIN
2      IF c1 THEN
3          IF c2 THEN
4              COMMIT;
5          ELSE
6              COMMIT;
7          END IF;
8      ELSEIF c3 THEN
9          ROLLBACK;
10     END IF;
11 END
```

Este programa tem quatro caminhos de execução possíveis. O primeiro termina na instrução *commit* da linha 4. Neste caso, o conjunto de pré-condições inclui os valores *precond(c1)* e *precond(c2)*. O segundo termina na instrução *commit* da linha 6 e o respectivo conjunto de pré-condições inclui os valores *precond(c1)* e *precond(not c2)*. O terceiro termina na instrução *rollback* da linha 9. Assim, não se extrai informação semântica deste caminho de execução. O quarto caminho de execução termina no fim do programa e o conjunto de pré-condições inclui os valores *precond(not c1)* e *precond(not c3)*.

Desta forma, associado e esta transacção existem três grupos de informação semântica. Para que seja possível executar a transacção num dado estado da base de dados é apenas necessário que todos os valores de um dos conjuntos de pré-condições seja verdadeiro nesse estado. ∎

Na figura 10.5 apresenta-se uma transacção móvel que introduz uma nova encomenda (semelhante à transacção apresentada na figura 8.1, mas usando identificadores numéricos para identificar os produtos, vendedores e clientes). Neste caso, apenas existe um caminho de execução possível, sendo a informação semântica extraída apresentada na figura como comentários (linhas iniciadas por --).

10.5.2 Inferir relações

As relações estáticas entre cada par de transacções são inferidas a partir da informação semântica extraída de cada uma das transacções usando as regras que se apresentam nesta subsecção. Diz-se que o valor de uma relação é verdadeiro se a relação se estabelece entre as duas transacções e falso no caso contrário.

Durante a avaliação das regras definidas é necessário verificar se as descrições semânticas obtidas a partir das transacções referem os mesmos elementos de dados. Esta verificação inclui os elementos

```
BEIGN
  SELECT stock,price INTO l_stock,l_price FROM products WHERE id = 80;
      -- l_stock = read(products,id=80,stock)
      -- l_price = read(products,id=80,price)
  IF l_price <= 50.0 AND l_stock >= 10 THEN
          -- precond(read(products,id=80,stock)>=10)
          -- precond(read(products,id=80,price)<=50.0)
      UPDATE products SET stock = l_stock - 10 WHERE id = 80;
            -- update(products,id=80,stock,"read(products,stock,id=80)-10")
            -- readset += {read(products,id=80,---)}
      INSERT INTO orders VALUES (newid,8785,80,10,l_price,'to process');
            -- insert(orders,(id,client,product,qty,price,status),(newid,80,10,l_price,'processing'))
      COMMIT;
            -- readset = {read(products,id=80,stock),read(products,id=80,price),read(products,id=80,---)}
            -- writeset = {update(...),insert(...)}
            -- precondset = {precond(read(products,id=80,stock)>=10),precond(read(products,id=80,price)<=50.0)}
  ELSE
      ROLLBACK;
  END IF;
END;
```

Figura 10.5: Informação extraída de uma transacção móvel nova encomenda. A informação obtida de cada instrução é apresentada na linha seguinte (o bloco de declaração de variáveis é omitido).

de dados alvo das operações e os elementos de dados usados na selecção dos registos afectados. Por exemplo, uma instrução de leitura acede não só às colunas lidas, mas também às colunas usadas nas condições de selecção (expressas no elemento *where* da instrução de leitura).

No protótipo desenvolvido, estas verificações são executadas usando o sistema genérico de resolução de restrições JSolver [74]. Análises semelhantes são efectuadas no âmbito dos sistema de bases de dados na optimização de interrogações [4, 75, 21] e nos sistemas de replicação secundária semântica [35, 142].

Por vezes, é impossível determinar estaticamente se duas descrições semânticas referem um mesmo elemento de dados. Por exemplo, duas instruções de leitura que usem condições sobre diferentes colunas para seleccionar os registos a ler não podem ser comparadas estaticamente de forma precisa. Nestes casos, se o resultado desta comparação for importante para inferir o valor da relação, deve usar-se o valor definido por omissão como o valor da relação estabelecido entre as duas transacções. Uma possível solução para este problema é apresentado na secção 10.5.4.1.

Os valores definidos por omissão para cada relação foram escolhidos para garantir que é possível obter uma solução óptima com o algoritmo de reconciliação apresentado anteriormente. Para tal, em caso de dúvida, as transacções devem ser agrupadas conjuntamente (relação comutativas falsa) e integradas no conjunto das transacções candidatas (relação favorece verdadeira).

A correcção da solução produzida pelo algoritmo de reconciliação não é posta em causa por assumir um valor diferente para qualquer das relações definidas (com excepção da relação compensa). Em particular, assumir que duas transacções são comutativas apenas pode levar a que essas transacções sejam agrupadas em diferentes partições podendo (eventualmente) impedir a criação de uma solução óptima caso elas não sejam independentes.

De seguida apresentam-se as regras usadas para inferir as relações estáticas. Estas regras assumem que cada transacção apenas tem associada um grupo de informação semântica. Caso as transacções tenham associados vários grupos de informação é necessário aplicar as regras que se seguem às diferentes combinações. Se para uma relação forem obtidos valores diferentes para diferentes combinações, o valor da relação é o definido por omissão.

Adicionalmente, as regras apresentadas apenas têm em consideração as restrições de unicidade. No caso de existirem outras restrições ou invariantes na base de dados, a leitura/escrita de um elemento de dados, que esteja relacionado com outros elementos através de uma restrição/invariante, deve ser considerada como se todos esses elementos fossem lidos/escritos. A informação relativa a estas restrições deve também ser usada na inferência das relações torna possível, impossibilita, favorece, desfavorece.

10.5.2.1 Comutativas

Duas transacções móveis t_1 e t_2 são comutativas a menos que alguma das seguintes regras seja verdadeira:

- t_1 lê um elemento de dados escrito (inserido, actualizado ou removido) por t_2 ou vice-versa.
- t_1 escreve um elemento de dados escrito por t_2, com excepção de escritas comutativas, ou vice-versa.

Se for impossível obter o resultado das regras anteriores, deve assumir-se que t_1 e t_2 não são comutativas.

10.5.2.2 Impossibilita

Uma transacção t_1 impossibilita uma transacção t_2 se alguma das seguinte regras for verdadeira.

- t_1 modifica (insere, actualiza ou remove) a base de dados de forma a tornar falso um dos elementos do conjunto de pré-condições[6] de t_2.
- As modificações conjuntas de t_1 e $t2$ violam uma restrição de unicidade da base de dados.

Se for impossível obter o resultado das regras anteriores, deve assumir-se que t_1 não impossibilita t_2.

10.5.2.3 Torna possível

Uma transacção t_1 torna possível uma transacção t_2 se a seguinte regra for verdadeira.

[6] Note-se que um caminho de execução apenas é executado se todas as condições do conjunto de pré-condições forem verdadeiras.

- t_1 modifica (insere, actualiza ou remove) a base de dados de forma a que sejam verdadeiros todos os elementos do conjunto de pré-condições de t_2.

Se for impossível obter o resultado da regra anterior, deve assumir-se que t_1 não torna possível t_2.

10.5.2.4 Desfavorece

Uma transacção t_1 desfavorece uma transacção t_2 se a seguinte regra for verdadeira.

- t_1 modifica a base de dados de forma a prejudicar a veracidade de, pelo menos, um dos elementos do conjunto de pré-condições de t_2. Por exemplo, seja x uma variável (coluna de um registo) na base de dados, $x \leftarrow x - 1$ prejudica a veracidade da pré-condição $x \geq 10$.

Se for impossível obter o resultado da regra anterior, deve assumir-se que t_1 desfavorece t_2.

10.5.2.5 Favorece

Uma transacção t_1 favorece uma transacção t_2 se a seguinte regra for verdadeira.

- t_1 modifica a base de dados de forma a favorecer a veracidade de, pelo menos, um dos elementos do conjunto de pré-condições de t_2. Por exemplo, seja x uma variável (coluna de um registo) na base de dados, $x \leftarrow x + 1$ favorece a veracidade da pré-condição $x \geq 10$.

Se for impossível obter o resultado da regra anterior, deve assumir-se que t_1 favorece t_2.

10.5.2.6 Absorve

Uma transacção t_1 absorve uma transacção t_2 se todas as seguintes regras são verdadeiras.

- Todos os registos inseridos por t_2 são removidos por t_1.
- Todos os registos removidos por t_2 são reinseridos por t_1 ou igualmente removidos por t_1.
- Todos os dados actualizados por t_2 são também actualizados (usando valores absolutos) por t_1.
- As modificações produzidas por t_2 não influenciam a execução de t_1.

Estas regras apenas são válidas se for possível executar t_1 após executar t_2. Assim, deve ser possível executar t_1 em todos os estados da base de dados que resultam de executar t_2 num qualquer estado da base de dados em que tal seja possível. Para tal, é necessário que as pré-condições de t_1 sejam tão gerais como as pré-condições de t_2, quando modificadas pelas operações de modificação definidas em t_2.

Se for impossível obter o resultado das regras anteriores, deve assumir-se que t_1 não absorve t_2.

```
1  BEGIN
2    SELECT status INTO l_status FROM orders WHERE id = 3;
3        -- l_status = read(orders,id=3,status)
4    IF l_status = 'to process' THEN
5            -- precond(read(orders,id=3,status)='to process')
6        UPDATE orders SET status = 'cancelled' WHERE id = 3;
7                -- update(orders,id=3,status,'cancelled')
8        UPDATE products SET stock = stock + 30 WHERE id = 79;
9                -- update(products,id=79),stock,stock+30)
10       COMMIT;
11   ELSE
12       ROLLBACK;
13   END IF;
14 END;
```

Figura 10.6: Informação extraída da transacção móvel *cancela encomenda* definida sem indirecções (o bloco de declaração de variáveis é omitido).

10.5.2.7 Compensa

Uma transacção t_1 compensa uma transacção t_2 se todas as seguintes regras são verdadeiras.

- Todos os registos inseridos por t_2 e apenas esses são removidos por t_1.
- Todos os registos removidos por t_2 são reinseridos por t_1.
- Todos as actualizações executadas por t_2 são compensadas por actualizações executados por t_1.
- t_1 não produz mais nenhuma modificação.

Como anteriormente, as regras anteriores apenas são válidas se for possível executar t_1 após executar t_2. Se for impossível obter o resultado das regras anteriores, deve assumir-se que t_1 não compensa t_2.

10.5.3 Exemplos

Para exemplificar o funcionamento do mecanismo de inferência automática de relações vamos considerar os exemplos apresentados na secção 9.1.

Suporte a uma força de venda Relativamente ao suporte a uma força de vendas móvel considerem-se as transacções de introdução e cancelamento de uma encomenda apresentadas nas figuras 10.5 e 10.6 respectivamente. A partir da informação extraída, apresentada nas figuras, é possível estabelecer as seguintes relações.

Duas operações, quaisquer que sejam, relativas a diferentes produtos são sempre comutativas. Assim, é possível dividir o problema da reconciliação em subproblemas, cada um relativo a um produto diferente.

Duas encomendas relativas ao mesmo produto não são comutativas e desfavorecem-se mutuamente porque a subtracção de uma constante positiva à existência (*stock*) do produto é prejudicial para a veracidade da condição que verifica se a existência disponível é suficiente para satisfazer a encomenda.

```
1  ------ INSERE MARCAÇÃO: '16-FEB-2002' às 10:00, vendedor='J.Smith'
2  BEGIN
3    id = newid;
4       -- id = 64374
5    SELECT count(*) INTO cnt FROM datebook WHERE day='16-FEB-2002' AND hour>=9 AND hour<12 AND user='J.Smith';
6       -- cnt = read(datebook,day='16-FEB-2002' AND hour>=9 AND hour<12 AND user='J.Smith',count(*))
7    IF (cnt = 0) THEN                                   --- Verifica disponibilidade
8            -- precond(read(datebook,...,count(*))=0)
9      INSERT INTO datebook VALUES( id, '16-FEB-2002', 10, 'J.Smith', 'Demonstração BLUE THING');
10           -- insert(datebook,(id,day,hour,user,info),
11           --                (64374,'16-FEB-2002',10,'J.Smith','Demonstração BLUE THING'))
12     COMMIT '16-FEB-2002';                             --- Conclui e devolve marcação
13   END IF;
14   ROLLBACK;
15 ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Marcação impossível... ');
16 END;
```

Figura 10.7: Informação extraída da transacção móvel que introduz uma nova marcação simples numa agenda partilhada (o bloco de declaração de variáveis é omitido).

Dois cancelamentos relativos à mesma encomenda impossibilitam-se mutuamente porque após cancelar uma encomenda (atribuindo-lhe o estado *cancelada*) não é possível voltar a cancelá-la (porque o seu estado já não é *a processar*). Dois cancelamentos de encomendas diferentes relativas ao mesmo produto são comutativas porque a operação executada sobre a existência do produto é comutativa.

O cancelamento de uma encomenda favorece a introdução de uma nova encomenda relativa ao mesmo produto, porque a adição de uma constante positiva à existência (*stock*) do produto é favorável à veracidade da condição que verifica se a existência disponível é suficiente para satisfazer a encomenda.

A introdução de uma encomenda torna possível o cancelamento da mesma encomenda porque o registo da encomenda torna verdadeira a pré-condição do cancelamento. No entanto, ao contrário do que se poderia esperar, o cancelamento não compensa a introdução da encomenda porque fica registado no sistema o facto de a encomenda ter existido e ter sido cancelada. Este exemplo mostra a dificuldade de utilizar, na prática, as operações de compressão de uma sequência de operações.

Adicionalmente, como se explica próxima secção, pode estabelecer-se a relação predecessor/sucessor fraco entre uma transacção que submete uma encomenda e outra que remove a mesma encomenda no mesmo caminho de execução porque a remoção acede ao registo da encomenda inserido anteriormente pela submissão da transacção.

As relações extraídas automaticamente são as esperadas, levando a que, relativamente a um dado produto, os cancelamentos sejam efectuados antes da introdução de novas encomendas. Esta aproximação aumenta a probabilidade de aceitar um maior número de novas encomendas.

Agenda partilhada Relativamente a uma agenda partilhada considerem-se as transacções de introdução e remoção de uma marcação, apresentadas nas figuras 10.7 e 10.8 respectivamente. A partir da informação extraída, apresentada nas figuras, é possível estabelecer as seguintes relações.

```
1  ------ REMOVE MARCAÇÃO: '16-FEB-2002' às 10:00, utilizador='J.Smith', id=3476554
2  BEGIN
3    SELECT count(*) INTO cnt FROM datebook
4                    WHERE id=3476554 AND day='16-FEB-2002' AND hour=10 AND user='J.Smith';
5       -- cnt = read(datebook,id=3476554 AND day='16-FEB-2002' AND hour=10 AND user='J.Smith',count(*))
6    IF (cnt > 0) THEN
7            -- precond(read(datebook,...,count(*))>0)
8      DELETE FROM datebook WHERE id = 3476554 AND day='16-FEB-2002' AND hour=10 AND user='John Smith';
9            -- delete(datebook,id=3476554 AND day='16-FEB-2002' AND hour=10 AND user='J.Smith')
10     COMMIT;
11   END IF;
12   ROLLBACK;
13 END;
```

Figura 10.8: Informação extraída da transacção móvel que cancela uma marcação numa agenda partilhada usando os detalhes da marcação (o bloco de declaração de variáveis é omitido).

Duas operações relativas a marcações não sobrepostas são comutativas. A remoção de uma marcação favorece a introdução de uma marcação sobreposta com a primeira, porque a remoção do registo da marcação é favorável à inexistência de um registo que impossibilite a introdução da marcação. A introdução de uma marcação torna possível a remoção da mesma marcação porque o registo inserido garante a veracidade da pré-condição de remoção.

As relações inferidas são as esperadas, levando à execução das operações de remoção antes das operações de introdução de novas marcações. As operações relativas à mesma marcação são executadas pela ordem esperada, i.e., as remoções após as inserções.

Sistema de reserva de bilhetes de comboio As relações inferidas relativamente às transacções definidas no sistema de reserva de bilhetes de comboio são idênticas às obtidas na aplicação de suporte a uma força de vendas. Note-se que o facto de ser necessário obter o identificador do bilhete vendido não influencia as relações inferidas porque esses elementos de dados não influenciam as pré-condições definidas.

10.5.4 Extensões

Até ao momento descreveu-se o funcionamento básico da inferência de relações semânticas a partir do código das transacções móveis. As funcionalidades descritas foram implementadas no protótipo do sistema SqlIceCube. Nesta subsecção propõem-se algumas extensões ao modelo básico descrito. Estas extensões estão a ser implementadas no sistema SqlIceCube.

Os exemplos apresentados na subsecção anterior mostram exemplos em que é possível inferir as relações esperadas a partir do código das transacções móveis. No entanto, como se referiu anteriormente, existem situações em que não é possível obter imediatamente a informação semântica necessária para o processo de inferência de relações. Nesta subsecção discutem-se essas situações e apresentam-se

soluções automáticas para ultrapassar as dificuldades surgidas. Os programadores podem também contribuir para melhorar o funcionamento do sistema de inferência automática evitando escrever transacções móveis que originem as situações descritas.

10.5.4.1 Descrições semânticas

Um dos problemas que pode surgir durante o processo de inferência de relações é a dificuldade ou impossibilidade de comparar duas descrições semânticas de forma precisa. Este problema pode ser consequência da utilização de valores lidos pela transacção nas condições de selecção de registos das operações de leitura e escrita (indirecções) ou da utilização de condições sobre colunas diferentes de uma mesma tabela (condições *incompatíveis*).

Indirecções A transacção móvel de cancelamento de uma encomenda apresentada na figura 9.1 exemplifica o problema da utilização de indirecções. Nesta transacção, a actualização da existência do produto cancelado (na linha 6) é efectuada usando a informação obtida a partir do registo da encomenda (na linha 3). Em particular, é impossível saber qual o produto envolvido no cancelamento e se a operação de actualização adiciona um valor positivo ou negativo à existência do produto. Assim, é impossível inferir de forma exacta as relações estabelecidas entre esta e qualquer outra transacção móvel de introdução ou cancelamento de uma encomenda porque não é possível verificar se as transacções acedem aos mesmos elementos de dados e como é que estes são actualizados.

Este problema pode ser ultrapassado combinando a análise estática com a avaliação dinâmica das constantes. Dado um conjunto de transacções a reconciliar, existem elementos de dados lidos que não são modificados por nenhuma transacção (qualquer que seja o caminho de execução percorrido na transacção). Durante o processo de reconciliação, estes elementos de dados são constantes e o seu valor pode ser obtido a partir da base de dados e usado no processo de inferência de relações.

Esta aproximação permite obter, em muitas situações, a informação necessária à avaliação das regras de inferência de relações. No exemplo anterior, a informação relativa à encomenda cancelada (identificador do produto e quantidade envolvida) são constantes. Assim, a transacção a processar, obtido o valor das constantes, será semelhante à apresentada na figura 10.6, a qual permite inferir as relações esperadas.

Para verificar se um elemento de dados é constante é igualmente necessário comparar as descrições semânticas das escritas. No entanto, como as instruções de escrita incluem geralmente o identificador do registo modificado, tende a ser possível efectuar essa verificação.

Condições *incompatíveis* As transacções móveis de introdução e cancelamento de uma marcação apresentadas, respectivamente, nas figuras 10.7 e 9.2 exemplificam o problema da utilização de condições *incompatíveis*. Quando se cancela uma marcação usa-se o seu identificador, enquanto na verificação da

possibilidade de inserir uma marcação se usam os detalhes das marcações (para verificar que não existe nenhuma marcação sobreposta). Assim, é impossível verificar imediatamente se as operações de leitura e escrita acedem aos mesmos elementos de dados[7].

Como a informação relativa a uma marcação é, em geral, constante, é possível estender a aproximação anterior para ultrapassar este problema. Assim, é possível obter informação adicional relativa aos registos acedidos nas operações de leitura e escrita. Esta informação pode ser suficiente para efectuar uma comparação precisa entre as condições expressas em diferentes operações de leitura e escrita.

Na operação de cancelamento apresentada anteriormente, seria possível obter a informação relativa à marcação a cancelar. Assim, após obter a informação adicional, a transacção a processar seria equivalente à apresentada na figura 10.8, a qual permite inferir as relações esperadas entre as transacções.

10.5.4.2 Ciclos

Uma transacção móvel que inclua ciclos (por exemplo, instruções *for*) pode ser processada da seguinte forma (durante a fase de extracção de informação).

- Quando é possível determinar o valor da variável de controlo do ciclo, o processamento é trivial. Neste caso, basta analisar repetidamente as instruções do bloco do ciclo enquanto o valor da variável de controlo satisfizer a condição de repetição. Por exemplo, num ciclo *for i in 1..3 loop*, deve repetir-se a análise das instruções internas ao ciclo três vezes, com a variável *i* a tomar consecutivamente os valores 1, 2 e 3.

- Quando o valor da variável de controlo depende de um valor lido da base de dados (por exemplo, o ciclo é controlado por um *cursor*) existem duas situações. Se é possível determinar que esses valores são constantes, é possível analisar o ciclo como anteriormente após resolver a indirecção. Caso contrário, a informação obtida incluirá sempre indirecções. O estudo deste problema, recorrente na análise estática de programas [49], bem como o benefício que se pode extrair da obtenção da informação adicional, é um tópico de trabalho futuro.

[7]Neste caso, apenas é impossível determinar de forma precisa se duas transacções são comutativas. Todas as outras relações podem ser inferidas correctamente apesar da informação obtida não ser óptima. Uma nova marcação torna possível o cancelamento da marcação respectiva, porque o registo inserido garante a veracidade da pré-condição do cancelamento. Qualquer outra nova marcação não favorece nem desfavorece um cancelamento porque é possível verificar que as modificações não têm influência na pré-condição do cancelamento (usando o identificador da marcação). Por omissão, qualquer cancelamento favorece uma nova marcação (como esperado). Um cancelamento não desfavorece uma nova marcação porque a remoção de um registo nunca modifica a base de dados de forma desfavorável para a pré-condição da inserção de uma nova marcação.

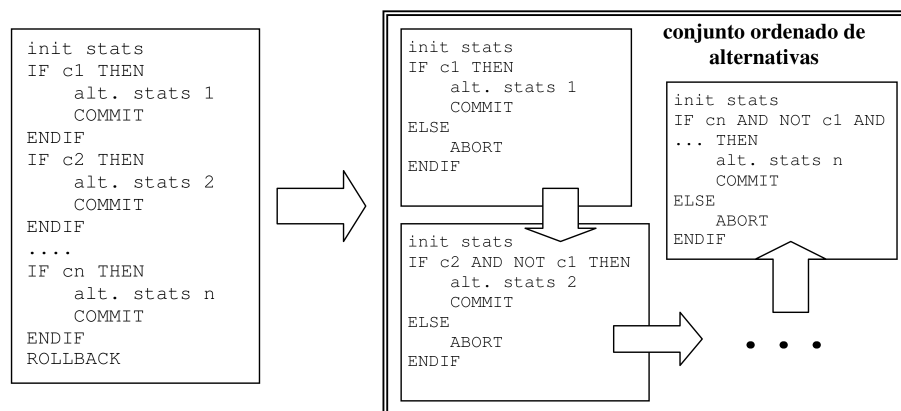

Figura 10.9: Divisão de uma transacção composta por uma sequência de instruções *if* num conjunto ordenado de transacções alternativas.

10.5.4.3 Transacções complexas

O mecanismo de inferência de relações pode ter dificuldade em lidar com transacções longas e complexas que acedam a muitos elementos de dados ou incluam vários caminhos de execução possíveis. Devido a estas características, estas transacções podem estabelecer relações, possivelmente contraditórias, com um elevado número de outras transacções. Assim, as relações estabelecidas podem tornar-se inúteis para guiar a pesquisa executada pelo algoritmo de reconciliação.

Por exemplo, a utilidade da relação de comutatividade consiste em separar as transacções em conjuntos de transacções independentes. O facto de as transacções complexas tenderem a ser comutativas com um menor número de outras transacções põe em causa a possibilidade de separar as transacções a reconciliar e, portanto, o papel que esta relação deve desempenhar.

Para evitar estes problemas, a API do sistema SqlIceCube permite aos programadores submeterem as transacções longas como um conjunto de pequenas transacções ligadas pelas relações da aplicação apropriadas. No protótipo do sistema Mobisnap, esta API não está directamente disponível[8].

O sistema SqlIceCube pode ainda pré-processar as transacções móveis complexas e dividi-las num conjunto de pequenas transacções ligadas pelas relações da aplicação apropriadas. As seguintes regras de pré-processamento podem ser aplicadas:

- A separação dos vários caminhos de execução de uma transacção móvel num conjunto ordenado de transacções alternativas (usando as relações alternativas e predecessor/sucessor fraco). Nas

[8]Para permitir que as aplicações especifiquem as relações da aplicação directamente, garantindo o respeito pelas relações definidas durante o processamento normal, poder-se-ia usar a seguinte aproximação. As transacções ligadas pela relação grupo indivisível são executadas como uma única transacção apenas após a submissão da última transacção que pertença ao grupo indivisível. Para respeitar as outras relações é apenas necessário abortar as transacções cuja execução viole as relações estabelecidas, tendo em conta as transacções já executadas com sucesso.

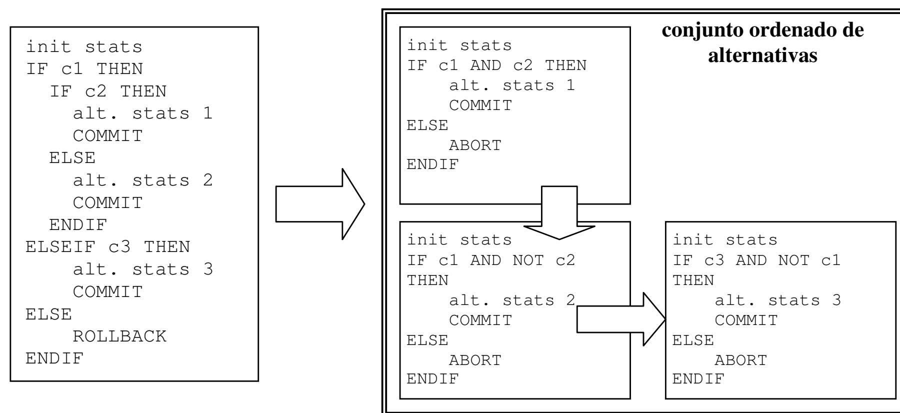

Figura 10.10: Divisão de uma transacção composta por um conjunto de instruções *if* encadeadas num conjunto ordenado de transacções alternativas.

figuras 10.9 e 10.10 apresentam-se exemplos deste pré-processamento.

- A divisão de uma transacção longa num conjunto de pequenas transacções associadas num grupo indivisível e ligadas, se necessário, pela relação predecessor/sucessor.

O sistema pode ainda inferir automaticamente a causalidade entre as transacções submetidas por uma aplicação usando a seguinte regra: se uma transacção aceder (ler ou modificar) a um elemento de dados escrito por uma transacção submetida anteriormente, pode assumir-se que existe uma relação predecessor/sucessor fraco (a segunda transacção deve ser executada após a execução da primeira).

10.6 Observações finais

O sistema SqlIceCube é um sistema de reconciliação genérico que aborda o processo de reconciliação como um problema de optimização, aproximação da qual foi pioneiro o sistema IceCube [85]. O sistema pesquisa o espaço de possíveis soluções eficientemente [133, 134] com base numa heurística baseada na informação semântica associada às transacções móveis.

Ao contrário dos sistemas de reconciliação que exigem que o programador exponha a semântica das operações, o sistema SqlIceCube é o primeiro a incluir um mecanismo de inferência automática que permite extrair a informação semântica necessária a partir do código das transacções móveis. Esta aproximação permite simplificar bastante a utilização dum sistema de reconciliação baseado na utilização da informação semântica. Uma aplicação pode, caso o deseje, impor relações arbitrárias adicionais entre as transacções que executa.

Os exemplos apresentados mostram que, caso se tenham alguns cuidados na definição das transacções móveis, o sistema SqlIceCube consegue inferir as relações semânticas esperadas. As extensões propostas permitem eliminar (a maioria) esses cuidados.

Capítulo 11

Trabalho relacionado

A gestão de dados em sistemas distribuídos de larga escala tem sido um tópico de investigação importante e inclui um grande conjunto de problemas. Em particular, nos últimos anos tem sido dedicada grande atenção ao desenvolvimento de soluções que permitam lidar com as características intrínsecas dos sistemas de computação móvel [7, 149]. Algumas das soluções propostas são apresentadas conjuntamente em vários artigos e livros recentes [124, 78, 147, 13].

Neste capítulo, discutem-se os problemas de gestão de dados abordados nesta dissertação e apresentam-se as soluções mais relevantes propostas anteriormente, com particular destaque para os problemas relacionados com a replicação dos dados. Alguns dos problemas descritos neste capítulo não foram investigados no âmbito desta dissertação. A sua descrição permite, no entanto, ter uma perspectiva mais abrangente dos problemas e abordagens propostas para a gestão de dados partilhados em ambientes de computação móvel.

11.1 Replicação

A replicação é uma técnica fundamental na criação de serviços distribuídos, na qual várias elementos de um sistema distribuído mantém cópias de um objecto. As suas potenciais vantagens são o aumento da disponibilidade e a melhoria do desempenho. Nesta secção discutem-se vários aspectos relacionados com a replicação.

11.1.1 Modelos

11.1.1.1 Replicação pessimista

Os algoritmos de replicação pessimista oferecem uma semântica de cópia única (*single-copy serializability*), fornecendo a ilusão que existe uma única cópia de cada objecto que evolui sequencialmente. Assim, as operações de leitura e escrita apenas podem ser executadas se for possível aceder ao valor actual do

objecto (caso contrário são bloqueadas), devendo por isso a execução das operações ser sincronizada entre si (por esta razão, também se chama a este tipo de replicação, replicação síncrona ou *eager replication*). Várias formas de alcançar este objectivo foram propostas [17], nas quais se podem distinguir dois elementos fundamentais. Primeiro, o número de réplicas que é acedido em cada operação, que pode variar entre uma estratégia simples de *lê-uma-réplica-escreve-todas* [114] até aproximações que lêem e escrevem quorum de réplicas [163]. Segundo, a estratégia usada para sincronizar e ordenar a execução das várias operações (de escrita), entre as quais se podem referir a utilização de uma réplica principal [8, 95], de um sistema de comunicação em grupo síncrono [83, 2] ou de protocolos de *desistência progressiva* (*backoff*) [28].

Embora funcionem bem num ambiente de rede local, as técnicas de replicação pessimista não são apropriadas para ambientes de larga escala. Em primeiro lugar, a latência mais elevada e a menor fiabilidade das redes de comunicação de larga escala levam a que o tempo de execução das operações seja elevado e, em alguns casos, a uma reduzida disponibilidade (para escrita) dos dados [30] (a utilização de sistemas de quorum minimiza este problema). Em segundo lugar, estas técnicas impedem qualquer acesso aos dados em situações de desconexão (ou, em alternativa, a desconexão de uma réplica impede a modificação dos dados por todos os outros utilizadores).

11.1.1.2 Replicação optimista

Os algoritmos de replicação optimista permitem aos utilizadores aceder a qualquer réplica dos dados, no pressuposto optimista que os conflitos são raros ou possíveis de resolver. Assim, as operações de leitura e escrita são executadas numa (qualquer) réplica sobre o valor dessa réplica. As operações de modificação são propagadas de forma diferida para as outras réplicas (por esta razão também se chama a este tipo de replicação, replicação retardada — *lazy replication*).

A replicação optimista apresenta várias vantagens relativamente à replicação pessimista, das quais se destacam: o aumento da disponibilidade, permitindo o acesso aos dados desde que uma réplica esteja acessível; a flexibilidade no uso da rede de comunicações, permitindo efectuar as comunicações nos períodos mais apropriados (em termos de intensidade de tráfego, custo e condições de conectividade) e recorrendo a computadores intermédios; a diminuição do tempo de resposta, permitindo obter o resultado localmente. Estas vantagens são obtidas sacrificando a consistência dos dados entre as várias réplicas, pelo que nem todas as aplicações podem utilizar este tipo de replicação. No entanto, para permitir a operação desconectada é necessário recorrer a este tipo de solução. Esta aproximação é usada, não só nos sistema apresentados nesta dissertação, mas também na generalidade dos sistema de gestão de dados para ambientes de computação móvel [124, 78, 147, 13]. De seguida, discutem-se várias características dos sistemas baseados em replicação optimista.

11.1.2 Arquitecturas e tipos de réplicas

Um sistema baseado em replicação optimista pode ser construído usando diferentes arquitecturas. De uma forma muito simplificada podem identificar-se dois tipos básicos de arquitectura: cliente/servidor e *participante-a-participante* (*peer-to-peer*). Enquanto numa arquitectura cliente/servidor, o sistema é composto por dois tipos de elementos com funções distintas (os servidores e os clientes)[1], numa arquitectura *participante-a-participante* todos os elementos são idênticos e exercem as mesmas funções.

Num sistema cliente/servidor, os clientes podem ser simples (*thin clients* [78]) ou estendidos (*extended client* ou *full client* [78]). Um cliente simples não mantém nenhuma réplica dos objectos, necessitando de contactar um servidor para executar qualquer operação. Desta forma, este tipo de sistemas não permite a operação desconectada a menos que um servidor execute na mesma máquina do cliente. Neste caso, estes sistemas têm propriedades semelhantes às dos sistemas *participante-a-participante*. O sistema Bayou [161] usa esta aproximação para suportar a operação desconectada.

Um cliente estendido mantém uma réplica (parcial) dos dados e executa localmente (um subconjunto) das funcionalidades do servidor. As operações submetidas pelas aplicações podem ser executadas localmente recorrendo à réplica local dos dados. Desta forma, pode suportar-se a execução de operações durante os períodos de desconexão. Os clientes do sistema de ficheiros Coda [87] usam esta aproximação.

Num sistema *participante-a-participante* todas as máquinas mantêm réplicas dos objectos. A operação desconectada sobre objectos replicados localmente é permitida. O sistema de ficheiros Ficus [66] usa esta aproximação, com cada máquina a manter uma cópia de todos os objectos. O sistema Roam [140] usa esta aproximação, mas cada máquina mantém apenas um subconjunto dos objectos. O sistema Roam usa ainda uma organização hierárquica para simplificar a gestão da replicação entre um número muito elevado de réplicas: as réplicas agrupam-se em conjuntos disjuntos (*wards*). Em cada *ward* existe um réplica eleita, responsável por comunicar com os outros *wards*.

Num sistema cliente estendido/servidor, podem identificar-se dois tipos de réplicas [87]. As réplicas de primeira classe nos servidores e as réplicas de segunda classe (ou cópias da *cache*) nos clientes. Em geral, as réplicas de primeira classe são mais completas, disponíveis e seguras — o algoritmo de replicação tem como missão principal actualizar estas réplicas. As réplicas de segunda classe são uma cópia das réplicas de primeira classe, a partir das quais devem ser actualizadas regularmente.

O número de réplicas principais nos sistemas cliente/servidor e o número de réplicas nos sistema *participante-a-participante* é geralmente variável. Em geral, a criação deste tipo de réplica é uma operação que deve ser efectuada pelo administrador do sistema [101, 87]. No entanto, existem sistemas [172, 86] que criam novas réplicas automaticamente para manter uma boa qualidade de serviço.

[1] Uma mesma máquina pode conter um elemento servidor e um elemento cliente.

Os sistemas apresentados nesta dissertação são baseados na arquitectura cliente estendido/servidor. Esta aproximação permite suportar um elevado número de clientes capazes de modificar os objectos, cada um mantendo réplicas de apenas um subconjunto de objectos. No entanto, o algoritmo de replicação principal, responsável por controlar a evolução dos objectos, é executado apenas entre os servidores. O reduzido número de servidores e a sua boa conectividade permite que as operações sejam reflectidas em todas as réplicas principais rapidamente. A existência de múltiplos servidores (DOORS) permite fornecer uma maior disponibilidade das réplicas principais à custa do aumento de complexidade na reconciliação. A criação de uma nova réplica num servidor (DOORS) deve ser efectuada pelo administrador do sistema. A possibilidade de os clientes comunicarem entre si para actualizar as suas cópias dos objectos permite que quaisquer dois elementos do sistema cooperem sem necessidade de contactar um servidor, ultrapassando uma das limitações geralmente apontadas aos sistemas cliente/servidor [139]. Relativamente a um sistema *participante-a-participante*, esta aproximação tem a desvantagem de a sincronização entre os clientes ser, em geral, mais primitiva. No entanto, o menor número de réplicas principais existentes simplifica a gestão da consistência das réplicas.

11.1.3 Unidade de replicação

A unidade de replicação é o menor elemento que pode ser replicado independentemente, a que se chama genericamente objecto. A utilização de unidades de replicação de grandes dimensões tem vantagens, de entre as quais se destacam as seguintes: simplifica a gestão da replicação, devido ao menor número de objectos existentes; e diminui a probabilidade da ocorrência de erros devido à não replicação de algum objecto. No entanto, a grande dimensão dos objectos impõe elevadas exigências de recursos para armazenar e obter uma nova réplica de um objecto. Estas exigências podem estender-se à propagação de modificações, no caso de ser necessário propagar o novo estado de todo o objecto.

Assim, enquanto computadores bem conectados com grande capacidade de armazenamento podem satisfazer as necessidades relativas à utilização de unidades de replicação de grandes dimensões, os computadores móveis fracamente conectados e com limitada capacidade de armazenamento necessitam de utilizar unidades de replicação de pequenas dimensões.

Nos sistemas cliente/servidor (Coda [87], Oracle Lite [115], Objectivity [109]), em que se pressupõe a diferença de qualidade entre os clientes e os servidores, é comum existirem duas unidades de replicação. Por exemplo, no sistema Coda, a unidade de replicação entre os servidores é o volume (conjunto de ficheiros), enquanto os clientes podem obter cópias de ficheiros individuais. A aproximação utilizada no sistema DOORS é semelhante: os servidores replicam conjuntos de *coobjectos*, enquanto os clientes podem obter cópias parciais de *coobjectos* individuais.

Nos sistema *participante-a-participante*, em geral, apenas existe uma unidade de replicação. Quando

a unidade de replicação é de grandes dimensões (conjunto de ficheiros no sistema Ficus [66] e base de dados no sistema Bayou [161]), o sistema tem dificuldade em suportar computadores com pequena capacidade de armazenamento, como é comum nos computadores portáteis. Quando a unidade de replicação é menor (um ficheiro no sistema Roam [140]), o sistema tem de gerir um elevado número de objectos.

Dimensão da menor unidade de replicação A dimensão da menor unidade de replicação deve ser suficientemente pequena para que não se repliquem dados desnecessários e suficientemente grande para que a gestão do processo não se torne demasiado complexa. Como se discutiu na secção 2.3.5, a unidade de manipulação de um sistema de gestão de dados é, por vezes, inadequada como unidade de replicação. Assim, no sistema DOORS, a unidade de replicação é definida pelos programadores de acordo com o modo como os objectos estão internamente organizados. Esta aproximação permite adequar a unidade de replicação às características dos dados manipulados por cada aplicação. Uma aproximação semelhante foi usada nos sistemas Javanaise [68] e Perdis [51] para melhorar o desempenho num ambiente de computação distribuída.

O agrupamento automático de objectos, como executado nos sistemas de bases de dados orientadas para os objectos [165], não soluciona o problema de gestão, porque o sistema continua a ter de lidar com todos os objectos no momento de determinar quais devem ser replicados. Pelo contrário, no sistema DOORS, a unidade de replicação define uma caixa negra, o que minimiza o número de objectos com que o sistema necessita de lidar. Num sistema de ficheiro é possível dividir um documento longo em vários ficheiros, obtendo um resultado semelhante ao proposto no sistema DOORS. No entanto, esta aproximação não é natural.

Adicionalmente, a unidade de replicação definida no sistema DOORS, o subobjecto, define um elemento no qual é possível executar operações de forma independente. Ao contrário do que acontece nos sistemas desenvolvidos para ambientes distribuídos, nos quais é possível replicar incrementalmente [166] os objectos necessários ao funcionamento da aplicação, num ambiente de computação móvel esta propriedade é importante por permitir o funcionamento (ainda que parcial) durante os períodos de desconexão.

No sistema Mobisnap, a unidade de replicação é o subconjunto de um registo. No entanto, a especificação dos dados a replicar é efectuada usando instruções de interrogação, o que permite lidar com os objectos a nível semântico. Esta aproximação é semelhante à adoptada nos sistemas de bases de dados relacionais que suportam computação móvel [115] e nas soluções de replicação secundária semântica [35, 142].

11.1.4 Replicação antecipada

Durante os períodos de desconexão é necessário recorrer a réplicas locais dos objectos. Assim, antes de se desconectarem, os computadores móveis devem obter uma réplica dos objectos que se prevê serem necessários. Este processo designa-se por replicação antecipada (*prefetching* ou *hoarding*) e deve ser executado por todos os computadores móveis que não repliquem a totalidade dos dados (o que só acontece em alguns sistemas *participante-a-participante*).

Uma técnica simples de replicação antecipada consiste em delegar nos utilizadores a especificação dos objectos que vão ser necessários. O sistema limita-se a manter uma cópia actualizada dos objectos especificados. Esta aproximação é usada, por exemplo, no sistema Oracle Lite [115].

Quando os clientes dispõem de muito espaço para armazenar as réplicas secundárias, a utilização de algoritmos de replicação secundária (*caching*) tradicionais [141, 54], como por exemplo o algoritmo *"menos recentemente utilizado"* (*least recently used*), pode ser suficiente para suportar a operação desconectada. Neste caso, o sistema de replicação antecipada limita-se a actualizar o estado das réplicas locais. No entanto, estes algoritmos simples têm dificuldade em suportar a mudança do foco da actividade do utilizador. Para ultrapassar este problema, pode combinar-se a utilização dum algoritmo de replicação secundária com a solução anterior, em que o cliente mantém cópias de um conjunto de objectos especificados pelo utilizador. Esta aproximação é usada no sistema Coda [87].

Recentemente, foram propostas técnicas de replicação antecipada baseadas numa análise mais complexa dos acessos efectuados pelos utilizadores [160, 91, 151].

O estudo de soluções de replicação antecipada está fora do âmbito do trabalho desta dissertação. Os sistema desenvolvidos incluem um subsistema de replicação antecipada que é informado dos objectos acedidos pelos utilizadores. Nos protótipos foram implementadas soluções simples, baseadas na utilização de um algoritmo de replicação secundária simples combinado com a especificação, por parte dos utilizadores, dos objectos que devem ser replicados.

11.1.5 Invocação cega

Durante os períodos de desconexão, o acesso aos dados é suportado recorrendo a uma cópia local. No entanto, desde que não se repliquem a totalidade dos objectos é sempre possível que o utilizador pretenda aceder a um objecto do qual não existe uma cópia local. Nesta situação, os sistemas de gestão de dados reportam a existência de um erro (*cache miss*) e não permitem a execução de qualquer operação.

Como se discutiu na secção 2.3.6, existem situações em que é possível adoptar uma aproximação optimista, permitindo aos utilizadores submeter operações de modificação sobre dados que não replicam. As soluções apresentadas nesta dissertação adoptam esta aproximação, que se chama genericamente de invocação cega.

O mecanismo de invocação diferida de procedimentos remotos (QRPC) do sistema Rover [79] é o que mais se aproxima da solução adoptada nesta dissertação. No sistema Rover, os objectos podem ser divididos em duas partes que correm no cliente e no servidor e cooperam entre si. O mecanismo de QRPCs permite invocar um procedimento remoto de forma assíncrona. Pode ser usado para obter uma cópia de um objecto ou para propagar uma operação executada na cópia local do objecto após esta ter sido importada do servidor.

O mecanismo de invocação cega proposto nesta dissertação é mais geral porque permite invocar operações sobre objectos dos quais apenas se possuí uma referência (DOORS) ou não se possui referência nenhuma (Mobisnap). O mecanismo de criação de cópias de substituição do sistema DOORS permite, adicionalmente, observar o resultado provisório das operações submetidas.

11.1.6 Propagação das modificações: propagar o quê?

Para sincronizar as réplicas de um objecto, é necessário propagar entre os elementos do sistema as modificações executadas. Para tal, existem duas técnicas fundamentais. Primeiro, a propagação do estado, na qual um elemento do sistema envia uma cópia do estado do objecto para outro elemento. Os seguintes sistemas utilizam esta técnica: Coda [87], Oracle Lite [115], Lotus Notes [101], Palm HotSync [117] e outro protocolos de sincronização para PDAs [1]. Segundo, a propagação de operações, na qual um elemento do sistema envia as operações que produziram as alterações no estado do objecto. Os seguintes sistemas utilizam esta aproximação: Bayou [161], Rover [79] e IceCube [85].

Para avaliar a eficiência de cada uma das técnicas existem vários aspectos que podem ser considerados, como se discutiu na secção 2.3.3.

Primeiro, o tráfego gerado ao propagar as modificações. Neste caso, a eficiência depende da situação e está directamente relacionada com a dimensão dos objectos e da representação das operações.

Em geral, um objecto tem uma dimensão muito superior à de uma operação. No entanto, um objecto pode reflectir um qualquer número de operações. Assim, quando o número de operações executadas é pequeno, a propagação de operações tende a ser mais eficiente. Para compensar este facto, alguns sistemas baseados em propagação do estado usam objectos de pequena dimensão (registos no Lotus Notes [101] e Palm HotSync [117]) ou usam a propagação de operações em algumas situações [152]. Alternativamente, quando um elemento tem acesso à versão do elemento a quem deseja enviar as modificações, pode propagar-se apenas a diferença entre ambas as versões [164].

Quando o número de operações executadas é grande, a vantagem da propagação de operações tende a reduzir-se (ou anular-se). Para compensar este facto, alguns sistemas baseados na propagação de operações comprimem a sequência de operações a propagar (por exemplo o sistema Rover [79]).

Segundo, é necessário avaliar a complexidade do processo de obtenção e propagação da informação.

Na propagação de operações é necessário interceptar e armazenar conjuntamente com os dados as operações executadas. Na propagação do estado apenas é necessário manter a réplica do objecto. Assim, a propagação de operações é mais complexa.

Terceiro, deve ter-se em consideração a informação disponível para o processo de reconciliação. A propagação de operações permite que o mecanismo de reconciliação utilize a informação semântica associada às operações executadas. Em ambos os casos é possível utilizar a informação semântica associada ao tipo de dados. Assim, a propagação de operações permite soluções potencialmente mais eficazes, embora dependa do mecanismo de reconciliação explorar ou não a informação adicional disponível.

Nos sistemas apresentados nesta dissertação utilizaram-se as duas aproximação. A propagação de operações (com compressão) é usada para transmitir para os servidores as modificações executadas pelos utilizadores, que se assume serem em pequeno número. Assim, maximiza-se a informação disponível para os mecanismos de reconciliação. A propagação do estado (utilizando objectos tendencialmente de pequenas dimensões) é usada para actualizar as réplicas dos objectos nos clientes. Neste caso, como o número de operações a enviar se assume ser grande é razoável usar esta técnica. No sistema DOORS é possível utilizar também a propagação de operações para actualizar as cópias dos clientes no caso de o número de operações a enviar ser reduzido. No sistema Mobisnap exclui-se esta hipótese por ser muito complexo manter a sequência das operações executadas (que incluem as operações executadas directamente por clientes legados).

11.1.7 Detecção das modificações

Para sincronizar duas réplicas é necessário compará-las para determinar se e como foram modificadas.

Uma aproximação possível consiste em comparar os seus valores. Como esta operação deve ser eficiente, a comparação integral das réplicas é inaceitável. Assim, é necessário definir um modo de identificar univocamente o valor de uma réplica. Para tal, pode calcular-se um identificador (quase) único a partir do estado do objecto utilizando, por exemplo, uma função de dispersão segura (SHA-1, MD5)[2].

Esta solução permite verificar a igualdade ao nível do conteúdo do objecto, mas não permite inferir qualquer informação sobre a evolução dos objectos. Assim, as soluções usadas nos sistemas de replicação privilegiam a obtenção de informação sobre a evolução das réplicas (em detrimento da informação exacta sobre a igualdade dos seus valores, porque se assume que a probabilidade de duas réplicas evoluirem concorrentemente para o mesmo estado é reduzido).

A utilização de relógios escalares (físicos ou lógicos [94]) para identificar os valores de uma réplica (ou operações) permite definir uma ordem total entre esses valores (ou operações) que respeita a causali-

[2]Esta aproximação é usada frequentemente para atribuir identificadores únicos a objectos imutáveis [29, 143].

dade[3]. Esta técnica permite verificar a desigualdade entre duas réplicas e identificar aquela que pode ser causalmente dependente da outra.

No entanto, esta técnica não permite verificar se os valores de duas réplicas foram modificados concorrentemente. Para tal, uma aproximação possível consiste em manter duas estampilhas temporais: a actual e a anterior. Esta aproximação é usada, por exemplo, no sistema Palm HotSync [117]. Esta técnica apenas é exacta quando se usam apenas duas réplicas.

A utilização de vectores-versão [119] permite verificar de forma exacta se duas réplicas foram modificadas concorrentemente (quer se use a propagação do estado ou de operações). Esta técnica é usada de forma generalizada nos sistemas de replicação com múltiplas réplicas (por exemplo, nos sistemas Ficus [66], Coda [87], Bayou [161]).

Como a dimensão do vector-versão é proporcional ao número de réplicas existentes no sistema[4], os vectores-versão podem ter dimensões significativas nos sistema de muito larga-escala. Assim, o problema da gestão dos vectores-versão, i.e, como se pode introduzir e remover novos elementos de forma eficiente e sem exigir coordenação global, foi estudada em vários sistemas [5, 140].

No sistema DOORS, cada sequência de operações é identificada no servidor com um relógio lógico que permite definir uma ordem total que respeita a causalidade. A cada sequência de operações é ainda adicionado, no cliente, um vector-versão que identifica o estado do objecto no qual foram produzidas as modficações. Como este vector-versão apenas reflecte as operações recebidas nos servidores, apenas permite traçar de forma exacta a dependência com as operações já executadas num servidor. Nas aplicações desenvolvidas, esta aproximação básica revelou-se suficiente. No entanto, este problema e possíveis soluções foram discutidos com maior detalhe na secção 4.1.1.

Quando se pretende reconciliar um conjunto de objectos, podem usar-se as técnicas anteriores, associadas ao conjunto de objectos (em vez de associadas a cada um dos objectos) para detectar modificações num nível de granularidade superior. Esta técnica é usada, por exemplo, em sistemas de ficheiros [107, 86] e sistema de bases de dados [137].

11.1.8 Propagação das modificações: propagar como?

A propagação das modificações entre as várias réplicas de um sistema pode ser dividido em dois subproblemas: a propagação entre as réplicas principais e a propagação entre as réplicas principais e secundárias. Dizemos que uma réplica principal está presente num servidor e uma réplica secundária está

[3]No caso dos relógios físicos é necessários manter os relógios sincronizados.

[4]Note que apenas é necessário manter uma entrada no vector para cada réplica em que são produzidas alterações. Assim, se se assumir que em cada momento apenas existe um pequeno número de réplicas em que são produzidas alterações, mas que a identidade das réplicas varia ao longo do tempo, o problema principal consiste em efectuar a reciclagem automática (*garbage-collection*) das referências às réplicas que já não são necessárias.

presente num cliente.

Primeiro, abordamos o problema da propagação entre réplicas principais. Nos sistemas de replicação síncrona [114], as modificações executadas numa réplica são propagadas imediatamente de forma síncrona para todas (ou para um quorum de) as outras réplicas. Esta solução tem dificuldade em lidar com as falhas e apenas pode ser usada com um número reduzido de réplicas. Para que o sistema suporte um elevado número de elementos é necessário que a propagação das modificações seja efectuada de forma assíncrona (como acontece sempre nos sistema de replicação optimista).

Uma aproximação simples consiste na utilização dum esquema *lazy master*, no qual apenas um servidor recebe as modificações que propaga assincronamente para as outras réplicas. Outra aproximação consiste na utilização de técnicas de propagação epidémica em que uma réplica propaga todas as modificações conhecidas independentemente da réplica em que tiveram origem. Vários algoritmos e sistemas usam esta aproximação [38, 57, 84, 101, 66, 122], nos quais se podem encontrar várias soluções para a escolha do parceiro e momento da sessão de propagação epidémica (topologias fixas – anel, árvore, etc. – ou variáveis – o parceiro é escolhido aleatoriamente), para determinar quais as modificações que devem ser propagadas e para o protocolo de propagação (unidireccional ou bidireccional). A solução ideal para cada sistema depende da topologia da rede de comunicações e da estratégia de reconciliação implementada.

No sistema DOORS, a propagação é epidémica uni ou bidireccional e usam-se vectores-versão para determinar de forma exacta quais as operações que necessitam de ser propagadas. A topologia, momento das sessões de propagação e *coobjectos* envolvidos, pode ser definida pelo administrador do sistema de forma específica para cada volume.

Na propagação das modificações de uma réplica secundária para uma réplica primária existem, pelo menos, três possibilidades. Primeiro, o cliente entrega a modificação a uma só réplica principal. Esta é a aproximação usual (adoptada nos sistemas implementados nesta dissertação). Segundo, o cliente propaga de forma coordenada a modificação para mais do que uma réplica. Por exemplo, no sistema Coda [87] um cliente instala uma nova versão de um ficheiro em vários servidores simultaneamente. Esta aproximação também pode ser usada para tolerar falhas bizantinas [28]. Terceiro, um cliente propaga de forma não coordenada uma modificação para mais do que uma réplica. Neste caso, é, em geral, necessário detectar a igualdade das modificações recorrendo a informação criada no cliente. Por exemplo, o sistema *lazy replication* [93] mantém identificadores únicos para cada operação.

Quando as réplicas principais são modificadas, os clientes têm de invalidar a cópias locais e obter novas cópias. Para tal, existem duas aproximações. Primeiro, o cliente verifica (periodicamente ou quando acede aos dados) se a sua cópia permanece actual. Segundo, o servidor informa os clientes das modificações. Esta informação pode conter os novos dados ou apenas informação sobre a sua modificação

e ser propagado individualmente para cada cliente [107] ou disseminado através dum canal (físico) de difusão de mensagens [14]. Este problema está fora do âmbito do trabalho desta dissertação, tendo-se implementado a solução mais simples nos protótipos do sistema DOORS e Mobisnap: o cliente verifica periodicamente a validade das cópias locais.

11.1.9 Reconciliação

A reconciliação consiste em unificar as modificações executadas sem coordenação em duas (ou mais) réplicas de um objecto. Este mecanismo elementar é usado num protocolo de reconciliação, o qual deve, em geral, garantir a convergência final das várias réplicas, i.e., garantir que após um período de tempo finito após a última modificação executada, as várias réplicas são iguais. De seguida abordam-se as técnicas de reconciliação elementares baseadas na utilização do estado das várias réplicas e na propagação das operações efectuadas.

11.1.9.1 Utilização do estado das réplicas

Nesta subsecção abordam-se as técnicas de reconciliação baseadas na utilização do estado das várias réplicas. Estas técnicas de reconciliação determinam, em cada réplica, qual o novo estado da réplica que resulta da unificação do estado de duas réplicas.

O primeiro tipo de aproximação consiste em, dado um conjunto de réplicas divergentes, seleccionar uma réplica como resultado da reconciliação. Usando a *regra de escrita de Thomas* [163], cada réplica tem associada uma estampilha temporal (sobre as quais se define uma ordem total) atribuída aquando da sua mais recente modificação. O resultado da reconciliação consiste em seleccionar a réplica com maior estampilha temporal. Vários sistemas [80, 155] usam esta aproximação. Uma estratégia alternativa consiste em permitir ao utilizador seleccionar qual o valor que deve ser mantido. Por exemplo, na sincronização de PDAs [117] é usual poder seleccionar-se qual o valor a manter: o valor do computador fixo ou o valor do dispositivo portátil. A verificação da validade das transacções usando os conjuntos de leitura e escrita [60] pode ser considerada uma extensão desta aproximação, em que se usam objectos de reduzidas dimensões, se seleccionam as modificações mais antigas e se detectam adicionalmente conflitos leitura/escrita.

O segundo tipo de aproximação que vamos considerar consiste na criação e manutenção de versões: duas réplicas de um objecto modificadas concorrentemente dão origem a duas versões, mantidas em cada uma das réplicas. Os utilizadores podem, posteriormente, unificar essas versões. Deve notar-se que, neste caso, a existência de várias versões de um objecto não é considerado um erro, decorrendo do modo de funcionamento normal do sistema. O sistema Lotus Notes [101] é um exemplo típico do uso desta aproximação.

O terceiro tipo de aproximação consiste em permitir a definição de funções que unificam as modificações concorrentes com base no estado dos objectos. Essas funções podem usar apenas o estado actual do objecto nas réplicas a sincronizar [87, 66, 115] ou podem usar versões anteriores [24]. A utilização de versões anteriores permite (tentar) inferir as modificações executadas em cada uma das réplicas (e usar essa informação no processo de reconciliação), mas requer a manutenção da história das modificações.

A remoção de um objecto é tratada de forma especial nestas aproximações. Assim, para cada objecto removido, mantém-se informação sobre a remoção (*death certificate* [38])[5].

11.1.9.2 Utilização das operações executadas

Nesta subsecção abordam-se as técnicas de reconciliação baseadas na utilização das operações executadas. Estas técnicas de reconciliação determinam, em cada réplica, o modo como se executam as operações recebidas directamente dos utilizadores e das outras réplicas.

Neste tipo de soluções, a aproximação base consiste em executar todas as operações por uma determinada ordem nas várias réplicas (por exemplo, ordem total, ordem causal ou sem ordem). Para tal, cada operação é, geralmente, identificada por um relógio escalar ou por um vector-versão. Na sua versão mais simples, o algoritmo de reconciliação executa em cada réplica e determina quais as operações que, em cada momento, podem ser executadas respeitando a ordem definida.

Na secção 4.1 já se discutiram as técnicas de implementação dos diferentes tipos de ordenação usando informação sintáctica (relógios escalares ou vectores-versão) e as suas propriedades. Relativamente à ordem total, existem várias aproximações possíveis: utilizar uma réplica primária responsável por ordenar as operações [161, 143]; usar o relógio associado a cada operação para ordenar a operação e informação sobre o estado das outras réplicas para garantir a estabilidade da ordem [57]; utilizar protocolos de eleição para determinar, em passos sucessivos, quais as operações a executar em cada momento [82]. Relativamente às outras ordens, cada réplica pode decidir independentemente quais as operações que pode executar usando apenas a informação associada a cada operação.

A execução simples de todas as operações por uma determinada ordem tem várias limitações.

Primeiro, podem existir situações em que ainda não é possível determinar a ordem final de uma operação. Neste caso, pode adoptar-se uma aproximação pessimista em que a operação não é executada enquanto não for possível determinar a sua ordem. Assim, o estado do objecto pode não reflectir todas as operações conhecidas.

Uma segunda alternativa consiste em adoptar uma aproximação optimista e executar a operação. Neste caso, o estado do objecto é provisório porque a ordem de execução pode ser incorrecta. Se a

[5] Note-se que remover apenas o objecto na réplica a leva a ambiguidade aquando da sincronização com uma réplica b, sendo impossível determinar se o objecto foi criado em b ou removido em a.

ordem de execução for incorrecta é necessário corrigir o problema. Uma solução genérica para este problema consiste em desfazer as operações executadas fora de ordem e executá-las novamente na ordem correcta. Esta técnica é conhecida por *desfazer-refazer* (*undo-redo*) [17, 81]. Alternativamente, é possível transformar a operação a executar de forma a que o resultado da execução seja equivalente ao da ordem definida. Esta técnica é conhecida por transformação de operações [46, 159]. Quando duas operações são comutativas, o resultado é independentemente da ordem de execução. Esta informação pode ser usada para optimizar o algoritmo de ordenação [170, 135].

Para lidar com a existência de operações para as quais não se pode garantir a sua ordem, existem sistemas que mantêm duas versões de cada objecto [161]. Uma versão reflecte todas as operações conhecidas e outra reflecte apenas as operações para as quais é possível determinar a ordem final de execução. Esta aproximação é usado nos clientes do sistema Mobisnap e pode ser usada no sistema DOORS.

Uma segunda limitação apresentada pela execução das operações submetidas concorrentemente por uma dada ordem é que esta aproximação não lida directamente com as consequências da submissão concorrente de operações, i.e., com a existência de operações que interferem entre si. Este facto leva a que o valor do objecto quando a operação é executada seja diferente do valor observado aquando da submissão da operação. Para lidar com este problema foram propostas várias aproximações, entre as quais:

- Definir apenas operações comutativas. Neste caso, as operações não interferem entre si e podem ser executadas por qualquer ordem.

- Guardar os valores lidos (ou conjunto de leitura) durante a execução provisória no cliente. Se, aquando da execução final da operação, os valores lidos forem diferentes, deve-se executar uma função de resolução de conflitos especificada com a operação [123].

- Delegar no código das operações o tratamento dos problemas relacionados com a execução concorrente das operações. As soluções definidas podem recorrer à informação semântica associada ao tipo de dados e à operação executada. Vários sistemas usam esta aproximação [79, 161]. O sistema Bayou [161] executa uma solução particular desta aproximação, na qual uma operação de escrita é composta pela verificação de uma pré-condição (*dependency check*), por uma modificação e por um procedimento que gera uma modificação alternativa quando a pré-condição não é verdadeira aquando da execução final da operação (*merge procedure*).

- Usar regras para definir o resultado de duas operações concorrentes. No sistema Sync [108], os programadores podem definir o resultado da execução de duas operações concorrentes. Este resultado pode ser a execução das duas operações pela ordem indicada, de apenas uma operação,

de uma nova operação ou de nenhuma operação. O sistema usa estas regras para tratar as operações executadas concorrentemente (e definir a sequência final de operações a executar). Uma característica interessante deste sistema é a possibilidade de as regras de resolução de conflitos de um objecto poderem ser definidas como a composição das regras de resolução dos objectos que o compõem. Uma aproximação semelhante foi usada para um sistema de ficheiros [138]. Neste caso, um sistema de prova, baseado num conjunto de regras algébricas que definem o resultado da combinação de duas operações executadas concorrentemente, é usado para produzir uma ordem canónica. As operações com conflitos que não podem ser resolvidos não são incluídas na sequência de operações a executar (os utilizadores podem posteriormente resolver estes conflitos manualmente).

- Optimizar a sequência de operações que podem ser executadas, ordenando as operações de acordo com as suas propriedades/efeitos. Em [123], os autores propõem que as transacções executadas provisoriamente num cliente móvel sejam integradas numa qualquer posição da história da réplica principal (implementando uma forma mais fraca de *snapshot isolation*). O algoritmo mantém uma sequência de versões da base de dados e integra uma transacção nesta sequência (criando uma nova versão da base de dados) de forma óptima. Para tal, cada transacção inclui uma função de resolução de conflitos (que especifica o resultado da execução da transacção numa dada versão da base de dados) e uma função de custo (que especifica o custo de executar a transacção numa dada versão da base de dados).

  No sistema IceCube [85], a ordenação das operações é vista como um problema de optimização, no qual se procura maximizar as operações que podem ser executadas com sucesso usando informação semântica associada às operações. O sistema SqlIceCube, apresentado nesta dissertação, estende esta aproximação através da extracção automática de informação semântica a partir do código das transacções (usando técnicas de análise estática de programas [49]).

- Transformar as operações de forma a preservar as intenções dos utilizadores. As operações submetidas concorrentemente têm de ser executadas sequencialmente, pelo que, quando se executa uma operação, o estado do objecto pode ser diferente do estado observado aquando da submissão da operação. A transformação de operações [46, 159, 118] é usada para modificar as operações de forma a que o resultado da sua execução seja o esperado aquando da sua submissão. Esta técnica é usada geralmente na edição síncrona de documentos e folhas de cálculo (e parece especialmente indicada para lidar com a modificação de parâmetros que indicam a posição de um elemento que pode ser modificada através de operações de inserção e remoção). Convém realçar que, como se referiu anteriormente, a transformação de operações pode ser usada igualmente para permitir a

execução das operações por ordem diferente nas várias réplicas.

Nenhuma das soluções propostas é óptima em todas as situações. Assim, no sistema DOORS, o *framework* de componentes permite reutilizar diferentes estratégias de reconciliação. Cada *coobjecto* pode seleccionar a solução mais adequada, i.e., aquela que permite implementar a estratégia de resolução de conflitos pretendida de forma mais simples.

Independentemente da estratégia de reconciliação escolhida no sistema DOORS ou da integração do sistema SqlIceCube no sistema Mobisnap, a execução final das operações no(s) servidor(es) permite que as operações lidem de forma específica com os problemas da execução independente de operações, por exemplo, definindo regras de detecção e resolução de conflitos.

11.1.10 Prevenção de conflitos

A replicação optimista pode levar à submissão concorrente de modificações que actuam sobre os mesmos objectos. Nas subsecções anteriores apresentaram-se técnicas de reconciliação que permitem unificar as modificações executadas concorrentemente. Devido à possibilidade de as modificações produzidas serem conflituantes, apenas é possível determinar o resultado final de uma operação após o processo de reconciliação. Nesta secção abordam-se técnicas que permitem evitar os conflitos e comparam-se estas técnicas com as implementadas no sistema Mobisnap.

Os trincos (*locks*) [60] são a técnica tradicional para evitar conflitos, permitindo que apenas um utilizador aceda a um objecto em cada momento. No entanto, num ambiente de computação móvel, no qual os dispositivos móveis podem permanecer desconectados durante longos períodos de tempo, esta técnica pode impor restrições inaceitáveis à concorrência em objectos partilhados por vários utilizadores: um dispositivo desconectado impede qualquer acesso a um objecto para o qual possua um trinco.

As versões fracas de trincos (por exemplo, *tickle locks* [64], trincos optimistas [61], trincos probabilísticos [63]) permitem que se aceda a um objecto sem possuir uma garantia de exclusividade de acesso. Estas técnicas optimistas foram propostas para ambientes com boa conectividade, nos quais fornecem uma razoável probabilidade de acesso exclusivo. No entanto, em situações de desconexão, as garantias que podem oferecer são praticamente nulas.

No sistema Prospero [42], estendeu-se a noção de trinco de forma a restringir a divergência entre vários fluxos de actividade. Para tal, os utilizadores especificam uma *promessa* relativa às operações que executam e obtém *garantias* relativas ao estado dos objectos. Estas garantias podem ser usadas para garantir a inexistência de conflitos. Ao contrário dos trincos, as promessas e garantias são especificadas (indirectamente) em termos da execução de uma dada operação. Por exemplo, a promessa de não modificar a estrutura de um documento está associada à não execução da operação de modificação da estrutura do documento. Assim, o utilizador pode obter a garantia da manutenção da estrutura do documento se

se impedir a execução da operação de modificação da estrutura.

O modelo de divisão (*escrow*) [111] permite dividir o valor de uma variável que represente um recurso divisível (por exemplo, a existência –*stock*– de um qualquer produto) sujeita a uma dada restrição (por exemplo, o valor da existência do produto deve ser maior ou igual a zero) por um conjunto de entidades (transacções, réplicas). Cada entidade pode garantir o sucesso das operações que apenas usam a parte dos recursos que lhe foram atribuídos (garantindo assim que a restrição global permanece válida). As operações executadas sem coordenação podem ser unificadas de forma simples porque apenas podem ser executadas operações comutativas (adição e subtracção) sobre as variáveis partilhadas. Para manipular estas variáveis, foram introduzidas funções especiais. Este modelo pode ser utilizado para suportar uma aplicação de suporte a uma força de vendas num ambiente de computação móvel [90].

O modelo de divisão pode ser generalizado explorando a semântica dos objectos [168]. A ideia consiste em dividir objectos grandes e complexos em fragmentos menores que obedecem a certas restrições. Cada fragmento pode ser modificado independentemente, desde que as restrições associadas não sejam violadas. A unificação dos fragmentos pode ser efectuada usando a informação semântica disponível.

Esta aproximação e a aproximação usada no sistema Prospero [42] apenas podem ser usadas de forma adequada em alguns tipos de dados e são mais apropriadas para sistemas de bases de dados orientadas para os objectos, nos quais as operações que podem ser executadas para modificar o tipo de dados são complexas e estão pré-definidas.

O sistema Mobisnap define um modelo de prevenção de conflitos baseado na utilização de reservas. Este modelo combina e implementa um conjunto de técnicas de prevenção de conflitos (entre as quais a utilização de trincos de pequena granularidade e do modelo da divisão). Como se mostrou na secção 8.5, a combinação de diferentes técnicas é fundamental para a sua utilidade num sistema genérico de bases de dados relacionais. O sistema Mobisnap verifica, de forma transparente, se as reservas disponíveis são suficientes para garantir uma transacção móvel. Esta aproximação permite aos programadores definirem as transacções da forma habitual (i.e., sem usarem funções especiais para manipular elementos de dados reservados) e utilizar todas as reservas disponíveis (e não apenas aquelas que tenham sido especificadas no código da transacção).

Além de evitar os conflitos é possível diminuir a probabilidade da sua ocorrência usando técnicas de controlo de divergência. Vários métodos de definir a divergência entre réplicas foram propostos, assim como algoritmos para limitar esta divergência [6, 135, 89]. O sistema TACT [173] permite controlar a divergência entre um conjunto de réplicas usando várias métricas de divergência. Esta aproximação permite às aplicações definirem a divergência máxima aceitável.

Os mecanismos de controlo de divergência não podem ser usados para garantir o resultado de uma operação, mas apenas para diminuir a probabilidade da ocorrência de um conflito que impeça a sua

execução. Assim, estes mecanismos podem ser considerados como complementares dos mecanismos de prevenção de conflitos.

11.2 Informação sobre a evolução dos dados

A informação sobre a evolução dos dados e da actividade cooperativa permite a um utilizador conhecer, não só as modificações produzidas por outros utilizadores, mas também o resultado final das suas modificações. Na generalidade dos sistemas de gestão de dados para ambientes de computação móvel, o problema da criação e tratamento desta informação é ignorado.

A importância desta informação para o sucesso da actividade cooperativa é identificada por vários autores [43, 65, 99]. Em geral, as aplicações cooperativas síncronas providenciam informação sobre a evolução dos dados e sobre a actividade cooperativa através de pequenos elementos na interface gráfica das aplicações. Por exemplo, nos editores cooperativos síncronos, além de se apresentar a lista dos utilizadores que estão a modificar o documento, é usual apresentar as modificações produzidas por diferentes utilizadores usando diferentes cores. O sistema de suporte à criação de aplicações cooperativas síncronas Groupkit [145] inclui um conjunto de pequenos elementos da interface gráfica que podem ser usados nas aplicações para fornecer informação sobre as actividades dos outros utilizadores.

Apesar de se ter identificado a importância desta informação, a maioria dos sistemas de suporte ao trabalho cooperativo assíncrono (por exemplo, Lotus Notes [80], Prospero [42], Sync [108]) não integram nenhum mecanismo de suporte à manipulação desta informação. Algumas excepções existem.

No sistema CVS [24], um utilizador pode incluir com cada nova versão uma descrição das modificações executadas. Esta informação é mantida com cada versão e permite aos utilizadores conhecer as modificações produzidas apenas quando acedem aos dados. Uma aproximação semelhante é usualmente adoptada nos sistemas de espaço de trabalho partilhado (*shared workspace*). Por exemplo, o sistema BSCW [16] permite aos utilizadores observar os eventos executados sobre os documentos do espaço partilhado. Estes eventos correspondem às operações que podem ser executadas sobre um documento (por exemplo, editar o documento). Assim, estes eventos têm uma elevada granularidade (apenas se específica que um documento foi modificado e não as modificações que sofreu) e incluem não só as operações de modificação, mas todas as operações executadas sobre o objecto.

O sistema de edição cooperativa assíncrona Quilt [97] permite que um utilizador peça para ser notificado quando são efectuadas modificações a um documento.

No sistema *Placeless Documentos* [44], usou-se o mecanismo de propriedades activas (que permite especificar acções a executar quando um objecto é acedido) para fornecer informação sobre a evolução dos dados em algumas aplicações.

Os sistemas apresentados nesta dissertação integram o tratamento desta informação. A informação

sobre a evolução dos dados é criada no código das operações através de funções definidas para o efeito. O tratamento desta informação é integrado no sistema, permitindo manter esta informação com os dados ou enviar notificações para os utilizadores usando diferentes transportes.

11.3 Integração de sessões síncronas

O problemas da integração de sessões de trabalho síncrono e assíncrono tem sido abordado em vários sistemas de suporte ao trabalho cooperativo. Várias aproximações têm sido propostas.

Uma possível aproximação consiste em gravar os eventos produzidos numa sessão cooperativa. Quando um utilizador acede à sessão cooperativa num momento posterior, ele pode observar as acções executadas reproduzindo os eventos armazenados [77, 98].

Greenberg et al. [62] definem um espaço de dados partilhado baseado na noção de quarto (*room*), onde os utilizadores podem guardar objectos partilhados. As aplicações executam no âmbito de um quarto, podendo um utilizador ligar-se ao servidor central para observar e modificar o estado do quarto (utilizando as aplicações que executam nesse quarto). Os utilizadores que acedem ao mesmo quarto simultaneamente podem modificar um objecto de forma síncrona. As modificações assíncronas resultam do acesso a um quarto em diferentes momentos.

O sistema Sepia [67] permite a edição de documentos de hipertexto de forma síncrona e assíncrona usando uma aproximação semelhante.

Qu et al. [136] descrevem um ambiente de educação à distância que combina o trabalho síncrono e assíncrono. Um repositório WebDav assíncrono mantém os objectos usando um modelo simples de trincos ou *check-in/check-out*. Um objecto pode ser manipulado sincronamente (usando o sistema Lotus Sametime). Para tal, o objecto deve ser transferido do repositório assíncrono.

As aproximações anteriores não são apropriadas para ambientes de computação móvel porque requerem o acesso a um servidor central. Adicionalmente, as primeiras soluções apenas permitem a existência de um único fluxo de actividade (com excepção de períodos de tempo muito curtos durante as sessões síncronas). Assim, a interacção assíncrona restringe-se à manipulação da cópia única de um objecto em diferentes momentos. No ambiente de educação à distância apresentado [136] é possível existirem múltiplos fluxos de actividade. No entanto, os conflitos são resolvidos criando múltiplas versões do objecto, i.e., não é possível integrar automaticamente as modificações executadas durante estas sessões.

O sistema Prospero [42] permite a manipulação concorrente de cópias de um mesmo objecto. Este sistema é baseado no conceito de *stream*, no qual se guardam as operações executadas em cada fluxo de actividade. A implementação de aplicações síncronas e assíncronas é possível usando diferentes frequências da sincronização dos *streams*. Os *streams* podem ser usados para integrar sessões síncronas e assíncronas fazendo variar a frequência de sincronização, mas apenas quando é possível usar as

mesmas operações e mecanismos de controlo de concorrência. Uma aproximação semelhante é proposta em [154].

O sistema SAMS [105] permite a edição de documentos estruturados em modo síncrono, assíncrono e multi-síncrono. A diferença entre os vários modos reside no período de tempo que cada elemento do sistema armazena as operações executadas antes de as propagar para os outros elementos do sistema. A integração, em cada réplica local, das operações executadas concorrentemente é efectuada de forma idêntica em todos os modos através da reexecução das operações e recorrendo a um mecanismo de transformação de operações.

No entanto, como se discutiu na secção 4.4.1, existem aplicações que necessitam de usar operações e mecanismos de reconciliação diferentes durante as sessões síncronas e assíncronas. No âmbito do sistema DOORS, propôs-se uma solução genérica para a integração de sessões síncronas e assíncronas com essas propriedades.

Capítulo 12

Conclusões

A generalização da utilização de computadores portáteis e de comunicações sem fios criou um novo ambiente de computação com características próprias. O sistema de gestão de dados é um dos elementos que se deve adaptar às novas características para permitir a partilha de dados entre múltiplos utilizadores. Esta dissertação descreve um conjunto de princípios e técnicas usados em dois sistema de gestão de dados para permitir a partilha de informação em ambientes de computação móvel.

Este capítulo apresenta um sumário das principais soluções de gestão de dados apresentadas nesta dissertação e direcções de trabalho futuro.

12.1 Sumário

Nesta dissertação estudou-se a problemática da gestão de dados partilhados em ambientes de computação móvel. As soluções propostas adoptam como princípio fundamental a exploração da informação semântica associada aos dados e às acções dos utilizadores.

Nesta dissertação identificou-se, ainda, um conjunto de princípios gerais que, explorando a informação semântica disponível, permitem a gestão de dados partilhados por múltiplos utilizadores. Estes princípios gerais foram desenvolvidos e validados durante o desenho, implementação e teste de dois sistemas de gestão de dados partilhados para ambientes de computação móvel. Nesta dissertação descrevem-se as técnicas específicas usadas para colocar esses princípios gerais em prática.

Os sistemas de gestão de dados criados nesta dissertação fornecem uma elevada disponibilidade de leitura e escrita através da combinação das seguintes técnicas: replicação parcial dos dados nos dispositivos móveis; modelo de replicação optimista; e reconciliação dependente da situação.

Relativamente ao modelo de replicação optimista, os sistemas apresentados nesta dissertação incluem uma nova característica: a possibilidade de submeter operações sobre dados não replicados localmente, designada de invocação cega. Ao contrário da aproximação habitual, em que se impede qualquer acesso

aos dados não replicados localmente, esta aproximação permite ao utilizador decidir as situações em que pode produzir contribuições úteis através da submissão de operações de modificação dos dados. Como estas operações são sujeitas ao mecanismo de reconciliação do sistema, garante-se a sua integração no estado comum dos dados de forma correcta.

Os sistemas de gestão de dados apresentados também integram mecanismos de tratamento da informação sobre a evolução dos dados. Este problema é geralmente negligenciado pelos sistemas de gestão de dados, apesar desta informação ter sido identificada como importante para o sucesso de uma actividade cooperativa [43, 65, 99]. O mecanismo implementado é igualmente importante na notificação do resultado final das operações executadas por cada utilizador, o qual apenas é conhecido após a execução final do mecanismo de reconciliação.

A implementação destas características comuns em cada um dos sistemas de gestão de dados apresentados exigiu a sua adaptação e possível extensão de acordo com o modelo de dados usado em cada um dos sistemas. Por exemplo, a implementação da invocação cega no repositório de objectos DOORS, inclui a integração de um mecanismo de criação de cópias de substituição para que se possa observar o resultado provisório das operações submetidas. Além destas características comuns, os sistemas de gestão de dados apresentam igualmente um conjunto de características específicas, das quais se resumem de seguida as mais importantes.

O DOORS é um repositório de objectos desenhado para suportar a criação de aplicações de trabalho em grupo tipicamente assíncrono. Uma das principais características deste sistema consiste na possibilidade de definir soluções específicas de gestão de dados partilhados usando diferentes estratégias em diferentes tipos de dados. Para tal, definiu-se o *framework* de componentes DOORS, que decompõe o funcionamento de um objecto em vários componentes. Cada componente lida com um aspecto específico da gestão de dados partilhados, entre os quais se destacam, a reconciliação e o tratamento da informação sobre a evolução dos dados e da actividade cooperativa. Os programadores podem criar uma solução global de gestão de dados com base num conjunto de soluções pré-definidas para cada um dos problemas identificados. O *framework* de componentes definido é uma contribuição original do sistema DOORS.

O sistema DOORS propõe igualmente um modelo de integração de sessões de trabalho síncrono de curta duração no âmbito de uma sessão assíncrona de longa duração, no qual se podem usar as técnicas de gestão de dados (por exemplo, reconciliação) adequadas a cada um dos modos. Ao contrário da generalidade dos sistemas de suporte ao trabalho cooperativo, este modelo lida explicitamente com o problema da diferença de granularidade entre as operações síncronas e assíncronas em algumas aplicações.

O Mobisnap é um sistema de gestão de bases de dados relacional para ambientes de computação móvel. Neste sistema, os utilizadores modificam a base de dados através da submissão de pequenos programas escritos em PL/SQL, a que se chamam transacções móveis. O código das transacções móveis

é executado no cliente e no servidor, o que permite definir regras de detecção e resolução de conflitos que explorem a informação semântica associada à operação (e às intenções do utilizador).

O Mobisnap integra um mecanismo de prevenção de conflitos baseado na utilização de reservas. Este mecanismo permite garantir o resultado das transacções móveis de forma independente. Relativamente a propostas anteriores, o mecanismo de reservas apresenta as seguintes contribuições. Primeiro, integra vários tipos de reservas, o que se demonstrou ser fundamental para garantir o resultado de um grande número de transacções. Segundo, o sistema verifica de forma transparente se é possível garantir o resultado de uma transacção móvel com base no seu código e nas reservas disponíveis localmente. Esta propriedade permite explorar todas as reservas para garantir qualquer transacção sem que exista necessidade de modificar a forma como estas são escritas. Terceiro, implementa o mecanismo de reservas num sistema baseado em SQL e de forma integrada com o modelo de execução das transacções móveis, permitindo definir um modelo integrado que combina técnicas de prevenção e resolução de conflitos.

Resultados obtidos por simulação mostram que o mecanismo de reservas permite suportar a operação independente numa aplicação de suporte a uma força de vendas móvel. Mesmo usando estratégias simples na obtenção de reservas, é possível garantir, de forma independente, mais de 80% das encomendas submetidas no pior cenário estudado, em que não existe previsão das encomendas a receber e os vendedores não obtém reservas inicialmente.

Nesta dissertação descreve-se ainda um novo sistema genérico de reconciliação de transacções móveis, o sistema SqlIceCube. Este sistema é usado quando é necessário executar um conjunto de transacções móveis no servidor do sistema Mobisnap. No SqlIceCube, a reconciliação é vista como um problema de optimização que consiste em determinar a melhor sequência de transacções móveis que podem ser executadas com sucesso. Em particular, a reconciliação é vista como um problema de planeamento em vez de um problema de resolução de restrições (binárias), como acontecia no sistema IceCube [134], seu antecessor. Esta aproximação permite lidar de forma correcta com pares de transacções cujos efeitos se anulam. Esta nova aproximação obrigou a definir um novo conjunto de relações para exprimir a informação semântica em problemas de reconciliação. Este sistema inclui ainda um mecanismo original de extracção automática de informação semântica a partir do código das transacções móveis. Este mecanismo é o primeiro a inferir automaticamente a informação necessária à reconciliação a partir do código das operações e é fundamental para permitir a submissão de transacções móveis genéricas.

A implementação do sistema Mobisnap descrita nesta dissertação, como uma camada de sistema intermédia (*middleware*), representa uma aproximação evolutiva à computação móvel. Assim, as aplicações que executam o sistema Mobsnap podem utilizar as novas funcionalidades enquanto se permite que as aplicações legadas continuem a aceder directamente à base de dados central sem nenhuma modificação. Adicionalmente, os mecanismos de reservas e reconciliação integrados no sistema são totalmente

independentes, o que permite a sua utilização isolada noutros sistemas de suporte à computação móvel.

12.2 Trabalho futuro

O trabalho realizado nesta dissertação, além das contribuições descritas anteriormente, revelou um conjunto de possíveis direcções para trabalho futuro. Nesta secção apresentam-se algumas dessas direcções (algumas das quais já foram indicadas ao longo da dissertação).

No âmbito do sistema DOORS, está-se a explorar a possibilidade de usar um suporte linguístico mais adequado na definição de novos *coobjectos*, com base na linguagem ComponentJ [153]. Esta aproximação deve permitir uma maior clareza na criação de *coobjectos*, com a separação da definição e composição dos componentes. Adicionalmente, deve permitir definir esqueletos de *coobjectos* que agrupam um conjunto de componentes para implementar uma dada solução de gestão de dados. Este trabalho está a ser desenvolvido no âmbito do projecto Databricks[36].

Relativamente ao sistema de reservas implementado no sistema Mobisnap, é necessário avaliar o seu desempenho em diferentes cenários e estudar a possibilidade de estender o conjunto de reservas propostas. Uma direcção que se pretende explorar é a possibilidade de definir reservas que forneçam garantias probabilísticas sobre o estado da base de dados, as quais devem permitir obter garantias igualmente probabilísticas sobre o resultado final das transacções móveis.

Adicionalmente, é necessário investigar o impacto da implementação do sistema de reservas no servidor central da base de dados (no cliente o desempenho não é um problema). No âmbito desta avaliação deve considerar-se igualmente o custo de uma implementação integrada no sistema de base de dados.

Outro problema importante é a definição de algoritmos genéricos de obtenção de reservas. Este problema pode ser visto como uma extensão do problema do replicação antecipada, no qual não se pretende apenas identificar os dados que vão ser acedidos, mas também o modo como vão ser modificados. Um solução a explorar consiste em inferir as necessidades de cada utilizador com base na combinação do seu comportamento anterior e de indicações explicitamente fornecidas pelo utilizador.

O mecanismo de reconciliação de múltiplas transacções do sistema Mobisnap está ainda sob desenvolvimento. Em particular, continua a analisar-se a melhor forma de implementar algumas das extensões referidas na secção 10.5.4.

Uma última direcção de trabalho futuro a considerar é a integração de um mecanismo de controlo de divergência no sistema Mobisnap (estendendo, por exemplo, ideias apresentadas anteriormente [6, 173]). Este mecanismo deve permitir definir métricas de divergência entre a cópia local e central de uma base de dados adequadas a diferentes aplicações. Com base nestas métricas, o sistema deve fornecer mecanismos que permitam controlar a divergência admissível. A integração destas métricas com o sistema de reservas constitui um desafio adicional.

Apêndice A

DOORS

Este apêndice apresenta, de forma detalhada, a implementação executada no protótipo do sistema DOORS do mecanismo de gestão da filiação dos servidores que replicam um volume e o mecanismo de sincronização epidémica das réplicas dos volumes. Estes mecanismos são essenciais para o funcionamento do núcleo do sistema DOORS e garantem que, em todos os servidores correctos que replicam um volume, serão conhecidas todas as operações submetidas em todos os *coobjectos* do volume.

A.1 Filiação

A.1.1 *Coobjecto* de filiação

O *coobjecto* de filiação mantém, em cada volume, a informação sobre o conjunto de servidores que replica o volume. Este *coobjecto* define três operações de modificação, executadas em consequência dos protocolos estabelecidos para o início e cessação da replicação de um volume: inserção de um novo servidor; remoção de um servidor; e eliminação das referências relativas a um servidor anteriormente removido.

O funcionamento do *coobjecto* de filiação garante que duas réplicas que tenham conhecimento do mesmo conjunto de operações estão no mesmo estado (i.e., na mesma vista). Garante-se ainda que o identificador da vista identifica univocamente uma vista, i.e., a cada conjunto distinto/idêntico de operações corresponde um identificador de vista distinto/idêntico. Estas propriedades são obtidas como se explica de seguida.

Cada servidor i que pertence ao grupo de replicadores tem associado um identificador único srv_{vol}[1]. As operações do *coobjecto* de filiação, submetidas num dado servidor, são identificadas univocamente

[1]Este identificador é gerado quando o servidor solicita a inserção no grupo de replicadores do volume e identifica univocamente uma presença do servidor no grupo de replicadores — este identificador é obtido como $(srv_{global}, id_{unico})$ a partir do identificador global do servidor, srv_{global}.

com um par (srv_{vol}, n_seq), em que n_seq é um número inteiro maior do que o n_seq atribuído a qualquer outra operação conhecida localmente.

O estado do *coobjecto* de filiação é constituído por um conjunto, $\mathcal{M}$, de triplos $(srv_{vol}, srv_{view}, estado)$, em que srv_{vol} é o identificador único do servidor (como explicado anteriormente), srv_{view} é o identificador do servidor na vista definida pelo estado actual do *coobjecto* de filiação do volume (é um número inteiro maior do que zero) e $estado$ tem o valor $activo$, caso o servidor pertença ao grupo de replicadores, ou $removido$, caso o servidor tenha sido removido do grupo de replicadores (usa-se $\mathcal{M}^i$ para designar o valor de $\mathcal{M}$ na réplica do *coobjecto* de filiação do servidor i). O estado inicial do *coobjecto* de filiação contém apenas a informação relativa ao servidor no qual o volume foi criado ($\mathcal{M} = \{(id_{srv}, 1, activo)\}$).

Seja O o conjunto de operações reflectidas no estado do *coobjecto* de filiação. Um sumário destas operações é definido através de um vector-versão v, em que $v[i] = n_{max}$, tal que, dado $(id, i, _) \in \mathcal{M}$ (com _ a designar qualquer valor possível)), se tem $n_{max} = max(\{n : o \in O \wedge ident(o) = (id, n)\} \cup n : o \in O \wedge ident(o) = (_, n) \wedge o = insert(id))$, com $ident(o)$ o par com que a operação o foi identificada (ou seja, cada posição, i, reflecte o maior n_seq das operações executadas que foram identificadas pelo servidor com identificador i na vista actual ou o n_seq da operação que o inseriu no grupo de replicadores). Se o identificador local i não estiver atribuído a nenhum servidor $v[i] = \top$ (com $\top > n, \forall n \in \mathbb{N}$). De forma idêntica define-se o vector versão das operações que modificam a filiação, considerando apenas o conjunto das operações que têm influência directa na filiação — inserção, remoção e eliminação.

O estado do *coobjecto* de filiação num servidor é obtido considerando todas as operações conhecidas localmente para as quais não existam operações anteriores submetidas no mesmo servidor que não sejam conhecidas (ou seja, é considerado o seguinte conjunto de operações: $\{op : ident(op) = (id_i, n_seq) \wedge \neg\exists op_1 : ident(op_1) = (id_j, n_seq_1) \wedge id_i = id_j \wedge n_seq > n_seq_1\}$, com $ident(o)$ o identificador atribuído à operação o). O mecanismo de propagação epidémico de operações (o único usado para propagar operações do *coobjecto* de filiação) garante esta propriedade para todas as operações conhecidas localmente.

A ordem total pela qual estas operações são executadas em todas as cópias do *coobjecto* é a seguinte. A partir do estado inicial, as operações são executadas sequencialmente escolhendo em cada iteração a operação com menor n_seq. Entre as operações com igual n_seq, escolhe-se aquela que tenha associado o identificador de servidor com menor identificador local na vista definida actualmente.

A execução da operação de inserção de um servidor no grupo de replicadores do volume insere o trio $(id_{new}, pos_{new}, activo)$ em $\mathcal{M}$, sendo id_{new} o identificador gerado para identificar o novo servidor no grupo de replicadores e pos_{new} o menor número inteiro maior do que zero que não é usado como identificador de outro servidor na vista actual. A cópia inicial do *coobjecto* de filiação no novo servidor reflecte sempre a execução da operação que o inseriu no grupo de replicadores.

A operação de remoção de um servidor apenas altera o *estado* no trio que representa esse servidor em $\mathcal{M}$. A operação de eliminação das referências de um servidor remove o trio respectivo do conjunto $\mathcal{M}$.

A garantia da igualdade entre o estado de duas réplicas de um *coobjecto* de filiação que conheçam o mesmo conjunto de operações (propriedade de segurança) é resultado de: (1) serem executadas todas as operações conhecidas; (2) as operações serem executadas pela mesma ordem total; e (3) as operações serem deterministas.

A propagação das operações de modificação entre as várias réplicas garante que todas as réplicas do *coobjecto* de filiação evoluem para o mesmo estado comum (propriedade de evolução contínua).

O sumário das operações que modificam a filiação é usado como identificador da vista. O algoritmo optimista usado para executar as operações pode levar a que o mesmo identificador seja (temporariamente) usado em cópias diferentes para identificar servidores diferentes (por exemplo, quando dois servidores entram concorrentemente no grupo de replicadores usando patrocinadores diferentes). No entanto, como se explica de seguida, o identificador da vista identifica univocamente uma vista.

Sejam s_i e s_j as sequências de operações que modificam a filiação executadas nas cópias i e j do *coobjecto* de filiação (a partir do mesmo estado inicial). Se $s_i = s_j$, a igualdade das cópias i e j garante a igualdade dos identificados da vista. Se $s_i \neq s_i$, é possível decompor $s_i \equiv s_{comum}, op_i, sf_i$ e $s_j \equiv s_{comum}, op_j, sf_j$ em que s_{comum} é o maior prefixo (eventualmente nulo) de operações comuns (sf_i e sf_j podem ser sequências nulas e op_j pode representar a ausência de uma operação se sf_j for a sequência nula).

Vamos supor, sem perda de generalidade, que op_i é anterior a op_j na ordem total pela qual as operações são executadas, e $ident(op_i) = (srv_k, n)$. Após executar s_{comum}, tem-se $(srv_k, m, _) \in \mathcal{M}$ em ambos as cópias do *coobjecto* de filiação.

O trio associado a srv_k existe necessariamente em $\mathcal{M}$ pelas seguinte razões. Primeiro, é impossível ser conhecida uma operação submetida num servidor antes da operação de inserção porque: a cópia do *coobjecto* de filiação no novo servidor é iniciada a partir da cópia do patrocinador que já reflecte o novo servidor; as operações submetidas no novo servidor são identificadas com um *n_seq* superior ao *n_seq* da operação que o inseriu no grupo de replicadores; e as operações são propagadas entre servidores usando um mecanismo de propagação epidémica (que propaga as operações por ordem de *n_seq*). Segundo, é impossível a entrada do servidor ter sido eliminada porque uma entrada apenas é eliminada quando todas as operações relativas a esse servidor foram executadas em todos os servidores.

Assim, após a execução da sequência de operações s_i, tem-se necessariamente $v[m] \geq n$[2]. Por outro

[2] Note que, se o servidor associado a m, srv_m, tiver sido removido, se tem: $v[m] = \top > n$ se a posição não tiver sido reutilizada por outro servidor, ou $v[m] > n$ se a posição tiver sido reutilizada — porque a operação de inserção do novo elemento no grupo terá de ter sempre um *n_seq* superior ou igual ao *n_seq* da operação de remoção do servidor (ou ser-lhe-ia atribuído um diferente identificador na vista), a qual por sua vez terá de ter sempre um *n_seq* superior a n (pelo modo como as operações de remoção

lado, na cópia j, tem-se necessariamente $v[m] < n$, porque a execução de op_j, posterior a op_i, impede esta de ser executada (e a ordem definida garante que nenhuma outra operação submetida no servidor identificado por m pode ser executada). Nesta cópia, é impossível que o servidor associado a m tenha sido removido (porque o modo de funcionamento do *coobjecto* garante que uma operação de remoção apenas é executada após a execução de todas as operações submetidas nesse servidor). Desta forma, a execução de dois conjuntos diferentes de operações leva a que o identificador da vista seja igualmente diferente.

A.1.2 Protocolo local de mudança de vista

O *protocolo local de mudança de vista* é executado localmente num servidor sempre que é necessário modificar o estado do *coobjecto* de filiação. Este protocolo executa, de forma atómica, as actualizações necessárias nos identificadores dos servidores usados em todos os *coobjectos* do volume. No fim da execução do protocolo diz-se que foi instalada a vista definida pelo *coobjecto* de filiação.

O *protocolo local de mudança de vista* consiste nos seguintes passos executados no servidor i, dado um conjunto de operações a executar sobre o *coobjecto* de filiação:

1. Qualquer acesso à cópia do volume no servidor i é impedido (incluindo os acessos aos *coobjectos* pertencentes a esse volume).

2. São executadas as operações desejadas no *coobjecto* de filiação (que podem incluir operações recebidas de outro servidor). O estado final do *coobjecto* define a nova vista que vai ser instalada na cópia do volume.

3. Se um novo identificador de vista foi definido (i.e., se foi executada alguma operação que altere a filiação), todos os *coobjectos* são informados da nova vista e das mudanças necessárias para converter os identificadores antigos nos novos. Normalmente, os identificadores já existentes mantêm-se e não é necessário executar nenhuma acção nos *coobjectos*.

4. O protocolo termina e permite-se, de novo, o acesso à cópia do volume.

Se durante a execução do protocolo existir alguma falha no servidor, o protocolo volta a ser reiniciado no ponto onde falhou. As falhas na execução do protocolo são detectadas da seguinte forma. O início do protocolo é registado em memória estável, onde se guarda a data de início — após a conclusão do protocolo esta informação é removida. Quando um servidor é reiniciado, qualquer registo de início do protocolo na memória estável corresponde a uma execução não terminada. Uma falha durante o passo 2 é detectada pelo facto de o estado do *coobjecto* de filiação guardado em memória estável ter uma data

são submetidas).

anterior à de início de protocolo. A detecção de falhas durante a execução do passo 3 é efectuada de forma idêntica para cada *coobjecto*.

A.1.3 Protocolo de entrada no grupo de replicadores de um volume

O *protocolo de entrada no grupo de replicadores* de um volume controla a entrada de um novo servidor no grupo de servidores que replicam um volume. Nesta situação, é necessário actualizar o *coobjecto* de filiação e propagar um cópia do volume para o novo servidor.

O *protocolo de entrada no grupo de replicadores* de um volume v do servidor i, tendo como patrocinador o servidor j pertencente ao grupo de replicadores de v, tem os seguintes passos:

1. i comunica a j o desejo de entrar no grupo e indica-lhe o identificador único que usará id_i. Até à conclusão do protocolo, o servidor j fica impedido de participar em protocolos de remoção voluntária do grupo de replicadores.

2. O servidor j executa o protocolo local de mudança de vista no volume v usando a operação de inserção no grupo do servidor i com identificador único id_i.

3. O servidor j envia para o servidor i o estado actual da sua cópia do volume durante uma sessão de sincronização epidémica, incluindo uma cópia de todos os *coobjectos* existentes — quando uma cópia de um *coobjecto* é instanciada pela primeira vez num servidor, é executada uma operação na *interface do coobjecto* a informar o *coobjecto* dessa situação. Após a recepção do *coobjecto* de filiação, o servidor i pode iniciar imediatamente a sua operação (embora se espere pela recepção de todo o estado do volume, a menos que exista alguma falha).

Se durante a execução do protocolo existir alguma falha, o protocolo é reiniciado, reexecutando o passo em que falhou — as falhas no passo 2 são resolvidas como explicado anteriormente. As falhas durante a propagação do estado do volume são resolvidas através da execução de uma nova sessão de sincronização epidémica.

A.1.3.1 Condições suficientes para a ordenação correcta de operações usando técnicas de verificação da estabilidade da ordem

Quando um novo servidor se junta ao grupo de replicadores de um volume, o seu identificador é provisório até se garantir que a operação de inserção executada no *coobjecto* de filiação não será desfeita no futuro. Assim, quando se usam as técnicas de estabilidade descritas na secção 4.1, não se deve usar a informação relativa aos servidores que têm identificadores provisórios[3].

[3] O identificador de um servidor adicionado ao grupo de replicadores de um volume pela operação de adição com identificador (srv_{vol}, n_seq) é provisório sse essa operação de adição ainda não foi ordenada de forma definitiva na réplica do *coobjecto*

Adicionalmente, para garantir que as operações são ordenadas correctamente, em cada um dos *coobjectos*, a primeira operação submetida no novo servidor deve ter número de sequência igual ou superior a $max+2$, com max o valor do maior número de sequência que o novo servidor sabe ter sido usado em qualquer réplica.

Relativamente aos servidores conhecidos, a verificação da estabilidade garante que as operações são executadas pela sua ordem correcta. Assim, apenas seria possível executar uma operação fora de ordem, executando num qualquer servidor uma operação, op_1, de um servidor conhecido que devesse ser ordenada após uma operação, op_2, submetida num novo servidor ainda não conhecido. Como $seq(op_2) \geq max+2$, então necessariamente $seq(op_1) \geq max+2$. No entanto, para ordenar uma operação com número de sequência $max+2$, um servidor tem de receber informação de todos os servidores que não existe mais nenhuma operação com número de sequência $max+1$. Para tal, é necessário contactar o patrocinador da junção ao grupo do novo servidor (porque no momento da junção ao grupo, no patrocinador, apenas se tinham usado números de sequência inferiores ou iguais a max). Como no contacto com o patrocinador, o servidor vai ser informado da existência do novo servidor, ele tem de considerar a informação recebida do novo servidor antes de ordenar a operação op_1. Como a informação do novo servidor não pode ser usada enquanto a posição do servidor na vista local não é estável, um servidor não pode executar nenhuma operação $op : seq(op) = max+2$ enquanto a posição do novo servidor não é estável, porque necessita de usar a informação que não existe nenhuma operação nesse servidor com número de sequência igual a $max+1$. Assim, as operações com número de sequência $max+2$ apenas são executadas quando o número de sequência do novo servidor está estável, o que garante a correcção da ordenação das operações.

A.1.4 Protocolo de saída voluntária do grupo de replicadores de um volume

O *protocolo de saída voluntária do grupo de replicadores* de um volume controla a saída voluntária de um servidor do grupo de servidores que replicam um volume. Nesta situação, é necessário que não se perca nenhuma informação que se encontre apenas nesse servidor (i.e., operações relativas a *coobjectos* do volume, incluindo o *coobjecto* de filiação, que apenas sejam conhecidas nesse servidor). Adicionalmente, a filiação do grupo de replicadores deve ser actualizada e os recursos usados para replicar esse volume devem ser libertos.

Quando o servidor que pretende cessar a replicação de um volume é o único que replica o volume (de acordo com o estado do *coobjecto* de filiação nesse servidor), o servidor limita-se a libertar os recursos

de filiação, i.e., sse existe pelo menos um servidor activo, $srv2_{vol}$, do qual pode existir um operação ainda desconhecida com número de sequência n_seq_2, em que $n_seq_2 \leq n_seq$ nos casos em que o identificador de $srv2_{vol}$ na vista actual é inferior ao identificador de srv_{vol}, ou $n_seq_2 < n_seq$ nos outros casos.

associados e o volume deixa de existir. Caso contrário, o servidor deve seleccionar outro servidor que replique o volume como patrocinador para a sua acção, e efectuar o protocolo de saída do grupo.

O *protocolo de saída do grupo de replicadores* de um volume v do servidor i, tendo como patrocinador o servidor j pertencente ao grupo de replicadores, tem os seguintes passos:

1. i coloca-se no estado de *pré-removido* em relação ao volume e não executa mais nenhuma operação relativa a esse volume (i.e., não participa em sessões de sincronização nem aceita pedidos de clientes).

2. i informa j que pretende sair do grupo de replicadores do volume v e que pretende que j seja o seu patrocinador. Se j não aceitar, i deve escolher um novo patrocinador.

3. Caso j aceite ser patrocinador de i, o protocolo continua e j fica impedido de sair do grupo de replicadores do volume até que o protocolo de saída de i termine.

4. i informa todos os *coobjectos* do volume (conhecidos em i) que esse servidor vai ser removido e o patrocinador da remoção é j, usando, para tal, uma operação definida na *interface do coobjecto* (em consequência desta informação os *coobjectos* podem efectuar as operações necessárias para a continuação normal da sua operação — por exemplo, em *coobjectos* que definam uma cópia como principal, caso a cópia a remover seja a principal, pode executar-se uma operação que transfere esse estatuto para a cópia de j).

5. j informa todos os *coobjectos* do volume (conhecidos em j) que o servidor i vai ser removido, usando, para tal, uma operação definida na *interface do coobjecto* (em consequência desta informação, os *coobjectos* podem efectuar as acções necessárias ao seu correcto funcionamento).

6. i estabelece uma sessão de sincronização de todos os *coobjectos* — incluindo o *coobjecto* de filiação — com o servidor j. Nesta sessão de sincronização, i também recebe as novas operações conhecidas em j, podendo executar as operações necessárias para o bom funcionamento do *coobjecto* (como se exemplifica, por exemplo, no apêndice A.1.8).

7. Após a conclusão bem sucedida da sessão de sincronização, o servidor j executa o protocolo local de mudança de vista executando a operação de remoção respectiva. O servidor i liberta os recursos usados pelo volume v.

Se durante a execução do protocolo existir alguma falha, o protocolo é reiniciado executando novamente o passo em que o protocolo falhou. O passo 6 garante que nenhuma operação é perdida durante a remoção voluntária dum servidor do grupo de replicadores de um volume.

Para que não se perca informação pela saída voluntária de um servidor do grupo de replicadores é necessário que este propague toda a informação que mantém para outro elemento do grupo de replicadores. No caso de a saída ser patrocinada por outro servidor, esta propriedade é garantida pela sessão de sincronização que é estabelecida durante o respectivo protocolo. As saídas não patrocinadas apenas acontecem no caso de não existir mais nenhum servidor que replique o volume.

A.1.5 Operação de *disseminação e execução*

Em alguns dos protocolos que se apresentam de seguida é necessário garantir que apenas se executa uma dada acção após todos os *coobjectos* do volume nesse servidor receberem as operações conhecidas num dado conjunto de servidores num dado momento. Por exemplo, aquando da falha definitiva de um servidor, apenas se podem executar as operações que levam à remoção desse servidor do grupo de replicadores do volume, após se conhecerem todas as operações submetidas no servidor a remover.

Para garantir esta propriedade, cada servidor necessita de manter informação sobre quais os outros servidores dos quais conhece o estado (i.e., os servidores dos quais recebeu o estado após ter sido executada uma dada operação). Esta informação é mantida pelo *coobjecto* de filiação no protocolo de *disseminação e execução* que se descreve de seguida.

Anúncio de início do protocolo Quando se pretende executar a operação *op* após todos os *coobjectos* do volume reflectirem as operações conhecidas actualmente no conjunto de servidores *S*, submete-se a operação $disseminate(uid, S, op, cond)$ no *coobjecto* de filiação, com *uid* um identificador único para este pedido de disseminação e *cond* uma função que verifica se as condições de disseminação do pedido se encontram satisfeitas. A execução desta operação num servidor dá início à execução do protocolo de disseminação e execução.

Informação de disseminação Cada réplica do *coobjecto* de filiação mantém, para cada pedido de disseminação pendente, a informação $(uid, S, op, cond, v, Gc)$, com: *uid*, *op*, *cond* e *S* os valores anteriores; *v* uma matriz de números inteiros tal que $v[i, j] = n$ indica que se sabe que *i* reflecte o estado de *j* após a execução da operação de disseminação e num momento em que, no servidor *j*, já tinha sido executada a operação submetida em *j* com número de sequência *n*; e *Gc* um conjunto usado para remover a informação sobre o pedido de disseminação, inicialmente vazio. Quando a operação de disseminação é executada no servidor *i*, a informação anterior é criada, sendo a matriz *v* iniciada da seguinte forma: $v[i, i] = n$, com *n* o número de sequência da última operação identificada em *i*; e $v[j, l] = -1$ nos outros casos. Adicionalmente, submete-se a operação $disseminated(uid, i, v_i)$, explicada de seguida. Esta informação permite informar todos os replicadores do volume do estado reflectido pela réplica *i*.

Actualização da informação de disseminação Quando, após uma sessão de sincronização, o servidor j fica a conhecer todas as operações conhecidas no servidor i relativas a todos os *coobjectos* do volume, j submete no *coobjecto* de filiação, após executar todas as operações do *coobjecto* de filiação recebidas, para cada pedido de disseminação pendente, a operação $disseminated(uid, j, v_i)$, com v_i o valor do vector $v[i]$ da matiz v associada ao pedido de disseminação que se está a considerar, com excepção de $v_i[j] = n_j$ com n_j o número de sequência da última operação identificada em j.

Esta operação indica que j conhece todas as operações conhecidas em i no momento em que se iniciou o protocolo de *disseminação e execução*, assim como todas as operações conhecidas noutros servidores que entretanto tenham sido propagadas para i.

A execução desta operação, em qualquer servidor, leva à actualização da informação relativa ao servidor j na matriz local v: $\forall n, v[j,n] = max(v[j,n], v_i[n])$.

A correcção do funcionamento desta operação resulta da correcção do valor $v[i]$ aquando da submissão da operação. Esta correcção resulta de se submeter uma operação $disseminated$ no início do protocolo e sempre que se actualiza v, e de executar as operações recebidas num servidor antes de submeter a operação $disseminated$.

Condição de execução da operação (*cond*) Por omissão ($cond = null$), a operação op, relativa a um pedido de disseminação, será executada, no servidor l, quando for possível determinar que l reflecte todas as operações conhecidas no conjunto de servidores S. Esta condição verifica-se quando $\forall n \in S, v[l,n] \neq -1$.

Se um servidor, i, pertencente ao conjunto S, tiver sido removido do grupo de replicadores por uma operação submetida no servidor j (i.e., j foi o patrocinador da remoção de i) e identificada com número de sequência nr_i, deve ter-se $v[l,i] \neq -1 \vee v[l,j] \geq nr_i$ (o protocolo de remoção de um servidor garante que, quando a operação de remoção é executada no patrocinador, o patrocinador reflecte o estado do servidor removido). Se o servidor j que foi patrocinador da remoção de i tiver igualmente sido removido, tendo como patrocinador k, através duma operação submetida em k com número de sequência nr_j, deve ter-se $v[l,i] \neq -1 \vee v[l,j] \geq nr_i \vee v[l,k] \geq nr_j$. De forma semelhante para outras eventuais remoções de servidores.

Reciclagem Quando o servidor i executa a operação op, submete a operação $end_{dissemination}(uid, i)$, indicando que o servidor i não necessita de informação adicional relativa ao pedido de disseminação uid. Quando esta operação é executada num servidor, o conjunto Gc associado ao pedido de disseminação uid é actualizado da seguinte forma $Gc \leftarrow Gc \cup \{i\}$. Quando, num servidor, Gc incluir todos os servidores activos da vista definida pelo *coobjecto* de filiação, a informação relativa ao pedido de disseminação é removida.

Optimizações Em relação a esta descrição existe algumas uma optimização simples: na matriz v apenas é necessário manter as entradas relativas aos servidores contidos em S (e, no caso de um desses servidores ter sido removido, dos servidores que foram os seus patrocinadores, como se explicou anteriormente). Adicionalmente, seria possível usar apenas uma matriz v para todos os pedidos de disseminação pendentes, com as modificações apropriadas no comportamento das operações.

Existe um caso especial para esta operação — a situação em que se pretende a disseminação entre todos os servidores do estado actual de todas as cópias do volume. Neste caso, o servidor que submete a operação do pedido de disseminação pode não ter conhecimento de todos os servidores que actualmente replicam o volume. Assim, é impossível, quando a operação é submetida, determinar exactamente o valor de S. Por isso, usa-se o valor especial $S = \mathcal{U}$. Neste caso, a operação op será executada no servidor i no momento em que, de acordo com a vista actualmente definida localmente, se tiver informação que o estado de todos os servidores activos ou removidos está reflectido na cópia actual do servidor, i.e: (1) para todo o servidor j activo se tem $v[i,j] \neq -1$; (2) para todo o servidor j removido com patrocinador k e número de sequência da operação nr_j se tem $v[i,j] \neq -1 \vee v[i,k] \geq nr_j$ (e de forma semelhante ao apresentado anteriormente no caso de o patrocinador ter sido igualmente removido).

A.1.6 Protocolo de eliminação do identificador de um servidor removido

O *protocolo de eliminação do identificador de um servidor removido* tem por objectivo garantir que o identificador do servidor removido apenas é eliminado quando não é necessário ao bom funcionamento de nenhum *coobjecto*, ou seja, quando as operações do servidor removido tiverem sido propagadas e definitivamente processadas em todos os servidores. A eliminação de um identificador permite que o mesmo seja utilizado por um novo replicador.

O *protocolo de eliminação do identificador de um servidor removido*, i, é iniciado pelo patrocinador da remoção, j, e consiste nos seguintes passos:

- O patrocinador, j, da remoção do servidor i, após submeter a operação de remoção, submete a operação $disseminate(uid, \{j\}, askEliminate(i), null)$. Desta forma, garante-se que a operação $askEliminate$ apenas é executada num servidor após se conhecerem nesse servidor todas as operações conhecidas actualmente em j relativas a todos os *coobjectos* (e, em consequência do protocolo de remoção, todas as operações que foram submetidas em i).

- O *coobjecto* de filiação mantém, para cada identificador i que se pretende eliminar, o par (i,R), em que R contém o conjunto de servidores que deram a sua concordância à eliminação de i.

- A execução da operação $askEliminate(i)$ num servidor l, leva à introdução do par $(i,\{\})$ na informação sobre os identificadores a eliminar.

- Periodicamente, cada servidor l verifica se pode eliminar algum identificador que esteja no conjunto de identificadores a eliminar (e para o qual ainda não tenha dado a sua concordância). Para tal, pergunta-se a todos os *coobjectos* do volume se eles concordam com a eliminação do identificador, usando uma função definida na *interface do coobjecto*. Em geral, um *coobjecto* apenas se opõe à remoção de um identificador quando as operações submetidas no servidor que tinha esse identificador ainda não tiverem sido executadas de forma estável na cópia local do *coobjecto*. Se nenhum *coobjecto* se opuser à remoção do identificador, o servidor l submete a operação $canEliminate(i,l)$ no *coobjecto* de filiação — a execução desta operação em qualquer servidor leva à introdução de l no conjunto R de servidores que deram a sua concordância à eliminação de i.

- Quando qualquer servidor l detecta que todos os servidores activos na vista corrente deram o seu acordo à eliminação de um identificador i (usando o conjunto R), ele submete a operação $eliminate(i)$. A execução desta operação, em qualquer servidor, leva à remoção da informação associada ao servidor identificado por i e à libertação do respectivo identificador local (assim como à remoção da informação associada a i no conjunto de identificadores a remover). A submissão concorrente desta operação por mais do que um servidor não põe problemas para a correcção do funcionamento do sistema (a ordenação total das operações leva a que as réplicas convirjam para o mesmo estado e que apenas a primeira operação de eliminação tenha efeitos no estado do *coobjecto* de filiação).

A.1.7 Protocolo de saída forçada do grupo de replicadores de um volume

O *protocolo de saída forçada do grupo de replicadores de um volume* controla a remoção forçada de um servidor do grupo de replicadores do volume. Esta operação é necessária quando, por exemplo, um servidor ficou danificado e a memória estável na qual estava armazenado o estado dos *coobjectos* do volume é irrecuperável.

Estratégia O funcionamento dos *coobjectos* obriga a que se informem todos os *coobjectos* do volume sobre a remoção do servidor. No entanto, os *coobjectos* apenas devem ser notificados após se conhecerem todas as operações submetidas no servidor a remover[4] (e identificadas com um número de sequência e o identificador desse servidor) — na remoção voluntária, o servidor que vai ser removido encontra-se nesta

[4]Por exemplo, quando um *coobjecto* ordena totalmente as operações usando técnicas baseadas na verificação da estabilidade da ordem é importante saber qual o valor do número de sequência da última operação identificada por cada servidor removido, por forma a permitir a correcta ordenação de todas as operações dos servidores removidos e a expedita ordenação das operações originadas noutros servidores.

situação e pode informar os *coobjectos*. Para que estas condições se verifiquem foi adoptada a seguinte estratégia para a remoção forçada de um servidor:

1. Um qualquer servidor pode efectuar o pedido de remoção forçada de um outro servidor j. No caso de mais do que um servidor pedir concorrentemente a remoção do mesmo servidor j, apenas o pedido que é executado primeiro (segundo uma ordem total definida entre estes pedidos) é considerado.

2. Cada servidor indica a sua concordância (ou discordância) na remoção do servidor (no caso da remoção ser motivada pela inacessibilidade ao servidor, cada servidor pode tomar a sua decisão em função de conseguir aceder ao servidor ou não). A partir do momento em que um servidor i concorda com a remoção forçada do servidor j, i não pode executar mais nenhuma comunicação (sessão de sincronização) com j. Se qualquer servidor mostrar a sua discordância na remoção do servidor j, o pedido de remoção forçada é abortado (e todos os servidores podem voltar a comunicar com j).

3. Concorrentemente, força-se a disseminação entre todos os servidores do estado actual de todos os *coobjectos* (por forma a que sejam conhecidas todas as operações identificadas pelo servidor a remover).

4. Em cada momento, define-se um número de ordem para cada servidor que replica um dado volume (definido pela ordem de entrada no volume segundo a ordem total definida para as operações de filiação e considerando apenas os servidores que actualmente replicam o volume).

5. A remoção forçada de um servidor j pode prosseguir se o servidor com menor número de ordem verificar que: (1) todos os servidores do volume, com excepção do servidor j, concordaram com a remoção (ou seja, mais nenhuma servidor aceitará comunicações de j); e (2) o estado actual do servidor reflecte os estados de todos os servidores no momento em que eles executaram a operação de concordância com a remoção (ou seja, o estado actual reflecte todas as operações submetidas por j conhecidas nos servidores que continuarão a replicar o volume — dado que mais nenhuma comunicação com origem em j é aceite pelos servidores após declararem a sua concordância com a remoção). Neste caso, no servidor com menor número de ordem, informam-se todos os *coobjectos* da remoção de j (de forma a que os *coobjectos* tomem as medidas necessárias à remoção segura do servidor), após o que se submete a operação de remoção de j no *coobjecto* de filiação (durante a execução destas acções, o servidor não efectua comunicações com outros servidores ou clientes).

Podem existir situações em que mais do que um servidor necessite de ser removido simultaneamente ou que o servidor com menor número de ordem seja um dos servidores a remover. Para considerar

estas situações, a última condição expressa anteriormente é generalizada para (considerando a remoção conjunta de um conjunto R de servidores, sendo que o conjunto S de servidores não removidos — com $U = R \cup S$, o conjunto de servidores que replica o volume — define um quorum (neste caso, uma maioria) no sistema U):

- A remoção forçada de um conjunto R de servidores pode prosseguir se o servidor com menor número de ordem, não contido em R, verificar que: (1) todos os servidores do volume, com excepção dos servidores incluídos em R, concordaram com a remoção de todos os servidores em R (convém realçar que é possível que alguns servidores em R tenham concordado com a remoção de outros servidores presentes em R e tenham posteriormente sido alvo de remoção forçada); (2) o estado actual do servidor reflecte os estados de todos os servidores no momento em que eles executaram a última operação de concordância com a remoção de um servidor pertencente a R (ou seja, o estado actual reflecte todas as operações submetidas pelos servidores em S).

Implementação Para implementar a estratégia descrita anteriormente, definiram-se as seguintes operações no *coobjecto* de filiação:

askRemove(uid,i) Pede a remoção do servidor i do grupo de replicadores, usando o identificador único *uid* para identificar univocamente o pedido. Esta operação, quando executada, coloca o estado do servidor i como $a_remover(uid)$ (no caso de o estado já ser $a_remover(uid_0)$, a operação falha e é submetida a operação *rollbackRemove(uid)*)[5].

agreeRemove(j,uid) Indica a concordância do servidor j com o pedido de remoção (do servidor i) com identificador uid. O *coobjecto* de filiação mantém, para cada pedido de remoção de um servidor, a lista de servidores que afirmaram a sua concordância com essa remoção. Após a submissão desta operação, o servidor j não pode retroceder na sua intenção de remover i, nem estabelecer qualquer sessão de sincronização com i, a menos que seja executada uma operação de $rollbackRemove$ noutro servidor. Um servidor l que inicie a replicação de um volume deve executar a acção *agreeRemove(l,uid)* se o seu patrocinador, o servidor j, tiver executado a operação *agreeRemove(j,uid)*

[5] A execução optimista das operações no *coobjecto* de filiação pode levar a que a primeira execução da operação *askRemove(uid,i)* coloque o estado do servidor i como $a_remover(uid)$, enquanto na execução definitiva se verifique que já existe outro pedido anterior para a remoção do mesmo servidor i — esta situação não afecta a correcção do funcionamento do sistema, devendo em cada execução serem executadas as acções indicadas. Quando a execução optimista da operação é desfeita, o estado associado ao servidor i é reposto no estado anterior, mas as operações submetidas em consequências da primeira execução não são removidas (uma optimização possível é submeter a operação de *rollbackRemove* apenas uma vez). A correcção de funcionamento do sistema também não é afectada se aquando da execução definitiva, ao contrário do que aconteceu numa execução optimista, o estado do servidor i não for $activo$ — as operações de $rollbackRemove(uid)$ encarregam-se de abortar este pedido. Neste caso, um novo pedido deve ser efectuado posteriormente.

antes de executar a operação de inserção de l no grupo de replicadores do volume. Caso contrário, o servidor l pode tomar uma posição independente.

rollbackRemove(uid) Indica que o pedido de remoção (do servidor i) com identificador *uid* deve ser cancelado. A execução desta operação recoloca o estado de i em activo (e permite que os servidores que tinham dado a sua concordância com a remoção de i possam voltar a comunicar com i).

readyRemove(uid) Indica que a cópia local do servidor está pronta para concluir o pedido de remoção forçada. A remoção apenas se executará se os outros servidores tiverem concordado com a remoção (como se explicou anteriormente).

Usando estas operações e a operação de disseminação definida anteriormente, a *remoção forçada de um servidor i do grupo de replicadores do volume v* processa-se da seguinte forma:

1. Um qualquer servidor j, pertencente ao grupo de replicadores do volume v, submete as operações $askRemove(uid, i)$ e $disseminate(uid_d, \mathcal{U}, readyRemove(uid), readyCond)$ no *coobjecto* de filiação do volume v. A operação *askRemove* dá início ao processo de remoção forçada de um servidor — a sua execução com sucesso coloca o estado do servidor em $a_remover(uid)$. A operação de disseminação garante a propagação das operações identificadas no servidor i para todos os replicadores do volume.
   A condição *readyCond* verifica se as condições para a execução da operação *readyRemove* (descritas anteriormente) são satisfeitas, i.e., se todos os servidores que não vão ser removidos concordam com a remoção e se, para todos os *coobjectos*, todas as operações submetidas em i, e conhecidas em qualquer servidor aquando da execução da operação *agreeRemove*, estão reflectidas na réplica local do *coobjecto*[6].
   Assim, *readyCond* será verdadeiro no servidor i, quando: (1) todos os servidores que não vão ser removidos submeteram a operação *agreeRemove* respectiva; (2) para todo o servidor activo, j, que submeteu a operação $agreeRemove(j, uid)$ identificada em j com o número de sequência t_{cr}, se tem $v[i, j] \geq t_{cr}$; (3) para todo o servidor removido j, removido pela operação de remoção submetida no patrocinador k com número de sequência t_{rj}, se tem $v[i,k] \geq t_{rj}$[7]. Se o servidor k tiver posteriormente sido removido pela operação de remoção submetida no patrocinador l com número de sequência t_{rk}, deve ter-se $v[i,k] \geq t_{tj} \vee v[i,l] \geq t_{rk}$ (e de modo semelhante para consecutivas remoções).

[6] O facto de a operação *agreeRemove* ser executada independentemente da operação de disseminação leva a que a condição definida por omissão não seja válida neste caso.

[7] Se o servidor removido, j, tiver submetido a operação $agreeRemove(j, uid)$, basta verificar a primeira condição.

2. Um servidor j, no qual o estado do servidor i é $a_remover(uid)$, após executar as verificações que considere necessárias, deve submeter a operação $agreeRemove(j, uid)$ ou $rollback(uid)$ se, respectivamente, concordar ou discordar da remoção do servidor (e, caso concorde, actuar em conformidade com essa decisão).

3. A execução da operação *readyRemove* no servidor i, verifica se i é o responsável por efectuar a remoção, considerando todos os pedidos de remoção pendentes (i.e., se é o servidor com menor número de ordem na vista actual dos servidores que não se incluem no conjunto de servidores prontos a serem removidos). O responsável pela remoção, actua como patrocinador da remoção e executa as seguintes operações (sem permitir acessos concorrentes de outros servidores ou clientes): (1) informa todos os *coobjectos* do volume quais são os servidores que vão ser removidos; (2) submete as operações de remoção respectivas (através do protocolo local de mudança de vista).

O mecanismo de remoção forçada de um servidor leva a que operações propagadas por um cliente para o servidor removido sejam perdidas caso não tenham sido anteriormente propagadas para outro servidor que permaneça no sistema. Assim, este mecanismo apenas deve ser usado em relação aos servidores que se encontram permanentemente desconectados do sistema (por exemplo, devido a terem sido danificados). Se um servidor removido não estiver permanentemente desconectado, ele deve propagar as operações que recebeu de clientes para um servidor que esteja a replicar o volume e, caso pretenda, pedir a sua inserção no grupo de replicadores.

A.1.8 Protocolo de execução dos blocos *só-uma-vez* em ordenações por verificação da estabilidade

Nos protocolos que ordenam as operações verificando a estabilidade da ordem (ordem causal – secção 4.1.3, ordem total – secção 4.1.4.2), quando um servidor sai voluntariamente do grupo de replicadores de um volume, o patrocinador assume a responsabilidade de executar os blocos *só-uma-vez* das operações que ainda não tenham sido executadas no servidor removido. No entanto, como o patrocinador pode já ter executado algumas das operações das quais devia executar os blocos *só-uma-vez*, o seguinte protocolo é usado para garantir que os blocos *só-uma-vez* são executados uma e uma só vez:

1. Quando a réplica de um *coobjecto* num servidor é informada que esse servidor vai ser removido do grupo de replicadores (passo 4 do protocolo de remoção), executa uma operação que indica quais as operações executadas até esse momento e quais as operações pelas quais é responsável por executar os blocos *só-uma-vez* (neste caso, as operações recebidas no servidor e aquelas cuja responsabilidade possa ter sido delegada anteriormente). Esta operação delega no patrocinador a responsabilidade de executar os blocos *só-uma-vez* das operações ainda não executadas.

2. Quando a réplica de um *coobjecto* num servidor é informada que esse servidor vai servir de patrocinador à remoção de outro servidor (passo 5 do protocolo de remoção), executa uma operação que indica quais as operações executadas até esse momento. Até à conclusão da sessão sincronização executada no âmbito do protocolo de remoção, mais nenhuma operação é executada nessa réplica do *coobjecto*. Esta operação delega no servidor a remover a responsabilidade de executar os blocos *só-uma-vez* das operações que já foram executadas no patrocinador.

3. Após a sessão de sincronização do protocolo de remoção (passo 6), a réplica do *coobjecto* no servidor a remover executa todas as operações que já tenham sido executadas no patrocinador e pelas quais é responsável por executar os blocos *só-uma-vez* (para tal, pode ter de executar outras operações). A sessão de sincronização executada garante que no servidor a remover é possível executar essas operações.

4. Após a sessão de sincronização do protocolo de remoção, a réplica do *coobjecto* presente no patrocinador retoma a sua execução normal, i.e., executa todas as operações possíveis. No entanto, fica responsável por executar os blocos *só-uma-vez* de todas as operações pelas quais o servidor removido era responsável e que ainda não tinham sido executadas quer no servidor removido quer no patrocinador no início do protocolo (i.e., as operações cujo número de sequência seja superior ao máximo dos números de sequência indicados como executados pelas operações submetidas nos passos 4 e 5 deste protocolo).

Quando se processa a remoção forçada de um servidor, os blocos *só-uma-vez* das operações recebidas nesse servidor não são executadas em nenhum outro servidor. Assim, neste caso, não se garante que todos os blocos *só-uma-vez* sejam executados.

A.2 Sincronização epidémica

Os protocolos de sincronização epidémica são estabelecidos entre pares de servidores para sincronizar o estado de um conjunto de *coobjectos* pertencentes a um volume. Durante estas sessões epidémicas os servidores propagam entre si as operações que conhecem (independentemente do servidor no qual foram recebidas). Desta forma, cada servidor recebe, para cada *coobjecto*, todas as operações submetidas em todos os servidores, directa ou indirectamente.

No sistema DOORS foram implementados dois algoritmos de sincronização. Primeiro, um algoritmo de sincronização bilateral, em que dois servidores trocam entre si as novas operações que conhecem. Segundo, um algoritmo de sincronização unilateral que permite a um servidor enviar para outro servidor as operações que conhece. Estes algoritmos são apresentados brevemente de seguida — uma discussão

mais detalhada foi apresentada anteriormente em [125]. Antes de apresentar os algoritmos, descreve-se a informação utilizada para controlar o seu funcionamento.

A.2.1 Informação utilizada

Em cada cópia do *coobjecto* é mantido um sumário das operações conhecidas. Este sumário é representado como um vector-versão l, em que, $l[i] = n_i$ significa que a cópia do *coobjecto* conhece todas as operações identificadas no servidor i com um número de sequência igual ou inferior a n_i (a cópia do *coobjecto* pode, ainda, conhecer operações com número de sequência superior a n_i). Estes sumários mantêm informação relativa, não só, aos servidores activos, mas também, aos servidores removidos ainda não eliminados. Em cada servidor i, $l[i]$ mantém o valor actual do relógio lógico usado para identificar as novas operações nesse servidor — a nova operação será identificada com o número de sequência $l[i]+1$ e o valor de $l[i]$ será actualizado para reflectir a nova operação ($l[i] \leftarrow l[i]+1$).

Cada cópia dum *coobjecto* mantém ainda informação sobre as operações conhecidas nos outros servidores. Como se referiu na secção 3.3.2, foram definidas duas estratégias que podem ser usadas pelos *coobjectos*. Na primeira estratégia, cada réplica mantém uma matriz m, em que $m[i,j] = m^i_j$ significa que se sabe que o servidor i conhece todas as operações submetidas em j com número de sequência menor ou igual a m^i_j. Na segunda estratégia, mantém-se um vector m, em que $m[i] = m_i$ significa que se sabe que o servidor i conhece todas as operações submetidas em todos os servidores com número de sequência igual ou inferior a m_i. Esta segunda aproximação, proposta em [57], permite reduzir o espaço necessário — em [125] discute-se a utilização desta ideia e a importância da actualização local do valor de $l[i]$ para que o valor de a possa evoluir rapidamente.

A informação sobre as operações conhecidas nos outros servidores é usada para remover as operações não necessárias. Assim, uma operação que seja conhecida em todos os outros servidores e já tenha sido executada definitivamente pode ser removida.

A.2.2 Protocolo de propagação epidémica bilateral

O *protocolo de propagação epidémica bilateral* é construído com base nas seguintes quatro operações executadas relativamente a *coobjectos* do volume, com *id* o identificador do *coobjecto* a sincronizar e a a informação sobre as operações que se sabe serem conhecidas noutros servidores (usando qualquer uma das estratégias descritas anteriormente):

$(asksync, id, l, a)$ Pede o envio das operações não conhecidas localmente, i.e., não reflectidas no sumário l.

Ao receber esta operação, o servidor pode actualizar a informação sobre as operações conhecidas nos outros servidores.

Em resposta a esta operação deve ser emitida a operação $(sync, \ldots)$, $(askstate, \ldots)$ ou $(removed, \ldots)$.

$(sync, id, l, a, l_0, ops)$ Envia o conjunto de operações conhecidas localmente, *ops*, com sumário *l*, e que não estão reflectidas no sumário l_0.

Ao receber esta operação, o servidor entrega as operações recebidas ao *coobjecto* e actualiza o sumário das operações conhecidas (com base no sumário do parceiro, *l*, e no sumário, l_0, usado para gerar *ops*). O servidor actualiza também a informação sobre as operações conhecidas noutros servidores.

Em resposta a esta operação deve ser emitida a operação $(sync, \ldots)$, $(askstate, \ldots)$ ou $(removed, \ldots)$.

$(askstate, id)$ Pede o envio do estado de um *coobjecto* (usado para propagar novos *coobjectos*).

Em resposta a esta operação deve ser emitida a operação $(newstate, \ldots)$.

$(removed, id)$ Indica que o servidor obteve informação suficiente para remover definitivamente o *coobjecto id*.

O servidor que recebe esta operação remove igualmente os recursos relativos ao *coobjecto*.

$(newstate, id, o)$ Envia o estado de um *coobjecto*.

Ao receber esta operação, o servidor cria a cópia local do *coobjecto*, caso ela ainda não exista.

O estado de um *coobjecto* transmitido entre servidores inclui os atributos do sistema e uma sequência opaca de bytes obtida através dum método definido na interface do *coobjecto*. Os atributos do sistema especificam a fábrica do *coobjecto* — esta fábrica é usada para criar uma cópia do *coobjecto* a partir da sequência de bytes recebida.

Nas figuras A.1 e A.2 apresentam-se, de forma simplificada, as funções usadas no protocolo de sincronização. O protocolo consiste numa sequência de passos executados assincronamente — em cada um dos passos um servidor processa as mensagens recebidas (usando a função *receive*) e envia mensagens para o seu parceiro (usando a função *send*). A sequência de mensagens enviadas pode ser propagada conjuntamente usando qualquer transporte, síncrono ou assíncrono.

O protocolo é iniciado executando a função *initBiEpidemic*, em que o servidor *i* pede que o servidor *j* lhe envie, para todos os *coobjectos* a sincronizar, as operações desconhecidas em *i* (usando a operação $(asksync, \ldots)$). Os passos seguintes do protocolo, executados alternadamente pelos servidores *j* e *i*, consistem na execução da função *stepBiEpidemic*. Nesta função, um servidor processa as operações submetidas pelo parceiro, como se explicou anteriormente (i.e., enviando e/ou recebendo operações ou pedindo/recebendo o estado de um novo *coobjecto*). Adicionalmente, o servidor pode submeter operações relativas a *coobjectos* não referenciados pelo parceiro. No início de cada passo do protocolo, o

```
Variáveis globais para um dado volume vol
U ← todos os coobjectos
Del ← identificadores dos coobjectos removidos
view_id ← identificador da vista actualmente instalada no servidor
srv ← identificador do servidor na vista instalada
memb ← coobjecto de filiação

Variáveis definidas para um coobjecto o
o.id ← identificador do coobjecto no volume Id_local
o.l ← sumário das operações conhecidas em o
o.a ← informação sobre operações conhecidas noutro servidores
o.ops ← conjunto de operações conhecidas em o

Variáveis usadas para identificar uma operação op
op.srv ← identificador do servidor
op.nseq ← número de sequência da operação

Variáveis usadas durante o protocolo
lastStep ← define se se está no último passo do protocolo

Funções auxiliares

function exists( id): boolean
(1)    return {o ∈ U : o.id = id} ≠ ∅

function removed( id): boolean
(1)    return {id0 ∈ Del : id0 = id} ≠ ∅

function getCoobject( id): coobject
(1)    return o ∈ U : o.id = id

function unknowOps( o, l): set
(1)    return {op ∈ o.ops : op.nseq > l[op.srv]}

function unknowMembOps( l): set
(1)    return {op ∈ memb.ops : op.nseq > l[op.srv] ∨
                 op.srv não representado em l}

procedure send0( msg)
(1)    if NOT lastStep then send msg end if

procedure updateLogged(o,l_0,l)
(1)    ∀ n, o.l[n] ≥ l_0[n] ⇒ o.l[n] := max(o.l[n],l[n])

procedure updateClock( o)
(1)    o.l[srv] := max{o.l[n] : ∀ n}

procedure updateAck( o, a)        // Golding
(1)    o.a[srv] := min{o.l[n] : ∀ n}
(2)    o.a[n] := max(o.a[n],a[n]), ∀ n, n ≠ srv

procedure updateAck( o, a)        // Matriz
(1)    o.a[srv][n] := o.l[n] : ∀ n
(2)    o.a[m][n] := max(o.a[m][n],a[m][n]), ∀ n,m, m ≠ srv
```

Figura A.1: Funções básicas e informação usada no protocolo de sincronização bilateral (simplificado).

Funções do protocolo

```
procedure syncViewSend(view_peer)
 (1)   send (sync view,view_id,
             unknownMembOps(view_peer))

procedure syncViewReceive()
 (1)   receive (sync view,view_peer,ops)
 (2)   memb.ops := memb_ops ∪ ops
      (no protocolo local de mudança de vista)
 (3)   if view_id ≠ view_peer then
 (4)     send (sync view,view_id,
               unknownMembOps(view_peer))
 (5)   end if

procedure initBiEpidemic( Objs)
 (1)   send (view_id)
 (2)   nStep := 1; full := Objs = U
 (3)   send (nStep,full)
 (4)   for ∀ o ∈ Objs do
 (5)     updateClock(o)
 (6)     send (ask sync,o.id,o.l,o.a)
 (7)   end for
 (8)   send (end)

procedure stepBiEpidemic()
 (1)   receive (view_peer)
 (2)   if view_id ≠ view_peer then
 (3)     syncViewSend(view_peer); return;
 (4)   end if
 (5)   receive (nStep,full)
 (6)   lastStep := lastStep(nStep)
 (7)   send0 (view_id)
 (8)   send0 (nStep+1,full)
 (9)   Objs = ∅
 (10)  forever do
 (11)    receive msg
 (12)    if msg.type = end then break end if
 (13)    processOne( msg)
 (14)    Objs := Objs ∪ {msg.id}
 (15)  end forever
 (16)  if full AND NOT lastStep then
 (17)    for ∀ o ∈ U : o.id ∉ Objs do
 (18)      send (ask sync,o.id,o.l,o.a)
 (19)    end for
 (20)  end if
 (21)  send0 (end)

procedure processOne( msg)
 (1)   case msg of
 (2)     (ask sync,id,l,a):
 (3)       if exists(id) then
 (4)         o = getCoobject(id)
 (5)         updateClock( o); updateAck( o, a)
 (6)         send0 (sync,id,o.l,o.a,unknownOps(o,l))
 (7)       else if removed(id) then
 (8)         send0 (removed,id)
 (9)       else
 (10)        send0 (ask state,id)
 (11)      end if
 (12)    (sync,id,l,a,l_0,ops):
 (13)      if exists(id) then
 (14)        o = getCoobject(id)
 (15)        o.ops := o.ops ∪ ops
 (16)        updateLogged(o,l_0,l);updateClock(o);
                                updateAck(o,a)
 (17)        send0 (sync,id,o.l,o.a,l,unknownOps(o,l))
 (18)      else
 (19)        send0 (ask state,id)
 (20)      end if
 (21)    (removed,id):
 (22)      U := U \ {getCoobject(id)}
 (23)      Del := Del ∪ {id}
 (24)    (ask state,id):
 (25)      if exists(id) then
 (26)        o = getCoobject(id)
 (27)        send0 (new state,id,o)
 (28)      else
 (29)        // do nothing
 (30)      end if
 (31)    (new state,id,o):
 (32)      if exists(id) then
 (33)        // error: do nothing
 (34)      else
 (35)        U := U ∪ {o}
 (36)      end if
```

Figura A.2: Funções do protocolo de sincronização bilateral (simplificado).

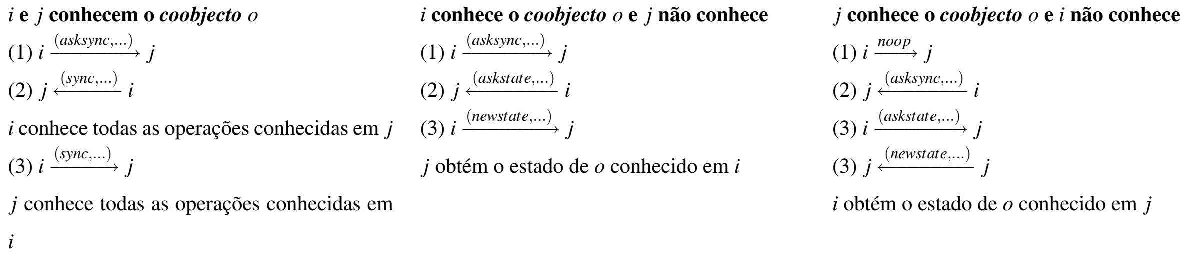

Figura A.3: Descrição da sincronização de um *coobjecto* usando o protocolo de sincronização bilateral.

servidor que recebe as mensagens verifica se se encontra na mesma vista do servidor que enviou as mensagens (linhas 2-4 da função $stepBiEpidemic$). Caso se encontrem em vistas diferentes, as mensagens recebidas são ignoradas e os servidores iniciam um protocolo de sincronização das vistas, enviando para o outro servidor todas as operações relativas ao *coobjecto* de filiação desconhecidas pelo outro servidor.

Na figura A.3 apresentam-se as diferentes possibilidades de sincronização de um *coobjecto*, dependendo de o *coobjecto* ser já conhecido nos servidores quando o protocolo é iniciado. Assim, verifica-se que o protocolo de sincronização se pode concluir ao fim da quarta interacção, i.e., quando o servidor que iniciou o protocolo recebe uma resposta pela segunda vez — nesse momento, cada um dos servidores recebeu as operações conhecidas pelo outro servidor no momento em que iniciou a participação no protocolo. No entanto, como o protocolo pode ser estabelecido usando meios de comunicação assíncronos, a latência da comunicação pode aconselhar a que o protocolo continue por mais alguns passos. No código apresentado na figura A.2 esta decisão deve ser definida na função $lastStep$. No último passo do protocolo, o servidor limita-se a processar as operações recebidas do parceiro sem lhes responder.

O protocolo implementado no sistema DOORS apresenta algumas diferenças de pormenor em relação ao código das figuras A.1 e A.2. Primeiro, ao conjunto de *coobjectos* a sincronizar, $Objs$, é adicionado o conjunto de *coobjectos* necessários para os instanciar (i.e., seja o_i^{code} o *coobjecto* que contém o código para instanciar o *coobjecto* o_i, tem-se $Objs \leftarrow Objs \cup \{o_i^{code} : o_i \in Objs\}$). Adicionalmente, os *coobjectos* que contêm código para instanciar outros *coobjectos* são processados antes desses *coobjectos* (i.e., o_i^{code} é processado antes de o_i) — desta forma garante-se que um servidor contém sempre o código necessário a processar as operações relativas aos *coobjectos*.

A segunda diferença consiste numa optimização para omitir o envio de informação que se sabe que o parceiro conhece. Assim, um servidor omite sempre o envio de operações $(sync, \ldots)$ desde que as seguintes condições se verifiquem: (1) não existem operações a enviar; (2) o sumário das operações conhecidas pelos dois parceiros é igual[8] — o sumário das operações conhecidas no parceiro é o recebido durante o

[8]Esta condição não é idêntica à anterior porque é possível sumários diferentes reflectirem o mesmo conjunto de operações — por exemplo, se $l[i] = n$ e $l[i] = n+1$ reflectem o mesmo conjunto de operações se não existir (no presente nem no futuro) nenhuma operação submetida em i com número de sequência $n+1$.

processo de sincronização numa operação $(asksync, \ldots)$ ou $(sync, \ldots)$ ou é obtido a partir da informação sobre as operações conhecidas noutros servidores; (3) a informação sobre as operações conhecidas noutros servidores é igual — quando a informação do parceiro não foi obtida no decurso do processo de sincronização, considera-se que a informação nos dois servidores é igual, se pela informação conhecida localmente se souber que todos os servidores conhecem todas as operações conhecidas localmente. Um servidor a efectuar uma sincronização completa omite igualmente as operações $(asksync, \ldots)$, se as duas últimas condições enunciadas anteriormente se verificarem (caso o parceiro conheça novas operações, ele enviará a operação $(asksync, \ldots)$ relativa a esse *coobjecto*). Esta optimização permite que durante as sessões de sincronização não se troque informação relativa aos *coobjectos* estáveis, i.e., aos *coobjectos* cujas últimas modificações já se encontram disseminadas.

A terceira diferença consiste na omissão dos pormenores relativos ao mecanismo de disseminação apresentado anteriormente. Assim, quando se conclui um processo de disseminação completo (i.e, $full$ é verdadeiro), o servidor j, submete para cada pedido de disseminação pendente a operação $disseminated(uid, j, v_i)$, como se tinha referido. Os valores de v_i a serem usados são enviados por i a j no início do protocolo de sincronização e mantidos por i até ao fim do protocolo.

Finalmente, omitiram-se os detalhes relativos ao restauro de *coobjectos*.

A.2.3 Protocolo de propagação epidémica unilateral

No *protocolo de propagação epidémica unilateral*, um servidor, i, envia para outro servidor, j, a informação que permite a j sincronizar o estado de j com o estado de i (i.e., o servidor j deve ficar a conhecer todas as operações conhecidas em i). Na figura A.4 apresentam-se as funções usadas no protocolo. O servidor i executa a função $uniEpidemic$ para enviar as operações que pensa serem desconhecidas em j. O servidor j executa a função $stepBiEpidemic$ usada no protocolo bilateral, mas não responde ao servidor i. Um aspecto merece atenção neste protocolo: a função $unknownObj$ é usada para estimar se o servidor a que se destina a comunicação conhece o *coobjecto* que se está a considerar (caso não conheça é necessário enviar o estado do *coobjecto* em vez das operações para sincronizar o estado). A função apresentada na figura A.4 é muito simples e poderia ser melhorada com o conhecimento sobre o envio do estado dos novos *coobjectos* em anteriores sessões de sincronização unilateral.

A.2.4 Sincronização epidémica durante as mudanças na filiação

O mecanismo de entrada e saída do grupo de replicadores exige que o servidor que entra ou sai do grupo de replicadores comunique apenas com outro servidor. Enquanto as operações de inserção/remoção de elementos no grupo de replicadores não são propagadas para todos os replicadores do volume (durante as sessões de propagação epidémica), os servidores vão ter uma visão incompleta ou incorrecta dos actuais

Funções auxiliares

```
// Devolve um vector das operações que se sabe serem conhecidas no servidor to_srv
function knownOps( o, to_srv): vector      // Golding
 (1)    return l : l[i] := o.a[to_srv], ∀ i

function knownOps( o, to_srv): vector      // Matriz
 (1)    return l : l[i] := o.a[to_srv][i], ∀ i

// Devolve verdadeiro se se pensa que o servidor to_srv desconhece o coobjecto o
function unknownObj( o, to_srv): boolean
 (1)    return o.l[to_srv] = −1
```

Funções do protocolo

```
procedure uniEpidemic( Objs,to_srv)
 (1)    send (view_id)
 (2)    nStep := 1; full := Objs = U
 (3)    send (nStep,full)
 (4)    for ∀ o ∈ Objs do
 (5)      updateClock(o)
 (6)      if unknownObj( o, to_srv) then
 (7)        send (new state,o.id,o)
 (8)      else
 (9)        l := knownOps( o, to_srv)
 (10)       send (sync,o.id,o.l,o.a,l,unknownOps(o,l))
 (11)     end if
 (12)   end for
 (13)   send (end)
```

Figura A.4: Funções do protocolo de sincronização unilateral (simplificado).

replicadores do volume. Assim, a política de sincronização deve ter em atenção este facto de forma a que as mudanças no grupo de replicadores sejam propagadas para todos os servidores.

Quando, devido à informação local desactualizada, um servidor selecciona executar uma sessão de sincronização com um outro servidor que deixou de replicar o volume, a sessão de sincronização é recusada. No entanto, os servidores que abandonam o grupo de replicadores mantém temporariamente a informação sobre o conjunto de servidores que replicavam o volume no momento da sua remoção — esta informação é devolvida aos servidores que tentam estabelecer sessões de sincronização e poderá ser usada pela política de sincronização para determinar o próximo servidor a ser contactado.

Em situações extremas, quando existem várias entradas e saídas no grupo de replicadores num curto intervalo de tempo sem que as respectivas operações possam ser propagadas entre os servidores, é possível que existam dois servidores, i e j, que replicam um mesmo volume e que não conhecem nenhum servidor activo que lhe permita conhecer o outro, i.e., (usando $a \rightarrow b$ para representar que a sabe que b pertence ao grupo de replicadores e $activo(a)$ para representar que o servidor a não abandonou o grupo de replicadores) $\neg\exists s_0,\ldots,s_n : i \rightarrow s_0 \wedge activo(s_0) \wedge s_0 \rightarrow s_1 \wedge activo(s_1) \wedge \ldots \wedge s_n \rightarrow j$. Nesta situação, a execução de sessões de sincronização apenas com o conjunto dos servidores que pertencem à vista actual poderia levar à criação de subconjuntos isolados de replicadores que não comunicam entre si.

Para detectar esta situação, para cada volume, mantém-se a informação sobre o momento em que se realizaram (directa ou transitivamente) as últimas sessões de sincronização com os outros servidores — esta informação é transmitida e actualizada durante as sessões de sincronização. O facto de um servidor activo na vista instalada localmente não participar nas sessões de sincronização durante um período de tempo longo, considerando a frequência esperada das sessões de sincronização, pode indiciar que o servidor foi removido usando como patrocinador um servidor que é desconhecido (outra hipótese é tratar-se de um servidor desconectado temporariamente do sistema).

Um servidor, i, que detecte a situação anterior deve tentar estabelecer uma sessão de sincronização com o servidor, j, que não tem participado nas sessões de sincronização. Se o servidor j responder, duas hipóteses são possíveis. Primeiro, j ainda pertence ao grupo de replicadores — neste caso a sessão de sincronização decorre normalmente. Segundo, j já não pertence ao grupo de replicadores e devolve a lista dos servidores que pertenciam ao grupo de replicadores no momento da sua remoção. No caso de existir algum elemento que não faça parte da vista instalada localmente, i deve tentar estabelecer uma sessão de sincronização com esse servidor. Caso todos os elementos da lista devolvida por j pertençam ao conjunto de replicadores conhecidos em i, i deve aguardar que a informação da remoção de j do grupo de replicadores seja propagada até si (ou pode, alternativamente, estabelecer uma sessão de sincronização com o patrocinador da remoção de j). Se o servidor j não responder, o servidor i pode usar o mecanismo de descoberta dos servidores que replicam um dado volume (descrito na secção 6.2.3) para tentar encontrar novos servidores que pertençam ao grupo de replicadores.

A.2.5 Disseminação de novas operações

Além do mecanismo básico de sincronização usando sessões epidémicas, cada servidor propaga os eventos submetidos localmente usando um canal de disseminação não fiável (descrito na secção 6.2.3). O evento propagado tem a seguinte forma $newop(view_{id}, id, op, nseq_{old})$, com $view_{id}$ o identificador da vista no servidor que submete o evento, id o identificador do *coobjecto* no qual a operação foi submetida, op a sequência de operações que o servidor está a disseminar ($op.srv$ e $op.nseq$ identificam a operação), e $nseq_{old}$ o número de sequência da operação que tinha sido identificada anteriormente em $op.srv$.

Um servidor, ao receber um evento $newop(view_{id}, id, op, nseq_{old})$, verifica se o identificador da vista é igual ao identificador da vista instalada localmente e se o *coobjecto* referido é conhecido localmente. Em caso afirmativo, a operação op é entregue ao *coobjecto* com identificador id e o sumário das operações conhecidas localmente é actualizado se se verificar que não existe nenhuma outra operação identificada nesse servidor que não seja conhecida, i.e., seja o o *coobjecto* com identificador id, tem-se que $o.l[op.srv] \geq nseq_{old} \Rightarrow o.l[op.srv] \leftarrow op.nseq$.

Apêndice B

Mobisnap

B.1 Transacções móveis:linguagem

As transacções móveis são especificadas num subconjunto da linguagem PL/SQL [112]. As (poucas) modificações introduzidas estão ligadas com as especificidades das transacções móveis e com limitações do protótipo do sistema Mobisnap. De seguida, apresentam-se brevemente os elementos que podem ser usados na definição de uma transacção móvel:

Tipos Estão disponíveis os seguintes tipos escalares pré-definidos no PL/SQL: carácter e cadeia de caracteres, número inteiro e real com precisão variável, data, booleano.

Constantes e variáveis É possível definir constantes e variáveis dos tipos pré-definidos.

Instrução de atribuição Atribui o valor de uma expressão a uma variável.

Bloco *begin — [exception —] end* Define uma sequência de instruções, com tratamento opcional de excepções. Uma excepção pode ser criada (lançada) usando a instrução *raise*.

Instrução *if* Permite definir código executado condicionalmente (estão definidas as seguintes variantes: *if — then*, *if — then — else*, *if — then — elseif*).

Instrução *select into* Atribui o resultado de uma interrogação à base de dados a uma variável. Ao contrário do PL/SQL normalizado, esta instrução pode ser usada para obter o primeiro resultado de uma pergunta que tenha múltiplas respostas.

Instruções *update*, *insert* e *delete* Modifica o estado da base de dados.

Função *notify* Envia uma mensagem a um utilizador — permite especificar o destinatário da mensagem, o transporte a utilizar, e o conteúdo da mensagem a transmitir.

Função *newid* Gera um identificador único que pode ser usado como identificador de um registo. A execução de cada invocação desta função numa transacção móvel devolve o mesmo valor em todas as suas execuções (no cliente e no servidor).

Instruções *commit* e *rollback* A instrução *commit* conclui a execução da transacção móvel com sucesso (e torna persistente as alterações efectuadas à base de dados). A instrução *rollback* aborta a execução da transacção móvel (e desfaz as alterações efectuadas à base de dados). Ao contrário do que é usual, a execução destas instruções permite especificar uma sequência de valores (resultados) a devolver à aplicação.

Blocos *on commit* e *on rollback* Definem código a ser executado após a transacção concluir a sua execução (de forma semelhante ao código de tratamento de excepções). Este código não pode incluir as instruções *commit* e *rollback*.

Uma transacção móvel é definida como um bloco anónimo com a seguinte estrutura.

```
[  declare
     -- Secção de declarações
     -- definição de tipos, variáveis e constantes        ]
   begin
     -- Secção executável
     -- instruções do programa
[  exception
     when ... then
     -- bloco a executar para tratamento de uma excepção  ]
[  on commit
     -- bloco a executar em caso de execução com sucesso  ]
[  on rollback
     -- bloco a executar em caso de execução falhada      ]
   end;
```

Por omissão, a execução de uma transacção móvel termina com sucesso se o caminho de execução alcançar o fim das instruções do programa (excluindo os blocos *on commit/rollback*), i.e., se não terminar numa instrução *commit* ou *rollback*. Neste caso, o bloco *on commit* será executado. Caso o processamento de uma transacção termine com a criação de uma excepção não tratada, a transacção móvel é abortada e o bloco *on rollback* é executado (caso exista).

Em relação ao PL/SQL, as instruções disponíveis para definir uma transacção móvel apresentam algumas limitações. Entre estas limitações incluem-se a definição de tipos compostos (*record*) e ancorados, a definição de ciclos, o acesso à base de dados usando *cursores*, e a definição de funções e procedimentos auxiliares que possam ser usados pelas transacções móveis. Estas limitações foram introduzidas apenas para simplificar a criação do protótipo do sistema Mobisnap, não representando nenhuma limitação do modelo do sistema apresentado nesta dissertação.

Bibliografia

[1] S. Agarwal, D. Starobinski, e A. Trachtenberg. On the scalability of data synchronization protocols for pdas and mobile devices. *IEEE Network (Special Issue on Scalability in Communication Networks)*, 16(4):22–28, 2002.

[2] D. Agrawal, A. El Abbadi, e R. C. Steinke. Epidemic algorithms in replicated databases (extended abstract). Em *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, págs. 161–172. ACM Press, 1997.

[3] Marcos Kawazoe Aguilera, Wei Chen, e Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[4] A. V. Aho, Y. Sagiv, e J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems (TODS)*, 4(4):435–454, 1979.

[5] Paulo Sérgio Almeida, Carlos Baquero, e Victor Fonte. Version stamps - decentralized version vectors. Em *Proc. of the* 22nd *International Conference on Distributed Computing Systems (ICDCS'02)*, págs. 544. IEEE Computer Society, Julho de 2002.

[6] Rafael Alonso, Daniel Barbara, e Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems (TODS)*, 15(3):359–384, 1990.

[7] Rafael Alonso e Henry F. Korth. Database system issues in nomadic computing. Em *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, págs. 388–392. ACM Press, 1993.

[8] Peter A. Alsberg e John D. Day. A principle for resilient sharing of distributed resources. Em *Proceedings of the 2nd international conference on Software engineering*, págs. 562–570. IEEE Computer Society Press, 1976.

[9] Yair Amir e Avishai Wool. Evaluating quorum systems over the internet. Em *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, págs. 26–35, Junho de 1996.

[10] Glenn Ammons, Rastislav Bodík, e James R. Larus. Mining specifications. Em *Proc. 29th ACM Symp. on Principles of programming languages*, 2002.

[11] Ronald M. Baecker, editor. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan Kaufmann Publishers, 1993.

[12] Arno Bakker, Maarten van Steen, e Andrew S. Tanenbaum. From remote objects to physically distributed objects. Em *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, págs. 47–52. IEEE Computer Society Press, Dezembro de 1999.

[13] Daniel Barbará. Mobile computing and databases - a survey. *Knowledge and Data Engineering*, 11(1):108–117, 1999.

[14] Daniel Barbará e Tomasz Imielinski. Sleepers and workaholics: caching strategies in mobile environments. Em *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, págs. 1–12. ACM Press, 1994.

[15] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. Em *Proceedings of the second annual ACM symposium on Principles of distributed computing*, págs. 27–30, 1983.

[16] R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, e G. Woetzel. Basic support for cooperative work on the world wide web. *International Journal of Human Computer Studies: Special issue on Novel Applications of the WWW*, 46(6):827–856, Spring de 1997.

[17] P. A. Bernstein, V. Hadzilacos, e N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[18] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[19] Bluetooth. `http://www.bluetooth.com`.

[20] Georges Brun-Cottan e Mesaac Makpangou. Adaptable replicated objects in distributed environments. Rapport de Recherche 2593, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt (France), Maio de 1995. `http://www-sor.inria.fr/publi/RCMFGCCOSDS_rr2593`.

[21] Diego Calvanese, Giuseppe De Giacomo, e Maurizio Lenzerini. On the decidability of query containment under constraints. Em *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, págs. 149–158. ACM Press, 1998.

[22] Michael J. Carey, David J. DeWitt, e Jeffrey F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[23] Miguel Castro, Atul Adya, Barbara Liskov, e Andrew C. Meyers. Hac: hybrid adaptive caching for distributed storage systems. Em *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, págs. 102–115. ACM Press, 1997.

[24] Per Cederqvist, Roland Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual`.

[25] U. Cetintemel, B. Özden, M. Franklin, e A. Silberschatz. Design and evaluation of redistribution strategies for wide-area commodity distribution. Em *Proc. of the* 21st *International Conference on Distributed Computing Systems*, págs. 154–164. IEEE Press, Abril de 2001.

[26] Bharat Chandra, Mike Dahlin, Lei Gao, e Amol Nayate. End-to-end wan service availability, Março de 2001.

[27] Jia-Bing R. Cheng e A. R. Hurson. Effective clustering of complex objects in object-oriented databases. Em *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, págs. 22–31. ACM Press, 1991.

[28] Gregory Chockler, Dahlia Malkhi, e Michael K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. Em *Proceedings of the* 21st *International Conference on Distributed Computing Systems*, págs. 11–20. IEEE Press, Abril de 2001.

[29] Ian Clarke, Oskar Sandberg, Brandon Wiley, e Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. Em *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, págs. 311–320, Junho de 2000.

[30] Brian A. Coan, Brian M. Oki, e Elliot K. Kolodner. Limitations on database availability when networks partition. Em *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, págs. 187–194. ACM Press, 1986.

[31] Cryptix. Cryptix jce implementation. `http://www.cryptix.org/`.

[32] Miguel Cunha. Protótipo de teste do sistema mobisnap. Relatório do projecto final de curso, Departamento de Informática, FCT, Universidade de Lisboa, Fevereiro de 2001.

[33] Miguel Cunha, Nuno Preguiça, José Legatheaux Martins, Henrique João Domingos, Fransisco Moura, e Carlos Baquero. Mobisnap: Um sistema de bases de dados para ambientes móveis. Em *Nas Actas da Conferência de Redes de Computadores 2001 (CRC'2001)*, Novembro de 2001.

[34] Grupo DAgora. Projecto dagora. `http://asc.di.fct.unl.pt/dagora`.

[35] Shaul Dar, Michael J. Franklin, Björn Jónsson, Divesh Srivastava, e Michael Tan. Semantic data caching and replacement. Em *Proc. VLDB'96*, págs. 330–341. Morgan Kaufmann, Setembro de 1996.

[36] Grupo DataBricks. Projecto databricks. `http://asc.di.fct.unl.pt/databricks`.

[37] Susan B. Davidson, Hector Garcia-Molina, e Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR)*, 17(3):341–370, 1985.

[38] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, e John Larson. Epidemic algorithms for replicated database maintenance. Em *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, págs. 1–12. ACM Press, 1987.

[39] Henrique João L. Domingos. *Suporte de Sessões de Trabalho Cooperativo Multi-Síncrono em Ambientes de Grande Escala - Estruturação dos Suportes de Comunicação, Colaboração e Coordenação*. Tese de doutoramento, Dep. Informática, FCT, Universidade Nova de Lisboa, Fevereiro de 2000.

[40] Henrique João L. Domingos, Nuno M. Preguiça, e J. Legatheaux Martins. Coordination and awareness support for adaptive cscw sessions. *CLEI Journal - Journal of Latino American CYTED RITOS Network, Special Issue of Collaborative Systems*, Abril de 1999.

[41] Paul Dourish. The parting of the ways: Divergence, data management and collaborative work. Em *Proceedings of the European Conference on Computer-Supported Cooperative Work ECSCW'95*, págs. 213–222. ACM Press, Setembro de 1995.

[42] Paul Dourish. Using metalevel techniques in a flexible toolkit for cscw applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(2):109–155, 1998.

[43] Paul Dourish e Victoria Bellotti. Awareness and coordination in shared workspaces. Em *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, págs. 107–114. ACM Press, 1992.

[44] Paul Dourish, W. Keith Edwards, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry, e James Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems (TOIS)*, 18(2):140–170, 2000.

[45] Sérgio Duarte, J. Legatheaux Martins, Henrique J. Domingos, e Nuno Preguiça. Deeds - a distributed and extensible event dissemination service. Em *Proceedings of the* 4th *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Maio de 2001.

[46] C. A. Ellis e S. J. Gibbs. Concurrency control in groupware systems. Em *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, págs. 399–407. ACM Press, 1989.

[47] Clarence A. Ellis, Simon J. Gibbs, e Gail Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[48] Dawson Engler, David Yu Chen, e Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. Em *Proc.* 18th *Symp. on Op. Sys. Principles*, págs. 57–72, 2001.

[49] H.R. Nielson F. Nielson e C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[50] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, e Alex Shvartsman. Eventually-serializable data services. Em *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, págs. 300–309. ACM Press, 1996.

[51] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedesand Daniel Hagimont, e Sacha Krakowiak. Design, implementation, and use of a persistent distributed store. Em *Advances in Distributed Systems*, volume 1752 de *Lecture Notes in Computer Science*, págs. 427–452. Springer, 1999.

[52] Michael J. Fischer, Nancy A. Lynch, e Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[53] T. Frank e M. Vornberger. Sales forecasting using neural networks. Em *Proc. ICNN'97*, volume 4, págs. 2125–2128, 1997.

[54] Michael J. Franklin, Michael J. Carey, e Miron Livny. Local disk caching for client-server database systems. Em *Proceedings of the* 19th *International Conference on Very Large Data Bases*. Morgan Kaufmann, Agosto de 1993.

[55] Carsten A. Gerlhof, Alfons Kemper, e Guido Moerkotte. On the cost of monitoring and reorganization of object bases for clustering. *ACM SIGMOD Record*, 25(3):22–27, 1996.

[56] David K. Gifford. Weighted voting for replicated data. Em *Proceedings of the seventh symposium on Operating systems principles*, págs. 150–162, 1979.

[57] Richard A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, Dezembro de 1992.

[58] C. Gray e D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. Em *Proc. of the* 12th *ACM Symposium on Operating systems principles*, págs. 202–210. ACM Press, 1989.

[59] Jim Gray, Pat Helland, Patrick O'Neil, e Dennis Shasha. The dangers of replication and a solution. Em *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, págs. 173–182. ACM Press, 1996.

[60] Jim Gray e Andreas Reuter. *Transction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[61] Saul Greenberg e David Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. Em *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, págs. 207–217. ACM Press, 1994.

[62] Saul Greenberg e Mark Roseman. Using a room metaphor to ease transitions in groupware. Technical Report 98/611/02, Department of Computer Science, University of Calgary, Alberta, Canada, Janeiro de 1998.

[63] Irene Greif e Sunil Sarin. Data sharing in group work. *ACM Transactions on Information Systems (TOIS)*, 5(2):187–211, 1987.

[64] Irene Greif, Robert Seliger, e William E. Weihl. Atomic data abstractions in a distributed collaborative editing system. Em *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, págs. 160–172. ACM Press, 1986.

[65] Carl Gutwin e Saul Greenberg. Effects of awareness support on groupware usability. Em *Proceedings of the SIGCHI conference on Human factors in computing systems*, págs. 511–518. ACM Press/Addison-Wesley Publishing Co., 1998.

[66] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page Jr., Gerald J. Popek, e Dieter Rothmeier. Implementation of the ficus replicated file system. Em *Proceedings of the Summer 1990 USENIX Conference*, págs. 63–71, Junho de 1990.

[67] Jörg M. Haake e Brian Wilson. Supporting collaborative writing of hyperdocuments in sepia. Em *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, págs. 138–146. ACM Press, 1992.

[68] D. Hagimont e D. Louvegnies. Javanaise: Distributed shared objects for internet cooperative applications. Em *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer, Setembro de 1998.

[69] Seth Hallem, Benjamin Chelf, Yichen Xie, e Dawson Engler. A system and language for building system-specific, static analyses. Em *Proc. 2002 Conf. on Programming language Design and Implementation*, págs. 69–82. ACM Press, 2002.

[70] Joanne Holliday, R. Steinke, Divyakant Agrawal, e Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), 2003.

[71] hsqldb group. hsqldb java database engine. `http://hsqldb.sourceforge.net/`.

[72] M. Hurfin, A. Mostefaoui, e M. Raynal. Consensus in asynchronous systems where processes can crash and recover. Em *Proceedings of the* 17th *Symposium on Reliable Distributed Systems (SRDS)*, págs. 280–286, Outubro de 1998.

[73] IEEE. Ieee 802.11. `http://grouper.ieee.org/groups/802/11`.

[74] ILOG. Ilog jsolver. `http://www.ilog.com/products/jsolver/`.

[75] Yannis E. Ioannidis e Raghu Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM Transactions on Database Systems (TODS)*, 20(3):288–324, 1995.

[76] ISO/IEC. Iso/iec 9075: Database language sql, international standard, 1992.

[77] Larry S. Jackson e Ed Grossman. Integration of synchronous and asynchronous collaboration activities. *ACM Computing Surveys (CSUR)*, 31(2es):12, 1999.

[78] Jin Jing, Abdelsalam Sumi Helal, e Ahmed Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys (CSUR)*, 31(2):117–157, 1999.

[79] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, e M. F. Kaashoek. Rover: a toolkit for mobile information access. Em *Proceedings of the fifteenth ACM symposium on Operating systems principles*, págs. 156–171. ACM Press, 1995.

[80] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozme, e I.Greif. Replicated document management in a group communication system. Em D. Marca e G. Bock, editores, *Groupware: Software*

for Computer-Supported Cooperative Work, págs. 226–235. IEEE Computer Society Press, Los Alamitos, CA, 1992.

[81] Alain Karsenty e Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. Em *Proceedings of the 13th International Conference on Distributed Computing Systems*, págs. 195–202. IEEE Computer Society Press, Maio de 1993.

[82] Peter J. Keleher. Decentralized replicated-object protocols. Em *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, págs. 143–151. ACM Press, 1999.

[83] Bettina Kemme e Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, 2000.

[84] Anne-Marie Kermarrec, Laurent Massoulié, e Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):248–258, Março de 2003.

[85] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, e Peter Druschel. The icecube approach to the reconciliation of divergent replicas. Em *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, págs. 210–218. ACM Press, 2001.

[86] Minkyong Kim, Landon P. Cox, e Brian D. Noble. Safety, visibility, and performance in a wide-area file system. Em *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST'02)*, págs. 131–144. USENIX Association, Janeiro de 2002.

[87] James J. Kistler e M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[88] M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work*, 3(3-4):359–378, 1995.

[89] Narayanan Krishnakumar e Arthur J. Bernstein. Bounded ignorance: a technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems (TODS)*, 19(4):586–625, 1994.

[90] Narayanan Krishnakumar e Ravi Jain. Escrow techniques for mobile sales and inventory applications. *Wireless Networks*, 3(3):235–246, 1997.

[91] Geoffrey H. Kuenning e Gerald J. Popek. Automated hoarding for mobile computers. Em *Proc. of the* 16th *ACM Symposium on Operating Systems Principles*, págs. 264–275. ACM Press, 1997.

[92] David Lacey, Neil D. Jones, Eric Van Wyk, e Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. Em *Proc. 29th ACM Symp. on Principles of Programming Languages*, págs. 283–294. ACM Press, 2002.

[93] Rivka Ladin, Barbara Liskov, Liuba Shrira, e Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[94] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[95] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[96] Yui-Wah Lee, Kwong-Sak Leung, e Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. Em *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, USA, Jun de 1999.

[97] Mary D. P. Leland, Robert S. Fish, e Robert E. Kraut. Collaborative document production using quilt. Em *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, págs. 206–215. ACM Press, 1988.

[98] Sheng Feng Li, Quentin Stafford-Fraser, e Andy Hopper. Integrating synchronous and asynchronous collaboration with virtual network computing. *IEEE Internet Computing*, 4(3):26–33, 2000.

[99] Olivier Liechti. Awareness and the www: an overview. *ACM SIGGROUP Bulletin*, 21(3):3–12, 2000.

[100] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, e L. Shrira. Safe and efficient sharing of persistent objects in thor. Em *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, págs. 318–329. ACM Press, 1996.

[101] Lotus. Lotus notes. `http://www.lotus.com`.

[102] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[103] Bruce M. Maggs, Friedhelm Meyer auf der Heide, Berthold Vocking, e Matthias Westermann. Exploiting locality for data management in systems of limited bandwidth. Em *IEEE Symposium on Foundations of Computer Science*, págs. 284–293, 1997.

[104] Microsoft. Microsoft access. `http://www.microsoft.com/office/access/default.asp`.

[105] Pascal Molli, Hala Skaf-Molli, Gérald Oster, e Sébastien Jourdain. Sams: Synchronous, asynchronous, multi-synchronous environments. Em *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, 2002.

[106] Mortbay. Jetty:// web server & servlet container. `http://jetty.mortbay.org/`.

[107] Lily B. Mummert e M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. Em *Proceedings of the USENIX Summer Conference*, págs. 279–289, Junho de 1994.

[108] Jonathan P. Munson e Prasun Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, Junho de 1997.

[109] Objectivity. Objectivity/db technical overview, 1999. `http://www.objectivity.com`.

[110] R. Oliveira, R. Guerraoui, e A. Schiper. Consensus in the crash-recover model. Technical Report TR-97/239, EPFL, Computer Science Department, Agosto de 1997. `http://lsewww.epfl.ch/~schiper/papers/1997/tr-97-239.ps`.

[111] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[112] Oracle. Pl/sql user's guide and reference - release 8.0, Junho de 1997.

[113] Oracle. Oracle8 enterprise edition, 1998.

[114] Oracle. Oracle8i advanced replication: An oracle technical white paper, Fevereiro de 1999.

[115] Oracle. Oracle8i lite replication guide - release 4.0, 1999.

[116] François Pacull, Alain Sandoz, e André Schiper. Duplex: a distributed collaborative editing environment in large scale. Em *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, págs. 165–173. ACM Press, 1994.

[117] Palm. Introduction to conduit development. `http://www.palmos.com/dev/support/docs/conduits/win/IntroToConduitsTOC.%html`.

[118] Christopher R. Palmer e Gordon V. Cormack. Operation transforms for a distributed shared spreadsheet. Em *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, págs. 69–78. ACM Press, 1998.

[119] D. Scott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, e Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, 1983.

[120] Vern E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, Abril de 1997.

[121] David Peleg e Avishai Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[122] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, e Alan J. Demers. Flexible update propagation for weakly consistent replication. Em *Proceedings of the sixteenth ACM symposium on Operating Systems Principles*, págs. 288–301. ACM Press, 1997.

[123] Shirish Phatak e B. R. Badrinath. Multiversion reconciliation for mobile databases. Em *Proceedings* 15th *International Conference on Data Engineering (ICDE)*, págs. 582–589. IEEE Computer Society, Março de 1999.

[124] E. Pitoura e G. Samaras. *Data Management for Mobile Computing*, volume 10. Kluwer Academic Publishers, 1998.

[125] Nuno Preguiça. Repositório de objectos de suporte ao trabalho cooperativo assíncrono. Tese de mestrado, Dep. Informática, FCT, Universidade Nova de Lisboa, Outubro de 1997.

[126] Nuno Preguiça, Carlos Baquero, Fransisco Moura, J. Legatheaux Martins, Rui Oliveira, Henrique Domingos, J. Orlando Pereira, e Sérgio Duarte. Mobile transaction management in mobisnap. Em *East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems for Advanced Applications, ADBIS-DASFAA 2000*, volume 1884 de *Lecture Notes in Computer Science*, págs. 379–386. Springer, 2000.

[127] Nuno Preguiça e J. Legatheaux Martins. Revisiting hierarchical quorum systems. Em *Proceedings of the* 21st *International Conference on Distributed Computing Systems*, págs. 264–274. IEEE Press, Abril de 2001.

[128] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, e Henrique Domingos. Reservations for conflict-avoidance in a mobile database system. Em *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, págs. 43–56. Usenix Association, Maio de 2003.

[129] Nuno Preguiça, J. Legatheaux Martins, e Henrique Domingos. Flexible data storage for mobile collaborative applications. Em *Proceedings of the Third European Research Seminar on Advances in Distributed Systems*, Abril de 1999.

[130] Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, e Sérgio Duarte. Data management support for asynchronous groupware. Em *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, págs. 69–78. ACM Press, 2000.

[131] Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, e Sérgio Duarte. Supporting disconnected operation in doors (position summary). Em *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, págs. 179. IEEE Computer Society, Maio de 2001. Uma versão estendida está disponível em `http://asc.di.fct.unl.pt/~nmp/papers/hotos-extended.pdf`.

[132] Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, e Sérgio Duarte. Supporting groupware in mobile environments. Relatório técnico TR-04-2002 DI-FCT-UNL, Dep. Informática, FCT, Universidade Nova de Lisboa, 2002.

[133] Nuno Preguiça, Marc Shapiro, e J. Legatheaux Martins. Sqlicecube: Automatic semantics-based reconciliation for mobile databases. Submitted for publication, 2003.

[134] Nuno Preguiça, Marc Shapiro, e Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Relatório técnico MSR-TR-2002-52, Microsoft Research, Cambridge (UK), Maio de 2002. `http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-2002-5%2`.

[135] Calton Pu e Avraham Leff. Replica control in distributed systems: An asynchronous approach. Em *SIGMOD Conference*, págs. 377–386, 1991.

[136] Changtao Qu e Wolfgang Nejdl. Constructing a web-based asynchronous and synchronous collaboration environment using webdav and lotus sametime. Em *Proceedings of the 29th annual ACM SIGUCCS conference on User services*, págs. 142–149. ACM Press, 2001.

[137] Michael Rabinovich, Narain H. Gehani, e Alex Kononov. Scalable update propagation in epidemic replicated databases. Em Peter M. G. Apers, Mokrane Bouzeghoub, e Georges Gardarin, editores, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 de *Lecture Notes in Computer Science*, págs. 207–222. Springer, 1996.

[138] Norman Ramsey e Előd Csirmaz. An algebraic approach to file synchronization. Em *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, págs. 175–185. ACM Press, 2001.

[139] David Ratner, Peter Reiher, Gerald J. Popek, e Geoffrey H. Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[140] David Howard Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. Phd thesis, University of California, Los Angeles, Los Angeles, CA, USA, Janeiro de 1998.

[141] Benjamin Reed e Darrell D. E. Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[142] Qun Ren e Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. Em *Mobile Computing and Networking*, págs. 210–221, 2000.

[143] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, e John Kubiatowicz. Pond: The oceanstore prototype. Em *Proceedings of the* 2nd *USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association, 2003.

[144] Tom Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[145] Mark Roseman e Saul Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(1):66–106, 1996.

[146] Antony I. T . Rowstron, Neil Lawrence, e Christopher M. Bishop. Probabilistic modelling of replica divergence. Em *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, págs. 55–60, Maio de 2001.

[147] Yasushi Saito e Marc Shapiro. Replication: Optimistic approaches. Relatório técnico HPL-2002-33, Hewlett-Packard Laboratories, Março de 2002.

[148] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, e Jacob Ofir. Deciding when to forget in the elephant file system. Em *Symposium on Operating Systems Principles*, págs. 110–123, 1999.

[149] M. Satyanarayanan. Fundamental challenges in mobile computing. Em *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, págs. 1–7. ACM Press, 1996.

[150] SavaJe. Savaje os 1.1. `http://www.savaje.com`.

[151] Yücel Saygn, Özgür Ulusoy, e Ahmed K. Elmagarmid. Association rules for supporting hoarding in mobile computing environments. Em *Proceedings of the* 10th *International Workshop on Research Issues in Data Engineering*, págs. 71–78, Fevereiro de 2000.

[152] Michael D. Schroeder, Andrew D. Birrell, e Roger M. Needham. Experience with grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems (TOCS)*, 2(1):3–23, 1984.

[153] João Costa Seco e Luís Caires. A basic model of typed components. Em *Proceedings of the ECOOP'2000* 14th *European Conference on Object-Oriented Programming*, volume 1850 de *Lecture Notes in Computer Science*, págs. 108–128. Springer, Junho de 2000.

[154] Haifeng Shen e Chengzheng Sun. Flexible notification for collaborative systems. Em *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, págs. 77–86. ACM Press, 2002.

[155] Hyong Sop Shim, Robert W. Hall, Atul Prakash, e Farnam Jahanian. Providing flexible services for managing shared state in collaborative systems. Em *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97)*, págs. 237–252. Kluwer Academic Publishers, Setembro de 1997.

[156] Jorge Paulo F. Simão, José A. Legatheaux Martins, Henrique João L. Domingos, e Nuno Manuel R. Preguiça. Supporting synchronous groupware with peer object-groups. Em *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, págs. 233–236, Portland, Oregon, USA, Junho de 1997. Usenix Association.

[157] Sun. Jsr-000053 java servlet 2.3 specification. `http://java.sun.com/products/servlet/`.

[158] Chengzheng Sun. Undo any operation at any time in group editors. Em *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, págs. 191–200. ACM Press, 2000.

[159] Chengzheng Sun e Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. Em *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, págs. 59–68. ACM Press, 1998.

[160] Carl Tait, Hui Lei, Swarup Acharya, e Henry Chang. Intelligent file hoarding for mobile computers. Em *Proceedings of the first annual international conference on Mobile computing and networking*, págs. 119–125. ACM Press, 1995.

[161] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, e C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. Em *Proc. of the* 15th *ACM Symposium on Operating systems principles*, págs. 172–182. ACM Press, 1995.

[162] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, e Brent B. Welch. Session guarantees for weakly consistent replicated data. Em *Proceedings of the*

third international conference on on Parallel and distributed information systems, págs. 140–150. IEEE Computer Society Press, 1994.

[163] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[164] A. Tridgell e P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, Junho de 1996. Available from `http://samba.anu.edu.au/rsync/`.

[165] Manolis M. Tsangaris e Jeffrey F. Naughton. On the performance of object clustering techniques. Em *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, págs. 144–153. ACM Press, 1992.

[166] Luís Veiga e Paulo Ferreira. Incremental replication for mobility support in obiwan. Em *Proc. of the* 22nd *International Conference on Distributed Computing Systems (ICDCS'02)*, págs. 544. IEEE Computer Society, Julho de 2002.

[167] Inês Vicente e Filipe Leitão. Base de dados discográfica cooperativa. Relatório do projecto final de curso, Departamento de Informática, FCT, Universidade de Lisboa, 1999.

[168] Gary D. Walborn e Panos K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. Em *Proceedings of the* 14th *Symposium on Reliable Distributed Systems*, págs. 31–40, 1995.

[169] WebGain. JavaCC: Java compiler compiler. `http://www.webgain.com/products/java_cc/`.

[170] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[171] Seth J. White e David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. Em *Proceedings of the* 18th *International Conference on Very Large Data Bases*, págs. 419–431. Morgan Kaufmann, Agosto de 1992.

[172] Ouri Wolfson, Sushil Jajodia, e Yixiu Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, 1997.

[173] Haifeng Yu e Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.