UNIVERSIDADE NOVA DE LISBOA
Faculdade de Ciências e Tecnologia
Departamento de Informática

Repositório de Objectos de Suporte ao Trabalho Cooperativo Assíncrono

Por

Nuno Manuel Ribeiro Preguiça

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Informática.

Orientador: Prof. Doutor José Augusto Legatheaux Martins

Lisboa
1997

Em memória dos meus avós

Sumário

Esta dissertação apresenta um repositório distribuído e replicado de objectos partilhados que visa servir de suporte ao trabalho cooperativo assíncrono. Neste tipo de actividade, cada participante produz as suas contribuições de forma independente (embora coordenada) e sem conhecimento *imediato* daquelas que estão a ser produzidas pelos outros, no sentido de alcançar um objectivo comum. Para tal, o repositório apresentado permite que os diversos participantes desenvolvam as suas contribuições sem restrições e que estas sejam conjugadas de forma eficiente e automática.

A arquitectura deste repositório é baseada num conjunto de servidores que replicam de forma optimista conjuntos de objectos relacionados, e num conjunto de clientes que fazem *caching* dos objectos que os utilizadores necessitam para continuar a sua actividade. Estas duas características permitem que o mesmo apresente uma elevada disponibilidade e escalabilidade, assim como possibilitam o suporte de trabalho desconectado.

As modificações introduzidas nos objectos por cada utilizador são guardadas e propagadas através da sequência de operações invocadas. Esta característica facilita, não só a determinação exacta da existência de alterações concorrentes em conflito, mas também a conjugação das diferentes modificações e a resolução dos conflitos surgidos. Esta conjugação é efectuada de forma específica para cada tipo de dados.

O repositório está estruturado em torno de um modelo de objectos que permite estender a noção de objecto tradicional através dum conjunto de novos componentes reutilizáveis que implementam diferentes semânticas de manipulação das operações e de resolução de conflitos.

Abstract

In this dissertation, we present a shared object repository to support asynchronous cooperative work. In this kind of work, each participant make her/his contributions in an independent (but nevertheless coordinated) way, without immediate knowledge of those being performed by others, to reach a common goal. The distributed repository presented allows several users to develop their contributions without any restriction, and merge them efficiently and automatically.

The repository's architecture is based on a group of servers which optimistically replicates sets of related objects, in order to promote high-availability and scalability. To enable disconnected operation, clients also cache key objects.

Changes made to objects by each user are recorded and propagated as sequences of updates. This enables the exact determination of conflicts among concurrently made updates. It also enables type-specific update merging and conflict-resolution.

The object repository is structured around an object model that extends traditional objects with a set of reusable components. These components implement different updates handling and conflict-resolution policies.

Agradecimentos

Quero agradecer a todos aqueles que, directa ou indirectamente, contribuíram para a realização deste trabalho. Este foi realizado com o apoio financeiro do Sub-Programa Ciência e Tecnologia do 2º Quadro Comunitário de Apoio, através da bolsa PRAXIS XXI/BM/6926/95.

Em particular, gostava de agradecer ao meu orientador, José Legatheaux Martins, não só pelo apoio e disponibilidade demonstradas, mas também pelos conselhos dados e oportunidades proporcionadas ao longo da realização deste trabalho.

Ao Jorge Simão, pelas intermináveis trocas de ideias, experiências e elementos bibliográficos, as quais contribuíram para o resultado desta dissertação.

Ao Henrique João, pelas discussões e disponibilidade demonstradas.

Ao Joaquim Jorge, pelas conversas sobre o *mundo real* do trabalho cooperativo.

Ao meus pais, pelo apoio e incentivo que sempre me deram e que só os pais sabem dar.

Aos meus amigos, e em particular à Elsa, pelo incentivo e compreensão demonstradas.

Índice

1. Introdução 1

1.1 Trabalho cooperativo 1

1.2 Aplicações cooperativas síncronas 2

1.3 Aplicações cooperativas assíncronas 2

1.4 Objectivo e contribuição 3

1.5 Resumo do trabalho realizado 4

1.6 Organização da dissertação 5

2. Apresentação da aproximação 7

2.1 Trabalho cooperativo e repositórios tradicionais 7

2.2 Replicação 11

2.3 Caching 12

2.4 Aproximação proposta 12

3. Sistemas relacionados 15

3.1 Rover 15

3.2 Bayou 16

3.3 Coda e Odissey 17

3.4 Ficus e Frigate 19

3.5 Lotus Notes e Domino 20

3.6 Thor 21

3.7 Sync 22

3.8 Resumo....22

4. Modelo do repositório de dados....25

4.1 Arquitectura e funcionamento do repositório....25

4.1.1 Arquitectura geral....25

4.1.1.1 Servidores....26

4.1.1.2 Clientes....28

4.1.2 Modelo de funcionamento do sistema....28

4.1.3 Nomes simbólicos....30

4.1.4 Modelo de remoção....31

4.1.5 Repositório e trabalho cooperativo....32

4.2 Modelo de objectos....33

4.2.1 Exemplo: criação e funcionamento do CoObjecto directório....36

4.2.2 Alternativas de componentes....40

4.2.2.1 Cápsula....40

4.2.2.2 Atributos....41

4.2.2.3 Log....41

4.2.2.4 Ordenação das operações....41

4.3 Considerações diversas....44

4.3.1 Integração com sessões síncronas....44

4.3.2 Administração, Controlo de Acessos e Segurança....45

4.3.3 Mecanismos de coordenação, *awareness* e notificação....46

5. Comunicação epidémica....49

5.1 Arquitectura Básica....50

5.1.1 Sistema de comunicações....51

5.1.2 Log de mensagens....51

5.1.3 Filiação....52

5.1.4 Política de comunicações....52

5.1.5 Ordenação das mensagens....53

5.2 Protocolos de sincronização 53
5.2.1 Algoritmo genérico 54
5.2.1.1 Comunicação síncrona bidireccional 54
5.2.1.2 Comunicação Assíncrona 55
5.2.2 Algoritmo epidémico simples 56
5.2.3 Algoritmo epidémico com libertação de recursos 59
5.3 Adaptação do sistema de comunicação epidémica ao repositório de CoObjectos 61
5.3.1 Adaptação dos protocolos de sincronização 62
5.4 Filiação 65
5.4.1 Adaptação dos protocolos de sincronização 66
5.4.2 Sincronização da filiação 67
5.4.3 Sincronização das referências emitidas por clientes 69
5.4.4 Protocolo de entrada no grupo 69
5.4.5 Protocolo de saída do grupo 70
5.4.6 Identificadores nas *views* 71

6. Implementação 73
6.1 Suporte da implementação 73
6.2 Arquitectura modular do sistema 74
6.2.1 API de interface do sistema 75
6.2.2 API interna do sistema 77
6.2.3 API de baixo nível do sistema 78
6.2.4 Outros módulos 79
6.3 Subsistemas de comunicação 79
6.3.1 Subsistema de comunicação cliente/servidor 80
6.3.2 Subsistema de comunicações servidor/servidor 83
6.4 Núcleos dos macro-componentes 86
6.4.1 Núcleo do cliente 86
6.4.2 Núcleo do servidor 87

6.5 Modelo de objectos 88
6.5.1 Cápsula 88
6.5.2 Atributos 89
6.5.3 Log 90
6.5.4 Ordenação das operações 91
6.5.5 Dados 92
6.5.6 Componentes do CoObjecto de filiação 93

7. Aplicações que utilizam o repositório de CoObjectos 95
7.1 CoObjecto hierárquico 95
7.1.1 MVSOOData 96
7.1.2 MVSOODataDir 97
7.1.3 naifVE 100
7.2 Bases de dados 102
7.2.1 Operações genéricas 103
7.2.2 Migração de código 104
7.3 Integração com programas actuais 104

8. Avaliação e conclusão 107
8.1 Avaliação do repositório e modelo de objectos associado 107
8.1.1 Desempenho 107
8.1.2 Funcionalidade 110
8.2 Possíveis evoluções futuras 111
8.3 Conclusões 112

Apêndice A - Protocolos de sincronização 115
A.1 Algoritmo epidémico simples 115
A.1.1 Comunicação síncrona 115
A.1.2 Comunicação assíncrona 115
A.2 Algoritmo epidémico com libertação de recursos 116

A.2.1 Comunicação síncrona 116
A.2.2 Comunicação assíncrona 116
A.3 Comunicação unidireccional 117

Apêndice B - Classes utilizadas na replicação dum volume 119

Apêndice C - Classes base do modelo de objectos 125
C.1 Cápsula 125
C.2 Atributos 128
C.3 Log 130
C.4 Ordenação das operações 131
C.5 Dados 132

Apêndice D - Definição dum documento de texto estruturado 135
D.1 Dados 135
D.2 Contentor 135
D.3 Folha 139

Bibliografia 141

Lista das figuras

Figura 2.1 – Evolução esperada duma edição cooperativa dum texto. Edições não conflituantes. 9

Figura 2.2 – Evolução esperada duma edição cooperativa dum texto. Edições conflituantes. 10

Figura 4.1 – Arquitectura geral do repositório de CoObjectos. 26

Figura 4.2 – Interface do repositório implementando espaço hierárquico de nomes simbólicos. 31

Figura 4.3 – Interface dos CoObjectos manipulados pelo repositório. 34

Figura 4.4 – Componentes do modelo de objectos e relações estabelecidas entre os mesmos. 35

Figura 4.5 – Esboço da implementação do CoObjecto directório. 37

Figura 4.6 – Esboço do código gerado a partir da definição do CoObjecto. 39

Figura 4.7 – Utilização do CoObjecto directório. 40

Figura 4.8 – Valor do *log* dum CoObjecto em duas réplicas diferentes. 41

Figura 4.9 – Grafo de precedências real das operações. 43

Figura 4.10 – Operações executadas durante uma sessão síncrona, e lista de participantes associados. 45

Figura 5.1 – Topologia dum conjunto de *principais*.. 50

Figura 5.2 - Arquitectura base dum sistema de comunicação epidémica. 51

Figura 5.3 – Protocolo genérico utilizado nas sessões de sincronização. 54

Figura 5.4 – Passo inicial numa comunicação assíncrona. 55

Figura 5.5 – Passo inicial alternativo numa comunicação assíncrona. 55

Figura 5.6 - Resposta ao passo inicial normal. 55

Figura 5.7 – Resposta a um passo intermédio, ou ao passo inicial alternativo. 56

Figura 5.8 – Integração do repositório de CoObjectos num sistema de comunicação epidémica. 62

Figura 5.9 – Alterações efectuadas ao algoritmo epidémico com libertação de recursos para permitir múltiplos CoObjectos. 63

Figura 5.10 – Passo inicial das sessões de sincronização no repositório de CoObjectos......................64

Figura 5.11 – Resposta ao passo inicial das sessões de sincronização no repositório de CoObjectos...65

Figura 5.12 – Passo inicial das sessões de sincronização no repositório de CoObjectos......................66

Figura 5.13 - Resposta ao passo inicial das sessões de sincronização no repositório de CoObjectos. ..67

Figura 5.14 – Protocolo de sincronização do CoObjecto de filiação. 67

Figura 5.15 – Exemplo de mudanças de filiação independentes e posterior sincronização..................68

Figura 5.16 – Evolução possível do filiação num volume. 72

Figura 6.1 – Organização modular dos componentes do repositório de CoObjectos. 75

Figura 6.2 – API de interface do repositório de CoObjectos. 76

Figura 6.3 – Identificador simbólico dum CoObjecto nas suas múltiplas formas. 76

Figura 6.4 – API interna do sistema..77

Figura 6.5 – API de baixo nível do sistema. 78

Figura 6.6 – API exportada pelos servidores para acesso pelos clientes..80

Figura 6.7 – Interface que as classes que representam as operações a invocar num servidor devem implementar. 81

Figura 6.8 - Interface que as classes que representam os protocolos a estabelecer entre servidores devem implementar...84

Figura 6.9 – Interface que as classes que definem acções a executar no decorrer dum protocolo devem implementar. 85

Figura 6.10 – Cápsula do CoObjecto que representa um ficheiro..89

Figura 6.11 – Métodos dos objectos de atributos relacionados com a libertação de recursos. 90

Figura 6.12 – Classe base dos executores das operações..91

Figura 6.13 – Interface de definição da política de sincronização dum volume. 93

Figura 7.1 – Interface do espaço de subobjectos com múltiplas versões...96

Figura 7.2 – Organização do espaço de subobjectos...97

Figura 7.3 – Interface da organização estrutural do espaço de subobjectos...98

Figura 7.4 – Objectos definidores da estrutura do espaço de subobjectos...99

Figura 7.5 – Edição dum documento de texto estruturado usando o *naifVE*. 100

Figura 7.6 – Interface de introspecção dos contentores e de definição de possíveis componentes. 101

Figura 7.7 – Interface de introspecção das folhas. 101

Figura 7.8 – Interface a implementar pelos editores dos subobjectos 102

Figura 7.9 – Interface de introspecção da linearização. 102

Lista das tabelas

Tabela 3.1 – Princípios arquitecturais básicos. 23

Tabela 3.2 – Soluções para o tratamento de modificações concorrentes. 24

Tabela 5.1 – Evolução possível dos vectores usados no algoritmo epidémico sem acerto dos números de ordem. 60

Tabela 6.1 – Opções possíveis na obtenção dum CoObjecto (*getObject*). 86

Tabela 8.1 – Desempenhos de leitura e escrita de CoObjectos do repositório. 108

Tabela 8.2 – Desempenhos de leitura e escrita no sistema Java. 108

Tabela 8.3 – Desempenho da invocação remota de métodos, utilizando RMI, no sistema Java. 108

Tabela 8.4 – Desempenhos esperados baseando-se apenas nos tempos de serialização e de invocação remota de métodos. 109

Capítulo 1

Introdução

1.1 Trabalho cooperativo

A conjugação dum conjunto de diversos esforços individuais permite, normalmente, alcançar de forma mais rápida e eficaz os objectivos comuns, os quais seriam por vezes impossíveis de alcançar apenas através do esforço dum único indivíduo. Este princípio é posto em prática por muitos seres vivos, de entre os quais o Homem, sendo as grandes obras por si efectuadas um testemunho *vivo* do mesmo. Tudo o que possa servir como um catalisador dessa colaboração deve por isso mesmo ser encarado com particular atenção.

Nos sistemas informáticos, apesar de muitas vezes serem usados por equipas no desenvolvimento dum objectivo comum, o suporte para essa mesma colaboração, salvo certas áreas específicas, é na maior parte das vezes incipiente. A disciplina que visa desenvolver novos suportes que permitam uma colaboração mais fácil e eficaz é denominada Suporte Computacional para Trabalho Cooperativo (*Computer Support for Cooperative Work*), merecendo hoje um redobrado interesse tanto por parte da comunidade científica como empresarial.

O trabalho cooperativo pode ser dividido em duas grandes classes:

- Síncrono, no qual os diversos participantes trabalham durante o mesmo período de tempo e têm conhecimento imediato do trabalho produzido pelos outros (por exemplo, o trabalho realizado durante uma reunião);
- Assíncrono, no qual os diversos participantes não trabalham necessariamente no mesmo período de tempo e principalmente não têm conhecimento imediato do trabalho produzido pelos outros participantes (como, por exemplo, no desenvolvimento conjunto dum projecto de *software*). Uma subclasse importante deste tipo de trabalho é o *workflow*, no qual um conjunto de dados segue um

caminho pré-determinado (em função dos próprios dados e das alterações que os mesmos vão sofrendo) entre os diversos participantes.

Estas classes não devem ser, no entanto, entendidas como alternativas, mas antes como complementares, pois o desenvolvimento dum projecto alternará períodos de trabalho síncrono e assíncrono. Além disso, elas representam apenas as extremidades dum espectro, que varia em função do tempo que leva a que um participante tenha conhecimento do trabalho produzido pelos outros. As aplicações onde se pode efectuar o controlo e a variação desse período de tempo chamam-se multi-síncronas.

Se a ideia de produzir *software* que permita a colaboração eficiente entre um conjunto de pessoas parece bastante atractiva, as técnicas para o conseguir são, em geral, algo complexas ou então pouco clarificadas. Por esta razão, as ferramentas existentes não atingem, geralmente, o grau de qualidade e ergonomia necessários para que os seus potenciais utilizadores se decidam pela sua utilização. Algumas excepções existem, das quais se destacam o Lotus Notes [Lot97a] e algumas ferramentas de *workflow* [SMD97].

1.2 Aplicações cooperativas síncronas

Nas aplicações cooperativas síncronas, a característica comum fundamental é a existência dum espaço de trabalho partilhado, o qual é usado pelos diversos participantes para produzirem as suas colaborações. Como se pressupõe que o trabalho produzido por cada elemento é fortemente influenciado pelo trabalho produzido pelos outros, existe a necessidade dos diversos participantes terem uma visão coerente do estado desse mesmo espaço partilhado e de terem acesso às acções realizadas por todos os elementos no menor período de tempo possível. A concretização destes espaços partilhados pode ser baseada em diversas arquitecturas.

1.3 Aplicações cooperativas assíncronas

Nas aplicações cooperativas assíncronas os participantes dão o seu contributo de forma isolada, criando novos dados e alterando os já existentes. Usualmente, estas aplicações pressupõem a existência, pelo menos, dum repositório de dados que armazena os dados produzidos durante as sessões colaborativas.

Este repositório necessita de apresentar uma elevada disponibilidade por forma a possibilitar que, em qualquer momento, um qualquer participante possa produzir as suas contribuições. Esta necessidade é exacerbada pela possibilidade dos participantes estarem geograficamente distribuídos e em situação de desconexão (quer seja voluntária, referente a situações de tele-trabalho, quer seja involuntária, referente a falhas nas comunicações).

O repositório também deve estar preparado para lidar *convenientemente* com as contribuições produzidas concorrentemente. Tradicionalmente, é oferecido um sistema de controlo de versões, deixando aos utilizadores o posterior tratamentos das diferentes versões que vão sendo criadas. No entanto, existem alguns tipos de dados para os quais o tratamento automático das diferentes contribuições produz os resultados esperados pelos utilizadores. Desta forma, o repositório deve permitir o flexível tratamento das contribuições produzidas.

De seguida apresentam-se genericamente as características que se argumentam ser importantes num sistema de suporte ao trabalho cooperativo assíncrono, após o que se faz um resumo do repositório de dados proposto. Este repositório adopta as características apresentadas.

1.4 Objectivo e contribuição

O objectivo desta dissertação é contribuir para a compreensão das necessidades específicas do trabalho cooperativo assíncrono e apresentar propostas para a resolução de algumas dessas necessidades, com especial ênfase no repositório de dados partilhado que serve de suporte à colaboração.

Argumenta-se que a característica principal que um repositório de dados com este objectivo deve possuir é a de possibilitar a utilização directa das operações que provocam alterações nos dados, por oposição, à utilização dos estados que reflectem essas mesmas modificações. Esta característica permite determinar de modo preciso as modificações produzidas e a existência de modificações concorrentes conflituantes. Permite ainda, tratar/conjugar mais facilmente as diferentes alterações concorrentes (quando não são conflituantes basta aplicar todas as operações realizadas de forma sequencial). Este tratamento deve ser efectuado de forma flexível, i.e., dependente do tipo de dados. Adicionalmente, a utilização das operações constitui um meio eficiente de propagar as modificações.

Num repositório de dados que se baseie na utilização das operações, advoga-se a necessidade de fornecer componentes pré-definidos, e com múltiplas semânticas disponíveis, para manipulação das mesmas – modo como as operações são aplicadas aos dados, modo como são armazenadas, … Assim, a construção de novos tipos de dados é efectuada à custa destes componentes, utilizando as semânticas mais apropriadas para o tipo de dados que se está a definir.

A utilização dum modelo de replicação retardada (*lazy*) baseado num esquema de comunicação epidémica é usada como uma forma eficiente de propagar as operações entre um conjunto de servidores que replicam de forma optimista os dados dos participantes. Apresenta-se ainda a utilização dum esquema de *caching* (replicação secundária) efectuado nas máquinas dos clientes como a base para o suporte de clientes móveis. Este, conjugado com o anterior, permite uma elevada disponibilidade do sistema.

Nesta dissertação defende-se que as características apresentadas anteriormente permitem construir um sistema de suporte ao trabalho cooperativo assíncrono, que seja um catalisador efectivo do trabalho desenvolvido.

1.5 Resumo do trabalho realizado

Esta dissertação propõe um repositório distribuído de dados para suportar a construção de aplicações cooperativas assíncronas. As características fundamentais do repositório são as seguintes:

- Os dados nele depositados são estruturados sob a forma de objectos – chamados CoObjectos[1];
- Os CoObjectos estão agrupados em conjuntos (de CoObjectos relacionados), chamados volumes;
- O repositório é constituído por um conjunto de servidores que replicam volumes de CoObjectos;
- Os clientes acedem aos CoObjectos obtendo uma cópia privada; as modificações a esta cópia são efectuadas através da invocação dos métodos associados ao CoObjecto; a sequência das invocações é armazenada e enviada para um servidor quando o CoObjecto é *gravado*;
- Os servidores propagam entre si, de forma epidémica, as sequências de invocações produzidas pelos clientes;
- Estas invocações são aplicadas às réplicas dos servidores de forma específica para cada tipo de CoObjecto;
- Os CoObjectos são construídos através da junção de componentes reutilizáveis, que tratam dos pormenores relacionados com a manipulação das operações (e para os quais podem existir várias semânticas), com um componente específico, que implementa o tipo de dados que se pretende definir;
- Os clientes efectuam *caching* dos CoObjectos, de forma a permitirem a continuação do trabalho em situações de desconexão.

Adicionalmente, foi desenvolvido um editor que visa mostrar a adequação do repositório de CoObjectos às necessidades das aplicações cooperativas assíncronas.

---

[1] Aos objectos manipulados pelo repositório proposto chamam-se CoObjectos – objectos cooperativos – devido a destinarem-se ao suporte da cooperação e serem compostos por um conjunto de objectos tradicionais – o que tornava a sua descrição confusa, caso se chamassem simplesmente objectos.

1.6 Organização da dissertação

Após a introdução efectuada neste capítulo, esta dissertação continua, no capítulo 2, com a apresentação da aproximação tomada e suas motivações. No capítulo 3 analisa-se um conjunto de sistemas que apresentam diversas soluções para os problemas tratados nesta dissertação. No capítulo 4 é apresentado o modelo proposto para o repositório de CoObjectos apresentado nesta dissertação, assim como o modelo de objectos associado. No capítulo 5 discutem-se os protocolos de comunicação epidémica, com particular destaque para as adaptações específicas introduzidas para utilização no repositório, assim como a gestão da filiação dos servidores que replicam cada volume. No capítulo 6 apresenta-se uma descrição da implementação do protótipo do repositório de CoObjectos. Na capítulo 7 apresenta-se um editor que manipula um tipo de dados hierárquico (definindo um documento estruturado) que permite a modificação concorrente por múltiplos utilizadores, e que é baseado na utilização do repositório apresentado. Discutem-se ainda as possibilidades de implementar outros tipos de dados. No capítulo 8 conclui-se esta dissertação com uma avaliação do repositório, modelo de objectos e protótipo implementados, apresentação de propostas de evolução futura e algumas conclusões.

Capítulo 2

Apresentação da aproximação

Neste capítulo descreve-se de modo breve, a aproximação global proposta nesta dissertação para os problemas colocados pelo trabalho cooperativo assíncrono. Apresenta-se inicialmente uma reflexão sobre as características apresentadas pelos repositórios tradicionais e os requisitos postos pelas aplicações cooperativas. De seguida, apresentam-se de forma sucinta os problemas inerentes à replicação e ao *caching* de dados, os quais são utilizados para permitir uma elevada disponibilidade do repositório apresentado nesta dissertação. Por fim, apresentam-se e fundamentam-se as principais características de funcionamento e organização do repositório.

2.1 Trabalho cooperativo e repositórios tradicionais

Qualquer sistema que permita suportar o trabalho dum grupo de pessoas na obtenção dum objectivo comum e que forneça um ambiente partilhado, pode ser considerado um sistema de trabalho cooperativo. Esta definição, com a generalidade que enferma, permite englobar um elevado número de aplicações. Diversas taxinomias foram construídas para classificar estas aplicações e os requisitos específicos que apresentam, como se descreve, por exemplo em [EGR91].

Um requisito essencial para um conjunto significativo de aplicações (cooperativas ou não) é a existência dum repositório de dados onde se possam guardar de forma persistente os dados dos utilizadores. Nos sistemas tradicionais (i.e., sem suporte para o trabalho cooperativo) existem dois grandes tipos de repositório de dados: os sistemas de ficheiros e as bases de dados. Existem duas diferenças fundamentais entre ambos:

1. Nas bases de dados o sistema conhece a estrutura interna da informação, a qual está, geralmente, estruturada em registos contendo um conjunto pré-definido de campos. Pelo contrário, nos sistemas de ficheiros a informação é guardada sem qualquer tratamento semântico por parte do sistema[2].

2. Nas bases de dados os registos são acedidos através de interrogações (*queries*) e navegação. A sua modificação processa-se através da invocação de operações que utilizam a estrutura interna da informação. Nos sistemas de ficheiros, os ficheiros são acedidos através do seu nome simbólico. Devido ao desconhecimento da estrutura interna por parte do sistema, as aplicações são responsáveis por modificar o ficheiro de forma adequada.

Adicionalmente, existem diferenças básicas em relação às unidades de informação manipuladas. Os registos guardados numa base de dados têm geralmente pequena dimensão quando comparados com os ficheiros guardados num sistema de ficheiros. O número de registos existentes é, normalmente, muito mais elevado do que o número de ficheiros. Estas diferenças permitem optimizações específicas.

Com a utilização generalizada de linguagens orientadas para os objectos, começaram a surgir repositórios de objectos (apresentados em [Mad92]), com o objectivo de guardar, persistentemente, o estado dos objectos usados pelas aplicações. As bases de dados orientados para os objectos são bases de dados que manipulam objectos genéricos definidos pelos utilizadores em vez dos tradicionais registos. Desta forma, têm associados métodos que são utilizados na sua manipulação, mantendo, no entanto, as características fundamentais duma base de dados (em [AG89] são apresentadas as características gerais destes sistemas). Por vezes, os objectos presentes numa base de dados têm um identificador único através do qual podem ser acedidos.

Dois exemplos clássicos de aplicações cooperativas assíncronas são os editores e as agendas partilhadas. Em ambos os casos é necessário um repositório de dados para guardar a informação manipulada. A inadequação dos repositórios tradicionais torna-se evidente quando confrontados com a situação que se descreve de seguida (situações semelhantes poderiam ser apresentadas para diferentes aplicações):

- Na edição cooperativa dum documento deve-se permitir que vários utilizadores o modifiquem concorrentemente. As modificações introduzidas devem ser conjugadas de forma a criar o documento final, como exemplificado na figura 2.1.

  A utilização dum sistema de ficheiros é inadequada, pois não permite a conjugação das alterações efectuadas. Dependendo do sistema poderíamos ter uma de duas situações: no documento final reflectir-se-ia apenas uma das modificações; seriam criadas duas versões do

---

[2] Os sistema de ficheiros baseados em registos, presentes usualmente em máquinas de grande porte, efectuam, por vezes, algum tratamento da informação tratada. Por exemplo, os ficheiros indexados fornecidos pelo sistema DEC VMS [Sha91] podem ser acedidos por um conjunto de chaves, que se encontram presentes nos registos e que são geridas de forma automática. No entanto, estes sistemas apresentam um ficheiro como um conjunto/sequência de registos, onde cada um deve ser manipulado como um todo.

documento total, as quais poderiam ser unificadas de forma manual ou através de ferramentas automáticas. No entanto, por comparação simples das diferentes versões é muitas vezes impossível fazer a unificação, o que se pode constatar na figura 2.1, pela impossibilidade de determinar quais são as novas e as antigas versões dos capítulos modificados.

A utilização duma base de dados, em que cada capítulo pudesse ser modificado separadamente solucionaria este problema. No entanto, a definição dum documento estruturado numa base de dados tradicional é uma tarefa complexa. Assim, a solução deveria passar pela utilização duma base de dados orientada para os objectos.

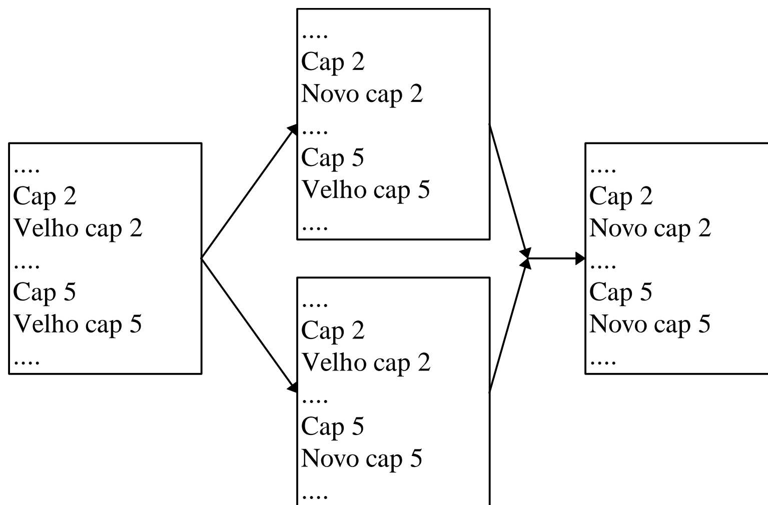

Figura 2.1 – Evolução esperada duma edição cooperativa dum texto. Edições não conflituantes são conjugadas.

Contudo, para o exemplo apresentado na figura 2.2, uma base de dados tradicional dificilmente levaria ao documento esperado. Esta situação fica a dever-se ao facto da utilização de transacções levar a que a segunda modificação seja abortada. Quando não se utilizam transacções apenas a última actualização fica registada. Para solucionar o problema seria necessário utilizar uma base de dados com versões. Estes sistemas apresentam, também, alguns problemas, pois são normalmente pouco flexíveis na definição das situações de conflito – usando habitualmente transacções. Assim, levam facilmente à explosão do número de versões, não permitindo a conjugação simples das diversas contribuições individuais.

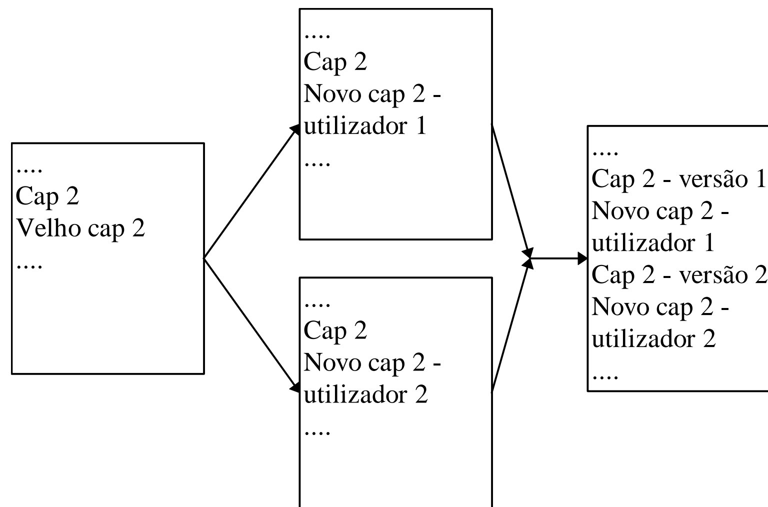

Figura 2.2 – Evolução esperada duma edição cooperativa dum texto. Edições conflituantes levam à criação de versões.

A inadequação dos repositórios tradicionais fica a dever-se, principalmente, ao facto de não permitirem um tratamento flexível das modificações efectuadas. Normalmente, existe uma política de tratamento das mesmas que privilegia a manutenção da consistência dos dados, em detrimento duma eficiente consideração das diferentes modificações.

Nos sistemas de ficheiros não existe, geralmente, qualquer controlo das modificações concorrentes – o ficheiro reflecte o último valor gravado. Associados a estes, existem, por vezes, sistemas de controlo de versões [Tic85] que permitem controlar as modificações concorrentes. Estes sistemas apresentam, no entanto, alguns problemas relacionados com o facto de trabalharem à granularidade do ficheiro: detecção de situações de falsos conflitos – as duas versões intermédias da figura 2.1 seriam apresentadas como conflituosas; falta de realce das situações de conflito efectivo – como o tratamento dos conflitos leva à criação de diferentes versões dos ficheiros totais, a determinação dos pontos de conflito efectivo torna-se difícil, necessitando do auxílio de ferramentas automáticas.

Nos sistemas de base de dados, o controlo da concorrência é efectuado, habitualmente, com recurso a transacções. Quando são detectadas duas modificações concorrentes em conflito, uma delas é abortada. Esta técnica apresenta dois problemas no âmbito do trabalho cooperativo: a definição de modificação conflituante é demasiado rígida (baseada em detecção de conflitos *write/write* e *read/write*), levando à detecção de falsos conflitos; o simples abortar duma transacção leva a que se perca o trabalho nela reflectido, o que no caso do trabalho cooperativo assíncrono não é razoável – numa edição cooperativa pode representar toda uma nova versão dum capítulo.

Um repositório que vise suportar o trabalho cooperativo assíncrono apresenta um conjunto de necessidades específicas diferentes dos repositórios tradicionais. Apresentam-se, de seguida e de forma resumida, os principais requisitos:

- Permitir a modificação concorrente dos mesmos dados por diferentes utilizadores;

- Permitir um tratamento flexível das diferentes modificações (o que passa pelos requisitos seguintes);
- Permitir a fácil determinação da existência de modificações conflituantes;
- Permitir a fácil conjugação das modificações não conflituantes;
- Permitir o tratamento flexível das modificações conflituantes.

Resumindo, o sistema deve estar preparado para a divergência dos dados manipulados por diferentes clientes, permitindo, no entanto, a sua sincronização de forma específica para cada tipo de dados e para cada situação particular (esta aproximação é defendida em [Dou95]).

Adicionalmente, o sistema deve apresentar uma elevada disponibilidade e suportar a existência de clientes móveis, com todas as peculiaridades que lhe são inerentes [Sat96a].

2.2 Replicação

A replicação constitui a chave para a elevada disponibilidade e escalabilidade dum serviço. Os repositórios de dados não são excepção. Assim, se um repositório de dados for constituído por um conjunto de servidores que mantem cópias dos dados, a disponibilidade do serviço mantem-se mesmo que algumas réplicas estejam indisponíveis. Esta solução permite, ainda, distribuir a carga pelos diferentes servidores, aumentando a eficiência na resposta aos clientes (e consequentemente aumentando o conjunto de clientes que o repositório pode servir).

A replicação traz consigo o problema da manutenção da consistência entre as diversas cópias dos dados. As diferentes estratégias de replicação, que foram sendo propostas, podem ser divididas em dois grandes grupos[3], face a este problema:

- Pessimistas, nas quais se evita que possam surgir problemas de consistência. Tal é conseguido limitando a possibilidade de alterar as réplicas, o que vem condicionar a disponibilidade completa do serviço de repositório de dados. Estas estratégias são justificadas por, face à posterior dificuldade em repor o estado de consistência dos sistemas, ser preferível evitar a sua possível divergência. Exemplos de estratégias pessimistas são a de réplica principal, esquemas de votação e de testemunhos (*tokens*).
- Optimistas, nas quais não se limitam as possibilidades de alteração das réplicas. Obviamente, estas técnicas apenas são passíveis de utilização nos casos em que seja possível a posterior unificação de réplicas que tenham eventualmente divergido. Este factor levou a que estas técnicas tenham na prática uma utilização muito reduzida. Um exemplo da utilização desta aproximação é o sistema *Refdbms 3.0* [Gol92a].

---

[3] Um estudo mais detalhado das diversas estratégias de replicação pode ser encontrado em [CDK94].

Como o repositório que se propõe nesta dissertação visa tratar eficazmente modificações concorrentes, pode pôr em prática uma aproximação optimista. Desta forma, permite-se maximizar a disponibilidade do repositório, pois o aumento do número de réplicas reflecte-se directamente no aumento da sua disponibilidade.

2.3 Caching

Duas situações que têm vindo a merecer uma redobrada atenção são:

- A computação móvel, onde se engloba a computação desconectada como situação extrema;
- O tele-trabalho no qual se permite aos trabalhadores realizar as suas tarefas no seu domicílio. As diferentes contribuições devem ser posteriormente coligidas. Devido aos custos das comunicações, este tipo de actividade terá normalmente as características da computação móvel desconectada.

Se a replicação é a chave para que um serviço apresente elevada disponibilidade e escalabilidade, o *caching* (por vezes também chamado replicação de segundo nível) é a chave para permitir o suporte de clientes móveis/desconectados. Assim, se os clientes fizerem *caching* dos dados que necessitam no seu trabalho, o serviço (de repositório) pode ser mantido mesmo em condições de total desconexão (em relação aos servidores). Deve, no entanto, ter-se em atenção que a extensão do *caching* é limitada, visando apenas manter cópias dum subconjunto limitado dos dados, de forma a que se possa manter o serviço durante um período que se supõe transitório e limitado no tempo. As cópias presentes na *cache* estão subordinadas às que se encontram nos servidores, i.e., a cópia presente no servidor representa a versão oficial dos dados. Obviamente que as modificações produzidas pelos clientes (e que podem estar reflectidas na *cache*) são propagadas para os servidores, de forma a criar uma nova versão oficial.

O repositório que se apresenta nesta dissertação recorre ao *caching* de forma a permitir o suporte de clientes móveis. Esta, associada com a replicação efectuada ao nível dos servidores, permite manter uma (quase) constante disponibilidade de serviço.

2.4 Aproximação proposta

Como já se referiu, a inadequação fundamental dos repositórios tradicionais prende-se com a sua inabilidade no tratamento de modificações concorrentes. Nesta dissertação apresenta-se um possível modelo alternativo, que tem em consideração os requisitos apresentados anteriormente. De seguida, apresentam-se as propostas fundamentais desse modelo:

- O repositório de dados armazena objectos – CoObjectos – cuja definição conhece. Estes CoObjectos são modificados através da invocação de métodos da sua interface. As modificações efectuadas por um utilizador são propagadas para o repositório como sequências de invocações de métodos.

Motivação: Tal como sucede nas bases de dados orientadas para os objectos, esta solução permite a definição de tipos de dados complexos. Possibilita, ainda, a adequação da granularidade das operações à das modificações efectuadas pelos utilizadores. Permite determinar, de forma rigorosa, quais as modificações efectuadas, evitando/minorando a detecção de falsos conflitos.

Reflexão: Os dados manipulados pelos utilizadores são, normalmente, tipados, e com uma estrutura bem definida – um documento LaTeX [Lam86] e um ficheiro fonte dum programa são exemplos bem significativos. Usualmente, é fácil definir as operações que permitem modificar esses dados (temos, pelo menos, modificações de estrutura e alteração dos valores dos elementos estruturais básicos). Desta forma, a definição de CoObjectos que representem os tipos de dados utilizados é possível, embora seja uma tarefa trabalhosa.

A propagação da sequência de métodos invocados, além de permitir uma fácil e rigorosa determinação das modificações, possibilita, muitas vezes, diminuir a informação que necessita de ser trocada, com a consequente diminuição dos recursos comunicacionais necessários e em algumas situações aumento do desempenho do sistema (os sistemas de ficheiros destinados especialmente a sistema paralelos de elevado desempenho têm por vezes uma organização em que guardam as alterações efectuadas aos ficheiros num *log* [AND+95]).

- As sequências de invocações propagadas pelos clientes para os servidores, associadas com informação que permita definir as suas dependências, são guardadas na cópia do CoObjecto presente no servidor. A aplicação destas operações, ao estado da cópia do servidor, é efectuada de forma específica para cada tipo de CoObjecto (podendo esta ser parametrizada individualmente para cada CoObjecto).

Motivação: Esta solução permite que cada tipo de CoObjecto defina uma política de tratamento das modificações de acordo com a sua semântica específica. Permite, igualmente, determinar quais são as modificações produzidas concorrentemente e quais destas são conflituantes.

Reflexão: Cada tipo de CoObjecto apresenta uma semântica específica, pelo que o tratamento das diferentes modificações deve ser executado, igualmente, de forma particular. Por exemplo, num CoObjecto que represente uma agenda distribuída, a aplicação das operações através duma ordem total é suficiente para garantir a consistência da mesma (os métodos que definem a marcação dum evento podem, em caso de impossibilidade da mesma, notificar os agentes envolvidos). Numa edição cooperativa, o método anterior já não é suficiente, devendo ser complementado com a detecção e tratamento de operações conflituantes – através da criação de diferentes versões de partes do documento.

A possibilidade de efectuar diferentes formas de tratamento está intimamente ligada com o armazenamento e datação que é efectuado das diferentes operações. É este armazenamento que permite a construção do grafo de precedências de cada operação e consequente

determinação de possíveis conflitos. No caso de existência de conflitos podem definir-se diferentes tipos de estratégias, as quais dependem, obviamente, da semântica do CoObjecto.

- O repositório fornece um modelo de objectos que decompõe os procedimentos necessários ao armazenamento e aplicação das modificações em diferentes componentes.

  Motivação: Esta solução permite a criação de componentes reutilizáveis, implementando diferentes semânticas para os diferentes problemas envolvidos no tratamento das modificações. Adicionalmente, permitem uma fácil construção de novos tipos de dados recorrendo aos componentes pré-definidos com as semânticas apropriadas.

  Reflexão: Na ausência dum modelo de objectos que permitisse a reutilização de componentes, a criação dum novo tipo de dados tornar-se-ia uma tarefa demasiado complexa. Existiria a necessidade de criar todos os procedimentos necessários ao armazenamento e aplicação das modificações de forma adequada, o que seria uma tarefa morosa e propensa ao aparecimento de erros.

  A existência deste modelo e dum conjunto de componentes pré-definidos não afecta, no entanto, a flexibilidade no tratamento das operações. Este facto deve-se à possibilidade de criação de novos componentes com diferentes semânticas.

A elevada disponibilidade e suporte para clientes móveis proporcionados pela replicação e *caching* no repositório de CoObjectos, permite que todos os participantes dum trabalho cooperativo possam produzir as suas contribuições individuais. As propostas anteriores têm como objectivo possibilitar um tratamento eficaz dessas mesmas contribuições.

As propostas apresentadas podem ser consideradas como uma evolução, em relação a um sistema de base de dados orientada para os objectos. Elas definem um sistema aberto, em que novas soluções podem ser integradas de forma a resolver problemas específicos. Adicionalmente, permitem criar uma solução específica para cada problema, não incorrendo, a resolução do mesmo, em custos que não sejam estritamente necessários – resultado da necessidade de fornecer uma solução generalista.

O repositório que se apresenta nesta dissertação destina-se a resolver os problemas específicos do trabalho cooperativo assíncrono. No caso em que se pretendem as mesmas funcionalidades que num sistema de ficheiros simples, assistir-se-á, obviamente, a uma degradação do desempenho. No entanto, em quase todas as áreas da informática foi existindo uma evolução baseada na troca de desempenho por novas funcionalidades, perda esta de desempenho que sempre foi compensada pela constante evolução do *hardware*, pelo que se pensa esta não deve ser apontada como óbice à evolução. Nos casos em que o desempenho é fundamental deve-se usar o sistema mais adequado.

Capítulo 3

Sistemas relacionados

Os problemas tratados nesta dissertação já foram abordados parcial ou totalmente por diversas equipas que desenvolveram diferentes sistemas. As soluções propostas variam conforme as aproximações tomadas e os problemas considerados primordiais. Neste capítulo apresentam-se, de forma sucinta e focando principalmente as soluções encontradas, os sistemas que se consideram mais representativos.

3.1 Rover

Rover [JLT+95] é um sistema de acesso a informação para clientes móveis, combinando duas técnicas fundamentais: objectos dinâmicos *móveis* (*relocatable dynamic objects* – RDO) e invocação diferida de procedimentos remotos (*queued remote procedure calls* – QRPC).

Os RDOs são objectos com interfaces bem definidas, podendo ser tão simples como os itens dum calendário, ou tão complexos como um módulo que encapsule parte duma aplicação. Eles podem ser partilhados pelas aplicações, i.e., pelos vários clientes. Para tal os RDOs são armazenados em servidores, sendo designados através de identificadores únicos. Os clientes (aplicações) acedem aos RDOs através dum esquema de *check in*/*check out*. Assim, podem importar os objectos para as suas máquinas criando uma réplica dos mesmos, invocar métodos localmente e exportá-los novamente para o servidor. Vários clientes podem importar o mesmo RDO concorrentemente.

Os QRPCs são utilizados para efectuar todas as comunicações entre clientes e servidores, permitindo uma forma de invocação remota diferida e assíncrona em que o sistema executa as comunicações nos momentos mais propícios. Desta forma, ao invocar um procedimento remoto, o cliente continua a sua execução sendo posteriormente notificado do resultado do mesmo. Esta característica permite a invocação simultânea de diversos procedimentos (o que foi usado num exemplo do sistema para construir um *proxy* de HTTP que permitia a fácil construção dum *browser* em que se podiam seguir várias ligações simultaneamente).

O resultado das operações aplicadas, localmente pelos clientes, aos RDOs têm um estatuto de tentativa. Estas operações são propagadas para os servidores, os quais mantém o estado oficial dos RDOs. Cada RDO tem um e um só servidor que mantém o seu estado oficial. As operações invocadas pelos clientes são aplicadas à cópia principal presente no servidor. Normalmente, os métodos do RDO começam por verificar se o mesmo foi modificado desde que o cliente o importou, o que é efectuado com o auxílio de *vectores versão* mantidos pelo sistema. No caso de se detectar um conflito, o método é responsável por tomar as acções consideradas correctas nessa situação – levando a uma resolução de conflitos dependente da aplicação. As acções efectuadas levam ao novo estado oficial do objecto, o qual pode não coincidir com o estado tentativa esperado nos clientes.

Os RDOs são usados igualmente para permitir exportar computações para os servidores, o que é particularmente útil quando se pretende aplicar operações sobre grandes quantidades de dados ou filtrá-los.

3.2 Bayou

O sistema Bayou [TTP+95,DPS+94] pretende fornecer um sistema de base de dados aos utilizadores de computadores móveis. Para tal, utiliza uma arquitectura em que as colecções de dados (organizadas em tabelas duma base de dados) são totalmente replicadas por um conjunto de servidores. A leitura dos dados por parte dos clientes é efectuada através do contacto com um qualquer dos servidores. As operações de escrita são igualmente executadas contactando apenas um servidor.

As operações de escrita enviadas pelos utilizadores para um servidor, são propagadas para os outros servidores através dum esquema epidémico. Cada servidor mantém dois valores da base de dados: o estável, resultado da execução das operações com número de ordem atribuído; e o tentativa, resultado da aplicação de todas as operações. Para cada base de dados existe um servidor que actua como primário e que se encarrega de atribuir um número de ordem a cada operação. Este número de ordem é posteriormente propagado para todos os servidores (usando o mesmo esquema epidémico). Os servidores, ao terem conhecimento duma operação não ordenada, aplicam-na, sendo o seu resultado reflectido no valor tentativa da base de dados. Ao terem conhecimento do número de ordem duma operação e caso as anteriores já tenham sido aplicadas, a operação é executada sobre o valor estável da base de dados, sendo o seu resultado nele reflectido. Adicionalmente, os servidores podem necessitar de actualizar o valor tentativa da base de dados, de acordo com a ordem estabelecida para a operação.

As operações de escrita neste sistema são constituídas por três partes: conjunto de dependências; actualização a executar; procedimento de unificação. O conjunto de dependências permite definir, de forma exacta e específica para cada operação executada, a ocorrência dum conflito. É constituído por um conjunto de interrogações (*queries*) sobre a base de dados e pelos respectivos resultados esperados. A aplicação duma operação começa por verificar se os resultados obtidos para as interrogações

especificadas no conjunto de dependências correspondem aos resultados esperados. Caso correspondam, a actualização especificada na operação é executada. No caso de os resultados obtidos diferirem dos esperados, executa-se o procedimento de unificação. Este procedimento, em função dos valores actualmente presentes na base de dados, devolve, se possível, uma actualização alternativa a ser aplicada.

Para que os utilizadores tenham uma visão dos dados replicados consistente com as suas próprias acções, o sistema permite a selecção dum conjunto de garantias de sessão. Assim, o sistema garante que as leituras e escritas executadas por um utilizador obedecem a determinadas condições – por exemplo, o utilizador nunca lerá um valor da base de dados anterior a outro valor já lido (o que seria possível contactando servidores diferentes). Adicionalmente, os utilizadores podem escolher qual o valor da base de dados que pretendem consultar – o tentativa ou o estável.

O controlo de acessos é efectuado com a granularidade das colecções de dados, as quais são igualmente a unidade de replicação. Para efectuar a autenticação dos clientes usa-se um esquema de chaves públicas, sendo o acesso aos dados efectuado através de certificados de acesso.

3.3 Coda e Odissey

O sistema Coda [SKK+90,KS92], descendente do AFS (ver, por exemplo, [CDK94]), é um sistema de ficheiros distribuído para ser usado num ambiente de larga-escala com suporte para clientes móveis. Apresenta uma elevada disponibilidade de acesso aos ficheiros e uma elevada tolerância às possíveis falhas existentes nos servidores e no sistema de comunicação. Para tal utiliza dois mecanismos fundamentais: replicação nos servidores – cada volume (conjunto de ficheiros formando um espaço hierárquico) é replicado por um conjunto de servidores; *caching* nos clientes – os clientes mantém cópias dos ficheiros de que necessitam.

Para cada volume, os clientes mantém dois conjuntos: VSG – conjunto dos servidores que contém réplicas desse volume; e AVSG, subconjunto dos servidores pertencentes ao conjunto anterior e que se encontram acessíveis. Quando os clientes pretendem aceder a um ficheiro (chamada do sistema *open*) do qual não têm uma cópia local, contactam os servidores do AVSG para conhecerem quais deles têm a cópia mais recente. A partir dum destes servidores obtém uma cópia (da totalidade) do ficheiro, ficando o servidor responsável de lhe enviar notificações (*best effort*) no caso do mesmo ser alterado por outro cliente (o que invalida a cópia presente na *cache* do cliente). Se o cliente verificar que alguns dos servidores do AVSG contactados não têm a cópia mais recente, informa-os da situação. Os clientes contactam periodicamente os membros do VSG de forma a poderem actualizar o AVSG e a detectarem notificações perdidas. Quando um ficheiro acaba de ser modificado (chamada de sistema *close*), o cliente tem a responsabilidade de enviar para todos os elementos do AVSG o novo estado do mesmo.

Quando o AVSG é nulo, o cliente encontra-se numa situação de desconexão – que pode ter sido originada de forma voluntária ou devido a falhas. Nesta situação o cliente pode continuar o seu trabalho, acedendo apenas aos ficheiros presentes na sua *cache*. As alterações executadas são armazenadas num *log* persistente para posterior reintegração. Quando o cliente volta a ter acesso a algum servidor, inicia a fase de reintegração, durante a qual refaz as operações presentes no *log*.

Tanto durante a fase de reintegração, como durante a fase de verificação do estado das cópias dos servidores pertencentes ao AVSG (aquando da obtenção duma cópia dum ficheiro), pode-se verificar que existem duas modificações concorrentes. A resolução destas situações é executada automaticamente no caso dos directórios, pois o sistema tem informação semântica suficiente para a efectuar[4]. No caso dos ficheiros, o sistema permite a existência de programas de resolução específica. Estes programas são chamados transparentemente, tendo acesso a todas as cópias dum ficheiro de forma a procederem à sua unificação. No caso da unificação ser conseguida, o novo estado do ficheiro é transmitido para todos os membros do AVSG. No caso contrário, o ficheiro é marcado para resolução manual, ficando o seu acesso normal interdito. A resolução manual é executada nos clientes (aliás como todas as outras resoluções) estando as diversas versões do ficheiro organizadas numa hierarquia.

Para que os clientes tenham cópias dos ficheiros que serão necessários durante os períodos de desconexão, o sistema combina duas técnicas: histórias das referências; e informações (base de dados) fornecidas pelos utilizadores.

No sistema Coda existe, ainda, um modo de operação fracamente conectado, em que a largura de banda disponível para efectuar as comunicações com os servidores é reduzida. Neste modo, as operações dos clientes são colocadas no *log* (como na situação de desconexão), mas existe um processo que vai fazendo a sua reintegração conforme as possibilidades (de forma a não saturar as comunicações). Adicionalmente, as operações permanecem no *log* durante algum tempo de forma a que se possa fazer a sua compressão (quando um ficheiro é modificado sucessivamente, apenas o último estado é relevante).

Este sistema oferece igualmente um mecanismo de transacções, pelo que um conjunto de modificações podem ser consideradas como uma unidade para propósitos de detecção e resolução de conflitos.

O sistema Coda pretende mascarar o ambiente, de forma a que os utilizadores não se apercebam das variações que ocorrem no mesmo – o que permite continuar a usar as aplicações correntes. Os membros da equipa que o desenvolveu estão actualmente a investigar uma diferente aproximação aos problemas da computação móvel no sistema Odissey [Sat96b]. Este novo sistema tem por objectivo

---

[4] Esta resolução falha nos seguintes casos: se um novo nome colide com outro já existente; se um ficheiro/directório tiver sido actualizado por um cliente e apagado por outro; se os atributos dum directório tiverem sido modificados concorrentemente.

permitir às aplicações adaptarem-se conforme as mudanças que ocorrem no ambiente em que a computação ocorre. Além da interface para negociação dos recursos disponíveis, o sistema apresenta as seguintes ideias fundamentais: divisão do espaço de ficheiros consoante o seu tipo; gestão da *cache* nos clientes dependente do tipo de ficheiros; existência de conjuntos dinâmicos de ficheiros relacionados, os quais devem ser replicados conjuntamente. Os autores esperam incorporar o Coda como parte do Odissey.

3.4 Ficus e Frigate

O sistema Ficus [GHM+90], descendente do sistema Locus [WPE+83], é um sistema de ficheiros distribuído tal como o seu antecessor. Os ficheiros estão organizados em volumes que são replicados por um conjunto de servidores – os servidores não necessitam de replicar totalmente cada volume. Ao contrário do que acontece no sistema Coda, os clientes acedem directamente aos ficheiros a partir dos servidores, não possuindo cópias locais dos mesmos. Por esta razão, o suporte para clientes móveis passa pela existência dum servidor (replicando os volumes desejados) no computador móvel. Os servidores encarregam-se de comunicar entre si, dois a dois, de forma a sincronizarem os seus estados e propagar as alterações executadas. A topologia e a frequência com que estas comunicações ocorrem podem ser programadas.

As modificações produzidas concorrentemente nos mesmos ficheiros e directórios são detectadas aquando da comunicação entre dois servidores. A resolução dos conflitos detectados é semelhante à do sistema Coda (com a variação de ser efectuada só entre dois servidores e portanto entre apenas duas versões do ficheiro): os conflitos nos directórios são resolvidos automaticamente; os conflitos surgidos nos ficheiros são resolvidos por programas de resolução associados aos ficheiros, ou, no caso desta resolução automática falhar, os ficheiros são marcados "em conflito", impossibilitando-se a operação normal.

O sistema Ficus é construído à base do sistema *UCLA stackable layers* [HP94], em que as várias características do sistema são construídas por componentes de forma independente, o que permite a fácil introdução de novas características. Existe uma camada que trata, de forma transparente para os utilizadores, a existência de várias réplicas do mesmo volume.

O sistema Frigate [KP97], baseado igualmente nos *UCLA stackable layers*, define um sistema de ficheiros, em que os ficheiros são objectos tipados chamados documentos (o ficheiro associado pode ser usado para guardar o estado do objecto). Assim, cada documento tem associado uma classe e uma versão, as quais disponibilizam um conjunto de serviços que estendem as operações normalmente oferecidas pelo sistema de ficheiros. Estas extensões são guardadas num repositório externo, onde é efectuada uma completa gestão das diferentes versões existentes para uma mesma classe.

Neste sistema existe uma camada que intercepta as operações dirigidas a um ficheiro – documento –, e se encarrega de executar as operações definidas na classe associada ao mesmo. Para tal, o sistema

lança servidores para cada tipo de documentos, os quais ficam responsáveis por manipular esses documentos e executar as operações solicitadas. As classes que implementam tipos de documentos são programadas no sistema ILU do XeroxPARC. Em tempo de execução, as referências a objectos (documentos) armazenados no sistema são resolvidas e é fornecido um *proxy* para os mesmos (o qual implementa a sua interface). Os métodos invocados são transformados em invocações aos servidores associados aos documentos. As habituais chamadas de sistema dum sistema de ficheiros (*open*/*read*/*write*/*close*/...) podem ser invocadas através do *proxy* ou através das habituais chamadas de sistema. Desta forma, pode alterar-se o comportamento destas chamadas sem que os programas que as usam disso tenham conhecimento.

3.5 Lotus Notes e Domino

Lotus Notes [KBH+88,Lot97a] é um produto comercial de suporte ao trabalho cooperativo, que tem como unidade central um repositório de documentos. Os documentos manipulados pelo repositório são definidos por *forms*, que indicam os campos constituintes e os respectivos tipos de dados. Existem diversos tipos de dados pré-definidos, entre os quais um tipo contentor, que permite armazenar qualquer outro tipo de dados: imagem, folha de cálculo, ... O repositório está organizado em bases de dados, as quais são constituídas por conjuntos de documentos. Pode existir uma organização hierárquica dos documentos, i.e., quando se cria um documento pode-se definir que esse documento é uma resposta (um filho) de outro documento já existente. Os documentos são acedidos através de *views*, que exibem os seus valores para um conjunto de campos pré-determinados (podem definir-se diferentes *views* para uma mesma base de dados). Adicionalmente, existe a possibilidade de aceder aos documentos através de interrogações (*queries*).

O repositório do Lotus Notes é constituído por um conjunto de servidores que replicam bases de dados completas. Os clientes acedem aos documentos pertencentes a uma base de dados contactando um qualquer destes servidores. De forma a permitir a existência de clientes móveis, os documentos podem ser replicados nos clientes, o que é efectuado a pedido e pode seguir regras pré-estabelecidas.

No caso dos clientes móveis, os estados dos documentos replicados são sincronizados com os estados presentes num servidor quando tal é possível, i.e., quando os clientes se conectam. Os outros clientes actualizam os documentos directamente nos servidores. Os servidores sincronizam os seus estados periodicamente através de comunicações entre pares de servidores. Durante este processo, utilizam os identificadores de versão associados a cada campo dos documentos de forma a determinarem a informação que necessitam de trocar e a detectar possíveis conflitos. Durante os processos de sincronização (entre um cliente e um servidor – incluindo o envio dum novo estado dum documento –, entre dois servidores), a detecção de actualizações conflituosas dá origem à criação de

diferentes versões dos documentos[5], as quais são geridas pelo repositório (normalmente as diferentes versões dum mesmo objecto estão organizadas como respostas ao documento considerado como primário). Aos utilizadores cabe a tarefa de criar, a partir das diversas versões, uma versão unificada (caso tal seja desejado).

Este sistema oferece ainda um conjunto muito completo de características de segurança, incluindo criptação de dados, autenticação e controlo de acessos. Existem diversas aplicações que são fornecidas com o sistema, as quais fazem uso do repositório descrito: gestão de correio electrónico, *workflow*, calendário de grupo, ...

O sistema Lotus Domino [Lot97b] é basicamente um ambiente de desenvolvimento de aplicações (especialmente dirigidas para a *Internet*) que utiliza o sistema Lotus Notes como base. Permite criar aplicações específicas a partir dum conjunto de serviços existentes e cujas características podem ser especificamente adaptadas para a aplicação em causa.

3.6 Thor

O sistema Thor [LAC+96,LDS94] é uma base de dados orientada para os objectos. Tem uma arquitectura cliente/servidor. Os clientes obtém e mantém cópias dos objectos presentes no servidor de forma a permitir a continuação do trabalho em situações de desconexão (e a melhorar o desempenho).

Os objectos geridos pela base de dados são genéricos, sendo definidos numa linguagem própria. Cada objecto tem um estado e um conjunto de métodos que o permitem manipular. As aplicações manipulam os objectos Thor em sessões, durante as quais executam sequências de transacções. Cada transacção pode incluir chamadas a métodos dos objectos e execução de código genérico, o qual é guardado pelo sistema. Quando uma transacção é executada num cliente móvel, ela é realizada sobre a cópia local do objecto e marcada como "*tentative commit*", sendo enviada para o servidor assim que possível. O sucesso duma transacção é determinada em último caso pela sua execução no servidor (de forma semelhante ao que acontece nas bases de dados tradicionais).

O sistema Thor possibilita a manipulação dos seus objectos através duma camada que permite expor as interfaces dos objectos a outras linguagens. Contudo, as execuções dos métodos dos objectos são sempre efectuadas no interior do sistema Thor – de forma a garantir um conjunto de características de segurança.

---

[5] Na versão original do Notes, cada documento apenas tinha uma versão, pelo que quando existiam modificações concorrentes, uma das modificações ganhava sobre as outras, sendo reflectida no estado final do documento em todas as réplicas da base de dados.

3.7 Sync

Sync [MD97] é um sistema que tem como objectivo permitir a criação de aplicações colaborativas que suportem clientes móveis. Está organizado numa arquitectura cliente/servidor, em que o servidor mantém o estado oficial dos objectos, e a lista das alterações executadas. Os clientes mantém localmente uma cópia dos objectos que necessitam para realizar o seu trabalho. As alterações efectuadas são aplicadas localmente pelos clientes (excepto aquelas definidas como *once-only* – por exemplo o envio de uma mensagem de correio electrónico –, as quais apenas são aplicadas pelo servidor), sendo posteriormente propagadas para os servidores. Os clientes sincronizam-se periodicamente com o servidor, através da troca das operações relativas aos objectos dos quais o servidor é responsável. Desta forma, o cliente ao receber as novas operações pode gerar o estado corrente do servidor.

As modificações executadas concorrentemente e verificadas aquando das sessões de sincronização são resolvidas através do modelo de unificação. Este modelo define quais as acções a executar na presença de duas modificações concorrentes. O sistema permite definir este modelo através de matrizes que indicam a acção a executar aquando da ocorrência de duas operações concorrentes (para cada tipo de objectos é necessário definir uma matriz que indica a acção a tomar para cada par de operações possíveis). O modelo de unificação pode alternativamente ser definido através de um método genérico que implemente qualquer política.

Para criar objectos que sejam manipulados pelo sistema, a classe que define esse objecto deve derivar uma de duas classes, consoante se trate dum objecto atómico – para o qual devem definir modelos de unificação – ou dum objecto construído à custa de outros objectos. Os programadores têm ao seu dispor um conjunto de classes pré-definidas, que implementam tipos e estruturas de dados básicas, para as quais estão definidos modelos de unificação considerados correctos. Quando um objecto não é atómico, a unificação entre duas actualizações concorrentes é efectuada descendo na cadeia de invocações e unificando as alterações executadas ao nível dos elementos atómicos – esta regra geral pode ser substituída em qualquer objecto não atómico.

3.8 Resumo

Nas secções anteriores foram apresentados alguns sistemas que se consideraram representativos das diferentes aproximações tomadas na resolução dos problemas tratados nesta dissertação. Todas as soluções apresentadas se baseiam em técnicas optimistas, pois como foi referido em [GHO+96] e em [Kin96b], os sistemas de trabalho cooperativo, mormente aqueles que pretendem suportar clientes móveis, não podem adoptar as tradicionais técnicas de controlo de concorrência pessimistas baseadas em *locks*. Nas tabelas 3.1 e 3.2 apresenta-se um resumo das principais soluções apresentadas pelos diversos sistemas.

Sistema	Entidade manipulada	Suporte para desconexão	Organização dos servidores
Rover	Objectos genéricos.	Replicação dos objectos nos clientes.	Servidor único para cada objecto.
Bayou	Tabelas de bases de dados.	*Caching* parcial das tabelas + colocação de servidor nos clientes.	Servidor replicado, com modificações propagadas entre servidores de forma epidémica. Cada tabela tem um servidor que actua como primário.
Coda	Ficheiro não-interpretado.	*Caching* dos ficheiros nos clientes.	Servidores replicam volumes, sendo o estado dos mesmos actualizado por iniciativa dos clientes.
Ficus	Ficheiro não-interpretado.	Colocação de servidor no computador móvel.	Servidores replicam volumes sincronizando-se de forma epidémica.
Notes	Documentos constituídos por campos tipados.	*Caching* de documentos nos clientes.	Servidores replicam bases de dados, sincronizando-se de forma epidémica.
Thor	Objectos genéricos.	*Caching* de páginas contendo objectos nos clientes.	Servidor único para cada objecto que pode, no entanto, migrar entre servidores.
Sync	Objectos genéricos.	Replicação dos objectos nos clientes.	Servidor único para cada objecto.

Tabela 3.1 – Princípios arquitecturais básicos.

Muitos outros sistemas e soluções foram apresentados, principalmente para apenas alguns dos aspectos envolvidos. De seguida apresenta-se uma lista não exaustiva de outros trabalhos que se consideram interessantes, dos quais se realçam as características consideradas principais: Mushroom [Kin96a] e Corona [SWP+97] – espaços partilhados de objectos baseados na arquitectura cliente/servidor; COAST [KS97] – sistema para composição de objectos à custa de classes pré-definidas; SHORE [CDF+94] – base de dados orientada para os objectos implementando algumas características de sistema de ficheiros; JavaSpace [Sun97a] – sistema de persistência de objectos Java, incluindo a possibilidade de executar transacções sobre os mesmos; [CP93] – repositório de objectos replicado *primary*/*backup*; [DGS85] – *survey* sobre técnicas de manutenção da consistência em bases de dados replicadas.

Sistema	Modificações propagadas	Detecção de modificações concorrentes conflituantes	Resolução de modificações concorrentes
Rover	Invocação de métodos.	Os métodos são responsáveis por detectar a existência de alterações concorrentes podendo recorrer a "vectores versão" mantidos pelo sistema. As operações são aplicadas sequencialmente no servidor.	Os métodos são responsáveis por executar as acções desejadas quando detectam situações conflituantes.
Bayou	Operações incluindo conjunto de dependências e procedimentos de unificação.	Através do conjunto de dependências associado a cada operação.	Através do procedimento de unificação associado a cada operação.
Coda	Ficheiros completos.	Efectuada pelas máquinas dos clientes de forma automática através do recurso a "vectores versão". Podem existir várias versões dos ficheiros.	Automática para os directórios. Programas associados a ficheiros, que tentam unificar automaticamente as múltiplas versões. Manual, quando método anterior falha.
Ficus	Ficheiros completos	Automática, através de "vectores versão" aquando dos processos de sincronização entre servidores.	Automática para os directórios. Programas associados a ficheiros, que tentam unificar automaticamente as múltiplas versões. Manual, quando método anterior falha.
Notes	Estados dos campos modificados num documento	Automática, através de identificadores de versão associados a cada campo	Criação automática de diferentes versões dum mesmo documento.
Thor	Transacções consituídas por invocações de métodos de objectos e código genérico	Transacções serializadas no servidor (usa técnicas usuais das transacções – *read set*, *write set).*	Falha (*abort*) das transacções.
Sync	Cadeia de invocações de métodos, possivelmente agrupados em transacções	Alterações aplicadas de forma sequencial no servidor – verificação para cada operação daquelas executadas desde o momento da última sincronização do cliente.	Definido para cada par de operações qual a acção a executar.

Tabela 3.2 – Soluções para o tratamento de modificações concorrentes.

Capítulo 4

Modelo do repositório de dados

Neste capítulo é apresentado o modelo de repositório de dados proposto, visando, acima de tudo, responder às necessidades específicas do trabalho cooperativo assíncrono. As soluções propostas, são, no entanto, geralmente aplicáveis a qualquer sistema de repositório de dados. Na primeira secção apresenta-se a arquitectura e funcionamento do sistema. De seguida apresenta-se o modelo de objectos associado ao repositório. Para concluir este capítulo discutem-se alguns problemas não abordados nesta dissertação, mas que devem estar presentes num sistema que pretenda servir como suporte ao trabalho cooperativo.

4.1 Arquitectura e funcionamento do repositório

Nesta secção apresentamos a arquitectura e funcionamento do repositório de CoObjectos. De seguida apresenta-se a arquitectura geral do mesmo, a qual é constituída por um conjunto de servidores e clientes. Na subsecção 4.1.2 apresenta-se o funcionamento geral do sistema, i.e., o modo como os vários componentes arquitecturais interagem. Na subsecção 4.1.3 apresenta-se uma interface para o sistema que utiliza um espaço hierárquico de nomes simbólicos. Na subsecção 4.1.4 apresenta-se o modelo de remoção implementado no sistema e termina-se esta secção fazendo uma correspondência entre as entidades envolvidas no sistema e aquelas presentes num sistema cooperativo.

4.1.1 Arquitectura geral

No repositório de CoObjectos que se propõe, os dados dos utilizadores estão estruturados sob a forma de CoObjectos. A definição destes CoObjectos é conhecida pelo repositório, devendo seguir a organização proposta no modelo de objectos associado (que é apresentado na secção 4.2). Os CoObjectos estão agrupados em conjuntos denominados volumes. Cada CoObjecto está contido num e num só volume. Cada volume tem um identificador global único, e cada CoObjecto tem um

identificador único relativo ao volume. Logo, para cada CoObjecto, a concatenação do seu identificador relativo com o identificador do volume define o seu identificador global único.

O repositório de CoObjectos é constituído por um conjunto de servidores que replicam de forma fracamente consistente (optimista) volumes de CoObjectos. Adicionalmente, os clientes efectuam *caching* dos CoObjectos que supõem vir a ser necessários no trabalho dos utilizadores. Na figura 4.1 mostra-se esta arquitectura geral, a qual tem por objectivo maximizar a disponibilidade de acesso ao serviço de repositório. Nas subsecções seguintes descrevem-se de forma mais completa os componentes da arquitectura: servidores e clientes.

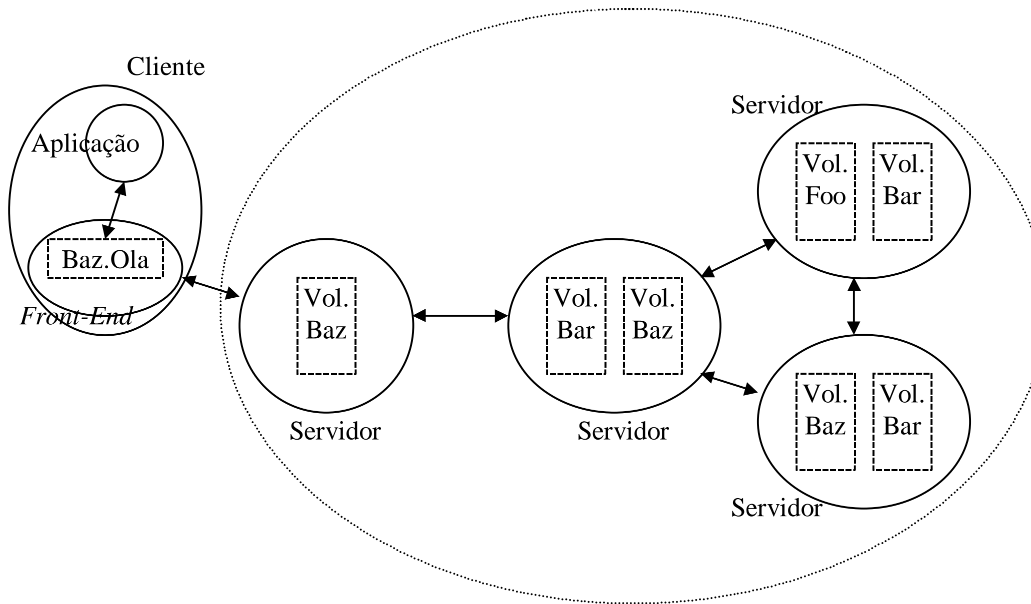

Figura 4.1 – Arquitectura geral do repositório de CoObjectos.

4.1.1.1 Servidores

Os servidores do repositório de CoObjectos armazenam volumes que contém os CoObjectos manipulados pelos utilizadores. O volume constitui a unidade de replicação do sistema. Cada volume pode ser replicado por um conjunto seleccionado e variável de servidores. Como se referiu no capítulo 2, a estratégia de replicação utilizada é optimista (*read any*/*write any*), permitindo a leitura e escrita com recurso a apenas uma (qualquer) réplica.

A existência de múltiplos servidores a replicarem um volume permite tolerar e mascarar algumas falhas existentes no sistema de comunicação e nos próprios servidores. Assim, desde que exista um servidor acessível, pode ter-se acesso ao serviço. A replicação contribui assim, para que o sistema aumente a sua tolerância às falhas.

Na ausência de falhas, a replicação pode originar uma melhor qualidade do serviço prestado pelos servidores devido a duas razões principais. Por um lado, permite a colocação de servidores em locais

mais próximos dos clientes (sendo que a noção de proximidade é ditada não pela distância física, mas antes pela qualidade do serviço de comunicações), o que pode levar a uma significativa melhoria no acesso ao sistema, principalmente quando de outra forma existisse a necessidade de recorrer a ligações lentas e/ou saturadas (como por exemplo, algumas ligações de WAN). Por outro lado, permite distribuir a carga pelos diferentes servidores, aumentando desta forma os recursos que cada um pode disponibilizar a cada cliente com a consequente melhoria do serviço oferecido. Esta melhoria do serviço prestado pelos servidores origina a possibilidade de mais clientes poderem usufruir do mesmo. Assim, a replicação contribui para a escalabilidade do repositório de CoObjectos.

A decisão de quais os servidores que replicam cada volume deve ser tomada criteriosamente de forma a que a replicação contribua o mais eficientemente possível para a tolerância a falhas e escalabilidade. Assim, deve ter-se em consideração, não só a localização dos clientes interessados nos volumes, mas também a topologia/qualidade da rede de comunicações. Através da monitorização e análise de utilização do repositório é possível construir um sistema que indique dinamicamente a localização ideal das réplicas dos volumes. Estas indicações podem servir para que o sistema faça automaticamente a relocalização, criação ou destruição de réplicas dos volumes, ou como meras indicações para os administradores do sistema. Uma parte significativa da complexidade associada ao problema advém da elevada incerteza que lhe está associada – a partir do comportamento passado dos intervenientes no sistema é necessário extrapolar o seu comportamento futuro. Nesta dissertação este problema não foi abordado, sendo a localização das réplicas feita por intervenção humana – tipicamente pelo administrador do sistema.

A estratégia de replicação optimista adoptada permite maximizar a disponibilidade do sistema, porque possibilita o acesso (leitura e escrita) com recurso a uma qualquer réplica. Esta aproximação apenas é possível devido ao modelo de objectos associado, o qual torna relativamente fácil a manutenção da coerência das várias réplicas através da conjugação das diversas modificações. Ao contrário do que sucede tradicionalmente nos sistemas de ficheiros distribuídos que suportam replicação, não é o conteúdo dos CoObjectos que é trocado entre os servidores, mas sim as modificações executadas[6]. A propagação destas alterações utiliza um esquema epidémico que será apresentado detalhadamente no capítulo 5. Este esquema não garante a coerência de todas as réplicas em nenhum momento, mas apenas que todas as réplicas terão acesso a todas as modificações executadas por todos os clientes. Assim, no caso das operações serem aplicadas nas diferentes réplicas por uma ordem coerente com a semântica das mesmas (não necessariamente a mesma ordem em todas as réplicas), garante-se a convergência final de todas as réplicas, i.e., que na ausência de novas modificações todas as réplicas atingem o mesmo estado.

---

[6] Como as modificações/alterações são traduzidas em invocações de métodos/operações do CoObjecto, usam-se indiferentemente os termos modificações, alterações e operações

4.1.1.2 Clientes

Os clientes podem ser divididos em dois subcomponentes: o *front-end* e as aplicações propriamente ditas. O *front-end* funciona como uma espécie de *proxy* do sistema, comunicando com os servidores de forma a servir os pedidos das aplicações. As aplicações têm ao seu dispor uma API[7] implementada por uma biblioteca que comunica com o *front-end* por forma a satisfazer as necessidades dos clientes.

O *front-end*, além de comunicar com os servidores para satisfazer os pedidos dos clientes, é responsável pelas seguintes tarefas: gestão da *cache*, garantindo que se mantêm cópias dos CoObjectos que se supõem ser necessários para que os clientes continuem o seu trabalho na impossibilidade de comunicar com os servidores; e armazenamento (*logging*) das alterações executadas pelos clientes aos CoObjectos – incluindo criações e destruições – para posterior propagação para os servidores no caso dos mesmos não estarem acessíveis. Estas duas actividades permitem o suporte dos clientes móveis e operação desconectada no sistema.

Para que os clientes possam continuar a executar o seu trabalho, existe necessidade da política de *caching* ser eficiente, i.e., de conseguir manter cópias dos CoObjectos que os clientes necessitam. Assim, a transposição simples das técnicas tradicionalmente utilizadas nos sistemas de computadores (para a *cache* de memória e do disco) pode não ser suficiente – embora em algumas situações o seja. Sobre o assunto existem vários estudos e propostas no domínio dos sistemas de ficheiros distribuídos [KS92,Kue94], que têm na sua base as seguintes características: monitorização dos acessos, mantendo cópias dos últimos CoObjectos acedidos (existe elevada probabilidade que um cliente volte brevemente a aceder aos mesmos CoObjectos); possibilidade dos utilizadores indicarem ao sistema os CoObjectos que pretendem ver replicados (os utilizadores sabem melhor do que ninguém os CoObjectos a que tencionam aceder); criação de conjuntos de CoObjectos associados cujas cópias são mantidas simultaneamente em *cache* (quando existem CoObjectos fortemente relacionados, é provável, que se um cliente acede a um deles, venha a aceder aos outros, por exemplo o ficheiro *.h* associado a um *.c* ou *.cc* ou *.cpp*). Este assunto não foi tratado em profundidade nesta dissertação, tendo sido implementada uma política simples baseada na técnica de *cache* tradicional conhecida por *least recently used* [Tan92].

4.1.2 Modelo de funcionamento do sistema

No repositório cada servidor mantém uma cópia de cada CoObjecto pertencente aos volumes que replica. Um utilizador (representado por uma aplicação), para ter acesso a um CoObjecto, necessita criar uma cópia privada desse mesmo CoObjecto. Para tal, precisa que o *front-end* do cliente tenha uma cópia do mesmo em *cache* – quando esta cópia não existe, ou está desactualizada, o *fornt-end*

[7] API (application progamming interface) é uma interface de programação, para criação de aplicações, que expõe os serviços disponibilizados por um sistema

encarrega-se de a obter a partir dum qualquer servidor[8]. A partir da cópia em *cache* do CoObjecto, o cliente cria a sua cópia privada, a qual manipula indiscriminadamente – como normalmente manipularia um qualquer objecto. Para tal, invoca os métodos desejados, os quais são reflectidos imediatamente no estado da sua cópia do CoObjecto. Posteriormente, o utilizador pode gravar as alterações produzidas – ou simplesmente esquecê-las. Temos assim, um modo de acesso do tipo *get*/*modify locally*/*put*, o qual é normalmente utilizado em todos os programas de edição de dados, dos quais os editores/processadores de texto são um exemplo significativo.

A gravação das alterações executadas a um CoObjecto é alcançada através do envio para um (qualquer) servidor da sequência de invocações executadas e de informação adicional que permita estabelecer as suas dependências – resumo das operações reflectidas no estado inicial (ver capítulo 5). Todas as informações anteriores são obtidas de forma transparente – os utilizadores usam os CoObjectos como se se tratassem de objectos normais. Como já se referiu, o *front-end* encarrega-se de armazenar as modificações dos clientes, para posterior envio, no caso dos servidores não estarem acessíveis.

As informações relativas às modificações executadas pelos clientes são armazenadas nos servidores, para cada CoObjecto. Os servidores, quando comunicam entre si, para sincronizarem os seus estados, trocam aquelas de que ainda não tiveram conhecimento. Assim, em cada momento, cada servidor que replica um mesmo CoObjecto pode ter tido acesso a um conjunto diferente de modificações. Desta forma, num dado momento, o estado do mesmo CoObjecto pode ser diferente em diferentes servidores. No entanto, o sistema garante, que na ausência de falhas eternas (i.e., partições do sistema de comunicações que isolem eternamente subconjuntos de servidores que replicam um dado CoObjecto), todos os servidores terão conhecimento de todas as modificações. Assim, um CoObjecto tenderá a ter um estado comum em todos os servidores que o replicam (esta propriedade está dependente da forma como as modificações são aplicadas, e será discutida posteriormente).

O estado da cópia de cada CoObjecto num determinado servidor depende naturalmente das operações que reflecte e da ordem pela qual estas foram aplicadas. Assim, cada CoObjecto tem associado um componente – ver modelo de objectos – que se encarrega de ordenar a aplicação das operações conhecidas. Geralmente, pretende-se que as cópias dum CoObjecto presentes em diferentes servidores convirjam para um estado comum, pelo que é necessário garantir que as operações são aplicadas por uma determinada ordem (a ordem necessária depende da semântica das operações – por exemplo, se tivermos um CoObjecto que modele um número, e a única operação existente for a adição, as operações podem ser aplicadas por qualquer ordem, enquanto se estiver disponível igualmente a multiplicação já é necessário garantir certas restrições). Por vezes, é necessário adiar a aplicação de algumas operações para garantir que as mesmas são aplicadas pela ordem desejada, o que

[8] Assim, a cópia do CoObjecto a que o cliente tem acesso representa sempre o estado desse CoObjecto num servidor (possivelmente com alterações provocadas por anteriores modificações efectuadas nesse cliente, quando em situações de desconexão).

faz com que o estado dum CoObjecto num servidor possa não reflectir todas as operações conhecidas por esse servidor.

4.1.3 Nomes simbólicos

Como se referiu, ao nível do sistema os CoObjectos têm um identificador único relativamente ao volume onde estão inseridos. Contudo, os utilizadores dum repositório de dados pretendem geralmente aceder aos dados através de nomes simbólicos (como nos sistemas de ficheiros) ou por interrogação e navegação (como nas bases de dados). No repositório de CoObjectos apresentado, qualquer uma das soluções anteriores poderia ter sido utilizada, pois a base do sistema proposto é completamente independente do modo de acesso aos CoObjectos. A escolha recaiu, no entanto, na utilização duma interface implementando um espaço hierárquico de nomes simbólicos. A utilização de interrogações e navegação poderia ser obtida por implementação duma infra-estrutura compatível ou através do recurso a serviços disponibilizados por sistemas de gestão de bases de dados como sugerido em [BRS96].

A interface do repositório implementa um espaço de nomes simbólicos hierárquico, baseada no conceito de directório – de forma idêntica ao existente nos sistemas de ficheiros. Assim, para cada volume existe um CoObjecto directório que representa a raiz do espaço de nomes. Este CoObjecto tem um identificador *bem-conhecido* (*well-known*), idêntico para todos os volumes. Cada CoObjecto directório representa uma função que liga um conjunto de nomes simbólicos a um conjunto de CoObjectos – referenciados pelos seus identificadores. Estes CoObjectos podem ser, por seu turno, outros directórios – permitindo assim a construção do espaço hierárquico de nomes. A interface do repositório permite ainda a definição de *links* simbólicos ([Tan92]). Desta forma, um mesmo CoObjecto pode ser acedido através de diferentes nomes simbólicos.

A interface do repositório é conceptualmente independente do sistema de base. Todas as entidades manipuladas – directórios e *links* simbólicos – são construídas à custa de CoObjectos normais do sistema. Os nomes simbólicos são interpretados pela interface que se encarrega de os transformar nos correspondentes identificadores ao nível do sistema. Na figura 4.2 apresenta-se a organização conceptual do repositório, esboçando a execução dum pedido. Qualquer outra interface poderia ser definida sem necessidade de modificar o comportamento básico do repositório e dos CoObjectos nele depositados.

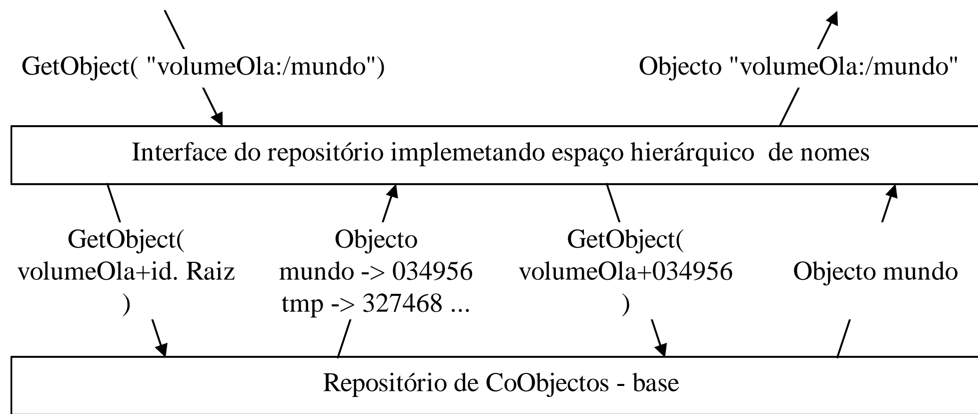

Figura 4.2 – Interface do repositório implementando espaço hierárquico de nomes simbólicos. Esta camada é responsável pela interpretação dos nomes simbólicos.

4.1.4 Modelo de remoção

Num sistema distribuído (com múltiplos utilizadores) a operação de remoção, quando aplicável, necessita de ser tratada cuidadosamente. Num sistema como o apresentado nesta dissertação merecem atenção as seguintes situações:

- Os identificadores dos objectos (CoObjectos no caso presente) devem ser únicos, de forma a que não exista qualquer dúvida sobre o objecto a que um dado identificador se refere. Desta forma, os identificadores dos objectos removidos apenas podem ser reutilizados no momento em que todas as referências ao objecto apagado tiverem sido destruídas. Como a determinação exacta desta condição é demasiado complexa, a solução geralmente utilizada é o recurso a identificadores únicos não reutilizados ([CDF+94]).
- Quando dados dois servidores que replicam um volume, se encontra uma cópia dum objecto em apenas um deles, pode-se estar na presença dum novo objecto ou de um objecto recém apagado. Nesta situação existe obviamente a necessidade de informação adicional que permita determinar qual a situação em que nos encontramos. Em muitos sistemas, a solução utilizada é guardar informação sobre os objectos que são removidos (*death certificates*). Esta solução levanta o problema da libertação dos recursos usados para guardar essa informação, sendo usualmente usada uma solução *ad hoc* baseada num período temporal ([Mac95]).
- Um cliente pode remover um objecto e outro concorrentemente alterá-lo. Nesta situação, os sistemas fazem usualmente reviver os objectos apagados, colocando-os num directório especial ([SKK+90,GHM+90]).

No repositório de CoObjectos proposto, a remoção é efectuada através do recurso a uma operação de remoção existente em todos os CoObjectos, e processa-se da seguinte forma (assumindo a utilização da interface de nomes simbólicos baseada em directórios apresentada anteriormente):

1. Aplicação da operação de remoção, que coloca o CoObjecto no estado de *removido*. Esta operação é propagada pelos mecanismos usuais de propagação de operações. Remoção da entrada respectiva no correspondente CoObjecto directório e adição na directório de CoObjectos apagados.
2. Após a operação de remoção ter sido propagada para todos as réplicas e ter decorrido um período pré-definido de tempo (que poderão ser alguns dias), se o CoObjecto se mantiver no estado de *removido*, os recursos a ele associados são libertos. Remove-se igualmente a respectiva entrada no directório de CoObjectos apagados.
3. No caso de ter sido aplicada uma operação concorrente ou posteriormente à aplicação da operação de remoção, o CoObjecto deixa o estado de *removido*, sendo transferido para o directório dos CoObjectos ressuscitados, continuando o seu funcionamento normal.

Além do exposto, o sistema proposto utiliza identificadores únicos não reutilizados. Os procedimentos atrás descritos visam garantir que nenhum CoObjecto é destruído enquanto existir algum cliente nele interessado. Assim, resolvem-se os problemas apresentados anteriormente: no caso de, para um dado CoObjecto, existir uma cópia apenas num servidor, essa cópia pode ser duplicada para o outro servidor sem problema, pois caso se trate dum CoObjecto no estado de *removido*, os recursos por ele ocupado serão posteriormente libertados pelo mecanismo normal; se um cliente submeter para um servidor novas operações sobre um CoObjecto que entretanto tenha passado à condição de *removido*, ele será facilmente retirado dessa condição, pois o estado do CoObjecto é mantido.

4.1.5 Repositório e trabalho cooperativo

As opções tomadas na definição do modelo de repositório apresentado foram fortemente influenciadas pelos objectivos específicos para que o mesmo foi criado. Assim, toda a arquitectura constituída por servidores e clientes tem por objectivo permitir uma elevada disponibilidade e escalabilidade, por forma a que todos os participantes dum trabalho cooperativo possam produzir, em qualquer momento, as suas contribuições.

Antecipa-se que para cada trabalho (sessão) cooperativo será criado um volume de CoObjectos, que conterá os CoObjectos produzidos durante esse trabalho. As contribuições dos diferentes participantes são reflectidas nos estados dos CoObjectos manipulados, pelo que a cooperação é conseguida através da modificação dos CoObjectos pertencentes a um volume. A utilização de um espaço hierárquico de nomes simbólicos baseou-se na antecipação de que os CoObjectos manipulados terão, em muitos casos, uma grandeza semelhante à dos ficheiros – por exemplo, existirá uma correspondência entre um ficheiro fonte dum programa e um CoObjecto contendo o mesmo código. Desta forma, permite-se que os utilizadores continuem a usar uma organização com a qual estão familiarizados e se sentem à vontade.

Em projectos cooperativos com um elevado número de participantes (e com uma grande quantidade de dados envolvida) existe tendência para a sua divisão em diferentes partes. Assim, para cada parte do trabalho global pode ser criado um volume, o qual precisa de ser replicado apenas para o subgrupo dos participantes envolvido. A visão global do projecto pode ser dada através da utilização de *links* simbólicos que apontem para as diferentes partes. Esta é também uma solução que permite reduzir as necessidades de escala do sistema para níveis realistas.

O modelo de remoção, ao garantir que nenhum CoObjecto será destruído, enquanto existirem utilizadores nele interessados, permite lidar convenientemente com divergências pontuais por parte dos diferentes participantes num trabalho cooperativo.

4.2 Modelo de objectos

Do ponto de vista do repositório todos os CoObjectos são idênticos, pois todos implementam uma interface que permite ao repositório aceder e manipular as operações executadas pelos clientes – esta interface é apresentada na figura 4.3. Como se referiu anteriormente, todos os componentes do repositório de CoObjectos lidam com estas operações de forma a garantir que: as novas operações produzidas por um cliente são propagadas para um servidor; todas os servidores têm acesso a todas as operações executadas[9]. As implementações de cada CoObjecto têm a obrigação de gerir as operações respectivas. Esta separação clara das responsabilidades atribuídas aos componentes do repositório e à implementação dos CoObjectos permite facilitar a implementação, depuração, optimização, extensibilidade e manutenção do sistema.

As responsabilidades atribuídas às implementações dos CoObjectos são, assim, as seguintes: armazenar e tratar adequadamente as operações executadas pelos clientes – quando na situação de cópia privada dum cliente; armazenar as operações entregues pelos componentes do repositório – que podem ser relativas a novas operações executadas por um cliente, ou operações obtidas a partir de outro servidor; aplicar as operações ao estado do CoObjecto por uma ordem que garanta a convergência dos estados das suas diferentes réplicas. Desta forma, a criação dum novo tipo de dados – CoObjectos – é muito mais complexa do que a criação simples dum objecto. Para que esta actividade, devido à sua complexidade, não se torne restrita a uma elite de programadores, propõe-se, igualmente, um modelo de objectos associado ao repositório de CoObjectos.

O modelo de objectos reflecte a divisão das responsabilidades atribuídas às implementações dos CoObjectos em componentes independentes, com interfaces bem definidas. Esta divisão permite a criação de implementações pré-definidas e reutilizáveis dos diferentes componentes, cada qual com as suas características próprias. Assim, os programadores podem concentrar-se no problema que

[9] Esta propriedade é alcançada através da execução de sessões de sincronização – capítulo 5. O sistema encarrega-se de actualizar a meta-informação associada a este processo.

pretendem resolver – a criação dum novo tipo de dados, traduzida num objecto *simples* – , utilizando os componentes pré-definidos para solucionar as responsabilidades adicionais atribuídas aos CoObjectos. Adicionalmente, permite uma mais fácil criação de novos componentes, pois isola os diversos problemas a resolver. Existem inúmeros exemplos de sistemas que apresentam este tipo de estratégia, nas mais diversas áreas, como por exemplo objectos distribuídos [BM95,HST+96] e protocolos de comunicação em grupos [RBM96].

```
public abstract class DSCSCapsule
{
        // Usada durante a troca de operações entre servidores para obter
        // operações ainda não observadas pelos parceiros
  public abstract LoggedOperation[] checkChanges( Timevector tv);

        // Usado no cliente para obter as operações executadas pelo utilizador e
        // que devem ser propagadas para o servidor
  public abstract OperationSeq checkChanges();
  public abstract void changesCommitted();

        // Operações de inserção de novas operações (sendo a primeira referente
        // a operações efectuadas nos clientes e a segunda a operações
        // transmitidas durante contacto com outros servidores)
  public abstract void insertOp( Operation op);
  public void insertOp( LoggedOperation[] ops);

        // Operação que determina a execução das operações armazenadas que
        // possam ser executadas segundo a ordem escolhida
  public abstract void executeOp();

        // Funções usadas para leitura e escrita dos CoObjectos num suporte
        // não-volátil - sistema de ficheiros
  public void writeObject();
  protected void readObject( DSCSAttrib localAttrib, ObjectHandle oH);

        // Operação que determina a limpeza do log de acordo com informação
        // actualmente disponível
  public abstract void cleanupOp();

  /* Outras funções pouco relevantes utilizadas na gestão do sistema */
}
```

Figura 4.3 – Interface dos CoObjectos manipulados pelo repositório. (O prefixo DSCS – Data Storage and Coordination Service – é reflexo da visão existente no início sobre o sistema que se iria implementar, a qual evoluiu não tendo os nomes do código acompanhado esta evolução).

Os componentes que fazem parte do modelo proposto são os seguintes:

- **Atributos**. Os atributos são constituídos pelas características comuns a todos os CoObjectos, entre as quais se incluem: a informação sobre qual o CoObjecto associado; a meta-informação necessária ao processo de replicação; a informação sobre o dono do CoObjecto, data de criação/alteração e outras. Além destas informações generalistas, é possível criar informação adicional específica para

cada CoObjecto, por extensão da definição base. Todos os CoObjectos contêm necessariamente um objecto de atributos. Este objecto pode ser manipulado de forma independente, da mesma forma que o podem os atributos num sistema de ficheiros.

- ***Log***. O *log* serve para guardar as operações executadas pelos utilizadores. Os métodos *checkChanges*, *changesCommited* e *insertOp* da interface do CoObjecto – DSCSCapsule – são geralmente redireccionadas para este componente sem qualquer tratamento. Os métodos usados variam caso se trate duma instanciação efectuada num cliente ou num servidor, o que pode levar a que, por questões de eficiência, existam implementações diferenciadas para cada um dos casos.

- **Ordenação (e aplicação) das operações**. Este componente serve para determinar a ordem pela qual as operações são aplicadas ao objecto dos dados. O método *executeOp* da interface do CoObjecto é normalmente redirigida para este componente sem qualquer tratamento. Este componente acede às operações que precisa de ordenar, através da interface do *log*.

- **Dados**. Objecto que encapsula as informações específicas e define as operações exportadas pelo tipo de dados que se define. É este o objecto com o qual o código das aplicações interage.

- **Cápsula**. Agrega os objectos que implementam um CoObjecto. Esses objectos devem ser instâncias de implementações dos componentes acima mencionados, e de outros que possam vir a ser considerados úteis. É através deste objecto e da sua interface (figura 4.2) que o repositório interage com os CoObjectos manipulados. Geralmente, este objecto limita-se a redirigir as invocações recebidas para os componentes apropriados.

Na figura 4.4 apresentam-se os componentes do modelo de objectos proposto com as relações que se estabelecem, normalmente, entre eles.

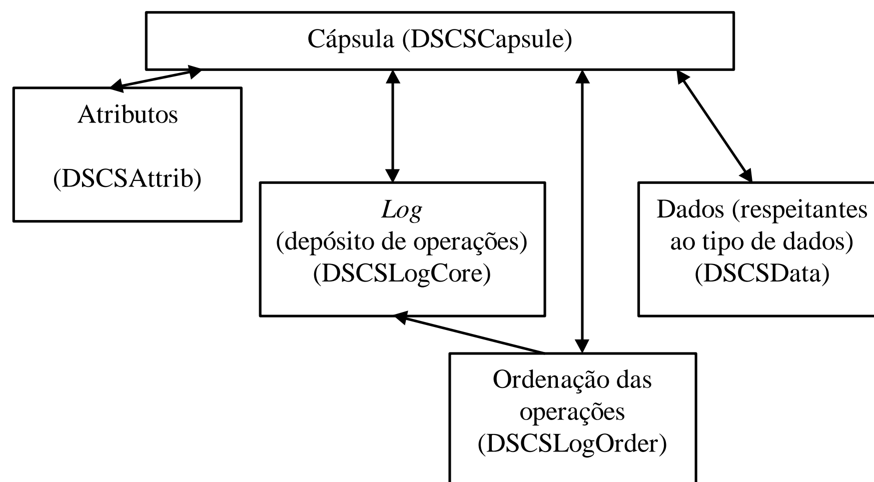

Figura 4.4 – Componentes do modelo de objectos e relações estabelecidas entre os mesmos. Uma seta indica que o objecto de origem tem conhecimento directo do objecto de destino.

A criação dum novo CoObjecto exige a criação do objecto de dados correspondente e a selecção das implementações dos outros componentes a utilizar. Esta selecção deve ser criteriosa, de forma a que o CoObjecto apresente o comportamento desejado – geralmente é necessário ter especial cuidado com o objecto de ordenação seleccionado, para que não só os estados das diferentes réplicas do CoObjecto convirjam, mas também que a aplicação das operações seja efectuada o mais rápido possível. Quando, para algum componente, não existe nenhuma implementação pré-definida que seja satisfatória, podem-se criar novas implementações que correspondam à especificação desejada. Antecipa-se, no entanto, que a necessidade de recorrer a esta solução seja pontual e esporádica, principalmente a partir do momento em que exista um número razoável de diferentes opções. De seguida explica-se, com recurso ao exemplo do CoObjecto directório, a forma como um CoObjecto é criado e como funciona o modelo proposto. Discutem-se ainda várias alternativas possíveis para os diferentes componentes.

4.2.1 Exemplo: criação e funcionamento do CoObjecto directório

O CoObjecto directório consiste basicamente numa sequência de pares ordenados (nome simbólico, identificador do CoObjecto relativo ao volume), tal que não existem dois pares com o mesmo nome simbólico. Disponibiliza três operações: *lookup*, que devolve o identificador dum CoObjecto dado o seu nome simbólico; *insertEntry*, que insere um novo par; *removeEntry*, que remove um par existente. No caso dum directório não existe necessidade de atributos específicos, nem de funcionalidades especiais ao nível do objecto de armazenamento de operações. Existe a necessidade de as operações serem aplicadas por uma ordem causal e total (causal para garantir que a inserção e remoção dum mesmo par apresenta o resultado esperado, total para garantir que a ordem final dos pares é igual em todas as réplicas, e que no caso de se tentarem inserir dois CoObjectos com o mesmo nome a resolução é efectuada da mesma forma – renomeando sempre o mesmo CoObjecto). Pretende-se, ainda, que as operações sejam aplicadas imediatamente, ainda que possa existir a necessidade de as desfazer e refazer posteriormente por forma a garantir a ordenação total. Além da decisão de quais os componentes a serem usados para construir o CoObjecto directório, é igualmente preciso criar o objecto de dados, ou seja, aquele que implementa as funcionalidades pretendida.

O código usado para definir os novos CoObjectos tem as seguintes adições relativamente à linguagem Java *standard* [Sun95]: a instrução **use** indica qual os componentes que se pretendem utilizar (esta parte poderia, alternativamente, estar num ficheiro de configuração) e a iniciação específica desses mesmos componentes; o modificador **loggable** aplicado aos métodos indica que a sua invocação deve ser registada no *log*; a *directiva* **undo** indica o código necessário para desfazer a aplicação duma dada operação (apenas necessário quando se pretendem usar ordenações que possam desfazer a aplicação das operações – ver 4.2.2.4) – neste caso, o valor retornado pela operação é usado como parâmetro do método de *undo*, de modo a que se possa guardar o estado necessário para que a operação seja desfeita.

Na figura 4.5 apresenta-se um esboço do código fonte que é necessário para criar o CoObjecto directório. Assim, é necessário definir quais os componentes seleccionados, o que é efectuado através da utilização das instruções **use**. O objecto de dados tem as seguintes diferenças, quando comparado com a definição dum objecto normal com as mesmas funcionalidades: identificação, com a directiva **loggable**, das operações que alteram o estado do objecto e que por isso devem ser registadas no *log*; definição do código necessário para desfazer uma operação, quando são utilizadas ordenações que fazem recurso a esta propriedade.

```
        // Cápsula composta pelos seguintes componentes:
        // atributos + log + ordenação + dados
use DSCSCapsule = DirectoryCapsule extends LogDataCapsule {
        // Código de iniciação dependente do tipo de dados
  public DirectoryCapsule( GlobalOID parent, GlobalOID current) {
        data = new DirectoryData( parent.objID, current.objID);
        data.link( this);
  }
}
        // Implementação standard do log
use DSCSLogCore = LogCoreImpl {}
        // Ordenação total (e causal) fazendo undo/redo simples
        // (i.e., sem explorar as propriedades das operações)
use DSCSLogOrder = LogTotalSimpleUndoRedo {}
        // Atributos baseados na proposta de Golding (ver capítulo 5)
        // para as notificações entre servidores
use DSCSAttrib = AttribGolding {}
...
public class DirectoryData
        extends DSCSData
        implements java.io.Serializable, UndoRedoOps
{
  private Vector  dirTable;

  public ObjectID lookup( String name) { ... }

  loggable public void insertEntry( String name, ObjectID objID) {
        /* Código normal para inserção dum par */ ... }
  undo ( Object undoInfo) {
        /* Código necessário para fazer undo da inserção */ ... }

  loggable public void removeEntry( ObjectID objID) { ... }
  undo ( Object undoInfo) { … }
}
```

Figura 4.5 – Esboço da implementação do CoObjecto directório.

O suporte linguístico deve transformar o código anterior em código Java *standard*, o qual pode ser compilado por qualquer ferramenta de desenvolvimento. Na figura 4.6 mostra-se um esboço do código que deve ser gerado. Na ausência de suporte linguístico é necessário gerar directamente este código.

```
public class DSCSCapsule
   extends LogDataCapsule
   implements java.io.Serializable
{
   public DirectoryCapsule( GlobalOID parent, GlobalOID current)
   {
          attrib = new AttribGolding( this.getClass().getName());
          attrib.link( this);

          logcore = new LogCoreImpl();
          logcore.link( this);

          logorder = new LogTotalSimpleUndoRedo();
          logorder.link( this, logcore);

          data = new DirectoryData( parent.objID, current.objID);
          data.link( this);
   }

   protected void readObject( DSCSAttrib localAttrib, ObjectHandle oH)
          throws DSCSIOException, ObjectTypeNotFoundException
   {
          super.readObject( localAttrib, oH);

          logorder = new LogTotalSimpleUndoRedo();
          logorder.link( this, logcore);
   }
}

public class DirectoryData
          extends DSCSData
          implements Serializable, UndoRedoOps
{
   private Vector   dirTable;

   public ObjectID lookup( String name) { ... }

   public void insertEntry( String name, ObjectID objID) {
          capsule.insertOp( new DDIE( name, objID)); // Coloca operação no log
   }

   // Classe que representa operação de inserção dum novo par no directório
   class DDIE
          implements java.io.Serializable, Operation
   {
          private Object[] a;
          private Object   u;

          public DDIE( String name, ObjectID oID) {
                 Object[]    args = {name, oID};
                 this.a = args;
          }
          public void applyOp( Object toObj, Object key, LoggedOperation lop) {
                 u = ((DirectoryData)toObj).insertEntry_Impl( (String)a[0], (ObjectID)a[1]);
          }
          public void undoOp( Object toObj, Object key, LoggedOperation lop) {
                 ((DirectoryData)toObj).insertEntry_Undo( (String)a[0], (ObjectID)a[1], u);
          }
```

```
        ...
    }
    Object insertEntry_Impl( String name, ObjectID objID) {
        /* código normal para inserção dum par */ ... }
    void insertEntry_Undo( String name, ObjectID objID, Object undoInfo) {
        /* código necessário para fazer undo da inserção */ ... }

    /* de modo semelhante para o removeEntry */
    ...
}
```

Figura 4.6 – Esboço do código gerado a partir da definição do CoObjecto.

O CoObjecto directório funciona da seguinte forma:

- Para criar um novo directório cria-se um CoObjecto de tipo *DirectoryCapsule*, o qual necessita para a sua criação do identificador do próprio directório e do directório pai. Esta operação cria imediatamente o conjunto de componentes associados a um directório.
- Este directório é gravado e lido de suporte estável por operações presentes na cápsula, as quais fazem recurso à propriedade *Serializable* do Java. O modo como esta gravação é feita pode ser redefinida.
- O código das aplicações acede ao CoObjecto directório através do objecto de dados (*DirectoryData*). Este redirige as operações que devem ser registadas para a cápsula (*DSCSCapsule.insertOp(Operation)*), a qual as envias para o *log* e invoca o ordenador de mensagens no sentido deste executar as operações ainda não executadas. O comportamento normal nos clientes é a execução imediata das operações executadas pelos mesmos, pelo que o método que implementa a operação é invocado no objecto de dados. O sistema obtém as alterações efectuadas aos CoObjectos nos clientes através da cápsula (*DSCSCapsule.checkChanges(void)*) – invocações essas redirigidas para o *log*. Estas operações são posteriormente enviadas para os servidores.
- As cópias nos servidores são instanciadas quando existem operações para serem armazenadas. É a cápsula que coordena o processo de armazenamento das operações (*DSCSCapsule.insertOp(LoggedOperation[])*), através do seu envio para o *log*, e sua execução, através da invocação do ordenador de mensagens. Este ordenador invoca os métodos das classes que implementam as operações de forma a garantir a ordenação desejada.

A sua manipulação pelos utilizadores é semelhante à efectuada em relação a um objecto normal, como se exemplifica na figura 4.7 – as funções do repositório utilizadas, apesar de ainda não apresentadas, são auto-explicativas.

```
        // Código de criação dum novo directório
GlobalOID   parentGOID = … // identificador do directório pai
GlobalOID   currentOID = … // identificador do próprio directório
DirectoryCapsule dirCapsule = new DirectoryCapsule( parentGOID, currentOID);
        // A partir deste momento o CoObjecto pode ser usado
        // gravar o novo CoObjecto
DSCSConfig.interfaceDSCS.putObject( "/NovoDirectório", dirCapsule);
        // Ler um CoObjecto
dirCapsule = (DirectoryCapsule)DSCSConfig.interfaceDSCS.getObject( "/NovoDirectório", 0);
        // Aplicar operações a um CoObjecto
ObjectID oid = … //identificador do CoObjecto
dirCapsule.data.insertEntry( "Novo CoObjecto", oid);
        // Gravar as alterações a um CoObjecto
DSCSConfig.interfaceDSCS.putObject( dirCapsule);
```

Figura 4.7 – Utilização do CoObjecto directório.

4.2.2 Alternativas de componentes

Nesta subsecção são apresentadas diversas alternativas possíveis para os diferentes componentes. Não se pretende ser exaustivo na apresentação das diversas possibilidades, mas antes mostrar a flexibilidade do modelo. Diferentes alternativas devem ser criadas como resultado da constatação de necessidades específicas na criação de novos CoObjectos.

4.2.2.1 Cápsula

O objecto cápsula coordena todo o funcionamento do CoObjecto implementado, determinando desde logo a constituição do mesmo. Um CoObjecto simples é normalmente composto por um componente de atributos, um de *log*, um de ordenação das mensagens, um de dados e uma cápsula. Esta apresenta um funcionamento simples, semelhante ao mostrado no CoObjecto directório, limitando-se praticamente a redirigir as invocações ao CoObjecto para os componentes respectivos. O modelo permite, no entanto, a criação de CoObjectos com constituições mais complexas para os quais existe a necessidade de criar novas cápsulas. Alguns exemplos destas constituições são as seguintes:

- CoObjecto com duas versões: a estável, resultado da aplicação das operações estáveis, i.e., aquelas que garantidamente não necessitam de ser desfeitas para que a ordem desejada seja alcançada; a tentativa, que resulta da aplicação de todas as operações conhecidas. A utilidade da existência duma versão tentativa tem sido repetidas vezes afirmada [TTP+95,JLT+95,GKL+94], pois permite observar imediatamente os resultados esperados duma determinada acção e manter a noção de que se está na presença dum resultado provisório.
- CoObjecto com diversas versões, as quais podem ser criadas aquando da existência de operações concorrentes conflituantes ou por instrução explícita dos utilizadores. A manipulação de versões pode ser efectuada igualmente por um sistema de nível superior, como os existentes para os sistemas de ficheiros [Tic85], ou alternativamente pelo objecto de dados.

4.2.2.2 Atributos

O objecto de atributos deve ser derivado sempre que necessário para criar atributos específicos para o CoObjecto que se está a criar. Os atributos podem ser acedidos separadamente do resto do CoObjecto, i.e., sem necessidade de obter uma cópia privada do CoObjecto. Assim, este acesso é mais rápido, pelo que por vezes pode ser útil utilizá-los para guardar informação sobre o estado do CoObjecto.

Adicionalmente, este objecto tem a responsabilidade de manter a meta-informação necessária no processo de sincronização, em particular, a informação relativa à libertação dos recursos ocupados pelas mensagens. Várias alternativas são possíveis como se discutirá nos capítulos 5 e 6.

4.2.2.3 *Log*

O *log* serve geralmente como um simples depósito de operações. Tem, ainda, a responsabilidade de adicionar às operações executadas pelos clientes informação que permita fazer a sua posterior ordenação (ver capítulo 5). Desta forma, podem criar-se *logs* específicos dependendo das informações necessárias e suficientes para uma determinada ordenação.

Uma característica que pode ser interessante em algumas situações é a da atribuição dum identificador único a cada operação, como acontece no algoritmo de *lazy replication* [LLS91]. Isto permite que a mesma operação seja enviada por um/vários clientes mais do que uma vez, sem que tal leve à sua repetida execução, o que pode ser útil, por exemplo, na integração com sessões síncronas (ver 4.3.1).

4.2.2.4 Ordenação das operações

Este componente coordena a ordenação e aplicação das operações. Como se referiu, este objecto tem, geralmente, a responsabilidade de garantir que as operações são aplicadas nas diferentes réplicas por uma ordem que leva à convergência das mesmas. Outras restrições podem estar envolvidas, como, por exemplo, a de tornar estável, o mais rapidamente possível, a aplicação duma operação, de modo a que se possam conhecer os efeitos (definitivos) que a mesma causa.

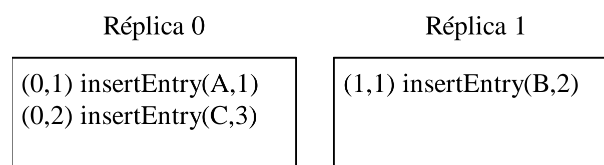

Figura 4.8 – Valor do *log* dum CoObjecto em duas réplicas diferentes. O par ordenado à esquerda é utilizado para ordenar as operações e tem a seguinte constituição (número da réplica, ordem na réplica). As operações ainda não foram propagadas entre as réplicas.

Este componente necessita de ter em atenção os dois problemas seguintes:

- **Obtenção da informação de ordenação**. Embora a informação adicionada pelos objectos de *log* seja suficiente para definir a ordenação desejada, o facto de a mesma ser efectuada distribuidamente pode levar à necessidade de comunicar com todas as réplicas para determinar a ordem relativa duma operação – esta situação acontece quando se usam protocolos de ordenação distribuída, para os quais se tem de determinar as condições de estabilidade que permitem a ordenação relativa das operações (basicamente garantir que em nenhuma réplica foram criadas operações anteriores na ordem definida). No exemplo apresentado na figura 4.8, e utilizando a ordenação total proposta em [Lam78][10], a inserção de *(C,3)* deve ser efectuada posteriormente à inserção do *(B,2)*. Assim, enquanto as operações não forem propagadas entre as duas réplicas não é possível aplicar as operações de inserção de *(B,2)* e de *(C,3)*. No entanto, mesmo na inexistência de *(B,2)*, era necessário contactar a réplica 1 para garantir essa inexistência e poder proceder à aplicação de *(C,3)*. No repositório proposto, estas informações são propagadas aquando das sessões de sincronização realizadas entre os servidores (ver capítulo 5).

  Na presença de réplicas que não estão sempre acessíveis (como, por exemplo, as réplicas situadas em computadores móveis), esta situação leva a que a determinação da ordem final duma operação, e suas efectivas consequências, não possa ser efectuada num período de tempo limitado. Quando tal não é razoável, o problema pode ser resolvido utilizando uma ordenação baseada num sequenciador (como usado no sistema Bayou). Neste caso, se o sequenciador estiver acessível, é possível determinar imediatamente a ordem duma qualquer operação e conhecer os resultados efectivos da sua aplicação (é necessário ter em atenção que os resultados duma operação podem estar dependentes do estado do CoObjecto, e portanto das operações que o possam alterar, não sendo por isso possível, no caso geral, conhecer o seu resultado antes de conhecer todas as operações que a antecedem na ordem estabelecida). No repositório proposto, uma solução baseada num sequenciador pode ser implementada usando o mesmo esquema de propagação epidémica, como se apresenta na subsecção 6.5.4.

- **Garantia da ordenação desejada.** A garantia da ordenação desejada na aplicação das operações pode ser efectuada por prevenção ou por resolução, i.e., ou se previne a existência de futuros problemas aplicando as operações apenas quando a ordem desejada é garantida, ou se aplicam as operações imediatamente e se resolvem os problemas que possam surgir desfazendo as operações executadas ou através de transformações dependentes da semântica das operações. No exemplo anterior e na réplica 0, corresponde a atrasar a inserção de *(C,3)*, até ao conhecimento de *(B,2)*, ou à inserção imediata, resolvida posteriormente através do desfazer da operação ou de manipulações semânticas. A decisão de qual a alternativa escolhida deve ter em atenção um conjunto de factores. Por um lado, nem sempre existem operações concorrentes, mas mesmo nesses casos pode ser

[10] $(r_1, o_1) \prec (r_2, o_2) \Leftrightarrow o_1 < o_2 \vee (o_1 = o_2 \wedge r_1 < r_2)$

necessário contactar as outras réplicas para garantir essa condição – o que leva a atrasar a aplicação das operações (as réplicas contactadas devem, neste caso, avançar o número de ordem a dar à próxima operação). Por outro lado, a resolução posterior dos conflitos só se torna possível numa das seguintes condições: quando é possível desfazer as operações executadas ([KB93]) – o que pode ser feito com auxílio da salvaguarda de parte do estado anterior do CoObjecto; ou quando se conhecem as transformações semânticas que levam a que o resultado da sua aplicação seja idêntico ao da ordenação desejada (casos particulares e facilmente tratáveis são aqueles em que: todas as operações são comutativas – neste caso a ordem de aplicação das operações é irrelevante; todas as operações se absorvem mutuamente – neste caso uma operação só é aplicada se for posterior àquelas que já foram aplicadas) [EG89].

Os diferentes modos de resolver os problemas atrás expostos levam à existência de várias soluções que providenciam uma mesma ordenação das operações. Obviamente, além dos problemas atrás expostos, existe o da ordem, propriamente dita, com que se pretende que as operações sejam aplicadas nas diferentes réplicas: sem ordem, ordem causal, ordem total, ordem causal total, …

Para cada CoObjecto é, geralmente, necessário seleccionar a ordem que garante que as diferentes réplicas evoluem para um estado comum, o que depende da semântica das operações definidas. Recorrendo a um exemplo já referido, se tivermos um CoObjecto que seja um valor numérico cuja única operação possível seja a adição de outro valor não existe a necessidade de nenhuma ordenação especial. No caso de existir a possibilidade de multiplicar o seu valor por outro, já temos necessidade de impor uma ordenação total à aplicação das operações.

A existência de informação semântica relativa às operações, e da possibilidade de determinar à priori as situações de conflito entre elas, abre várias hipóteses para o tratamento e aplicação das mesmas. Assim, por exemplo, o objecto de ordenação pode tomar em consideração o grafo de precedências reais das operações e criar diferentes versões do objecto de dados para os caminho que contivessem operações conflituantes entre si. Na figura 4.9, se as operações identificadas por (1,2) e (0,2) forem conflituantes podem criar-se as duas versões apropriadas.

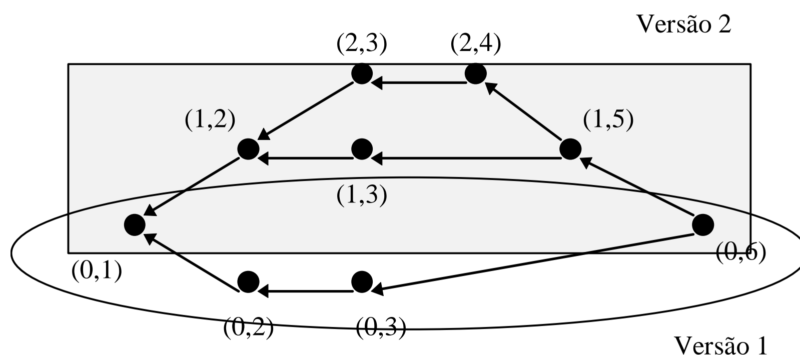

Figura 4.9 – Grafo de precedências real das operações (para a sua determinação é necessário que cada operação tenha associado um resumo das operações que já foram vistas, o que não é apresentado).

Outra possibilidade é a de reportar aos utilizadores a existência dos conflitos. Neste caso, poder-se-iam ainda tomar várias opções: não aplicar nenhuma das operações e esperar que os utilizadores resolvessem manualmente o problema; seleccionar de entre as operações em conflito uma para aplicação, ignorando as outras – que podiam ser enviadas aos utilizadores para as voltarem a executar caso desejem.

Como é fácil de perceber, este é o componente do modelo de objectos que pode apresentar mais variações, influenciando decisivamente o comportamento dos CoObjectos.

4.3 Considerações diversas

Nesta secção apresentam-se alguns problemas não abordados nesta dissertação, mas que devem ser considerados na criação dum sistema completo para suporte ao trabalho cooperativo assíncrono em larga-escala. Assim, começa-se por discutir a integração com sessões síncronas. Na subsecção 4.3.2 apresentam-se os problemas relacionados com a administração, controlo de acessos e segurança, e na subsecção 4.3.3 os mecanismos de coordenação, *awareness* e notificação.

4.3.1 Integração com sessões síncronas

Como já se referiu, um trabalho cooperativo pode englobar uma sequência de actividades síncronas e assíncronas. A sessão síncrona, devido ao seu elevado grau de interactividade, necessita que as acções produzidas por cada um dos participantes sejam do conhecimento dos outros no menor intervalo de tempo possível, pelo que necessitam de suporte adicional ao nível do sistema. Este suporte pode passar por soluções do tipo cliente/servidor [PDK96,SWP+97], de grupos de objectos parceiros (*peer*) [Sim97], ou outras.

Em qualquer um dos casos, as acções produzidas pelos diversos participantes repercutem-se em modificações executadas sobre o objecto manipulado. A representação persistente do mesmo (quando existe) pode ser um CoObjecto gerido pelo repositório apresentado, para o qual é transmitida a sequência de operações resultado da sessão síncrona. Estas operações são tratadas da forma habitual, apenas devendo ter em atenção a sua origem num conjunto de utilizadores.

Como a granularidade das operações semanticamente relevantes em sessões síncronas e assíncronas é normalmente diferente[11], pode existir a necessidade de fazer a conversão entre as operações executadas na sessão síncrona e aquelas que são enviadas para o repositório de CoObjectos. Esta conversão pode ser realizada nas aplicações que suportam a sessão síncrona de várias formas (por

---

[11] Por exemplo, na manipulação dum documento de texto, numa sessão síncrona a inserção dum carácter pode ser relevante (porque informa os outros participantes da existência dum participante a modificar uma dada zona do documento), enquanto numa sessão assíncrona apenas serão relevantes as modificações de um parágrafo/secção.

exemplo, pode-se resumir ao agrupamento de diversas operações síncronas sobre o objecto de dados em macro operações).

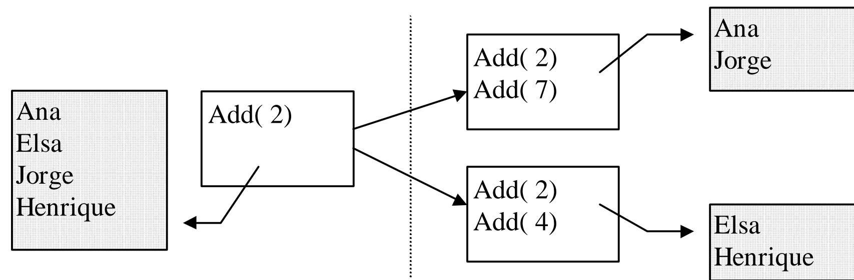

Figura 4.10 – Operações executadas durante uma sessão síncrona, e lista de participantes associados (a sombreado). O traço vertical tracejado representa a separação da sessão síncrona em duas sessões.

A integração com sessões síncrona suscita alguns problemas quando se permite que uma sessão síncrona dê origem a duas sessões devido a partições ocorridas no sistema de comunicações (ou outras falhas). Tomando o exemplo de figura 4.10, se ambas as partições gravassem todas as operações executadas, a operação *Add(2)* seria submetida duas vezes, o que poderia levar a que o resultado final não fosse o esperado no caso das aplicações enviadas em duplicado não sejam idempotentes. Para resolver este problema pode-se recorrer às seguintes soluções: utilizar um *log* que dê identificadores únicos a todas as operações e que trate adequadamente as operações duplicadas; utilizar um CoObjecto intermédio que unifique as sequências de operações geradas pelos diferentes subgrupos – estas sequências eram posteriormente aplicadas ao estado inicial do CoObjecto modificado (Em vez de enviar as modificações para o CoObjecto alterado, estas seriam enviadas para um CoObjecto que representava o resultado da sessão síncrona. Quando todos os subgrupos tivessem enviado a sua sequência gerava-se uma sequência unificada que seria então aplicada ao CoObjecto original).

Alternativamente pode evitar-se a criação de novas sessões, usando uma das seguintes estratégias: permitir que apenas um dos subgrupos continue a sessão síncrona, sendo este determinado pela presença duma maioria dos elementos, dum elemento com um papel especial (por exemplo, o coordenador), ou através de outra condição; abortar a sessão.

4.3.2 Administração, Controlo de Acessos e Segurança

A partilha de dados por utilizadores pertencentes a diferentes domínios administrativos levanta novos e renovados problemas administrativos e de segurança. Um dos problemas básicos levantado, é o da necessidade de criação duma identificação unificada para os utilizadores, que é exacerbado pela possibilidade do mesmo utilizador ter identificações diferentes nos diferentes domínios. Num sistema como o proposto, em que a partilha de CoObjectos representa a base do sistema, este problema ganha redobrada importância. Como se deseja permitir que a colaboração se possa alargar a qualquer

indivíduo, desde que os participantes assim o entendam, existe a necessidade de permitir a colaboração entre indivíduos pertencentes a domínios administrativos diferentes. Este problema básico está obviamente ligado com toda a política de segurança e controlo de acessos.

Para o sistema apresentado nesta dissertação propõe-se que cada volume defina um domínio administrativo, com a definição de utilizadores e controlo de acessos próprios. A informação sobre os utilizadores pode ser guardada num CoObjecto *bem-conhecido*, de modo semelhante ao que acontece nos sistemas UNIX com o ficheiro "*/etc/passwd*". Cada CoObjecto deve ter associado um esquema de controlo de acessos baseado nestes utilizadores que deve ter em atenção o facto do acesso aos CoObjectos ser realizado através da aplicação de operações.

Pode definir-se um espaço de soluções possíveis, em função do tratamento dado aos utilizadores e às operações possíveis. Quanto aos utilizadores, existe um espectro de hipóteses que vão desde o tratamento individual de cada utilizador até à formação de grupos de utilizadores com possíveis diferentes granularidade (possivelmente tomando em atenção os papéis exercidos pelos mesmos, por exemplo: coordenador, leitor, escritor, ...). Para o caso das operações estamos perante a mesma situação, podendo considerar cada operação individualmente ou criar grupos de operações (que no caso mais simples se podem reduzir a operações de leitura e de escrita). Deve ter-se em atenção que o esquema de acesso aos CoObjectos, baseado na criação duma cópia privada dos mesmos, é equivalente a permitir o acesso a todas as operações de leitura, pelo que o menor nível de controlo de acesso possível é o de permitir ou não o acesso, para *leitura*, dum CoObjecto.

O controlo de acessos será exercido em dois momentos: no momento em que um utilizador pede para criar uma cópia privada do CoObjecto, verifica-se se o mesmo tem permissão para tal; quando as operações são aplicadas aos CoObjectos nos servidores, verifica-se se os utilizadores têm permissão para as executarem. Para tal é necessário que se guarde com cada operação a informação do utilizador (ou grupo de utilizadores no caso do resultado duma sessão síncrona) que a executou.

Para que um esquema de controlo de acessos tenha sucesso é necessário que a autenticação dos clientes seja efectuada de forma segura. Existem vários métodos de fazer essa identificação, como, por exemplo, a assinatura digital das operações executadas. Quando a informação guardada pelo sistema é sensível e a rede de comunicações em que a mesma é propagada não é segura, é necessário utilizar comunicações cifradas.

As ideias apresentadas nesta secção necessitam de ser mais elaboradas, com recurso a um estudo profundo das soluções apresentadas noutros sistemas [RPP+,SNS88,CGR90], de novas soluções, e da adequação das mesmas às características específicas do sistema proposto.

4.3.3 Mecanismos de coordenação, *awareness* e notificação

Um sistema de suporte ao trabalho cooperativo deve possuir mecanismos de:

- Coordenação, por forma a que as contribuições produzidas pelos participantes sejam positivas no sentido de alcançar o objectivo comum que o grupo pretende alcançar. O planeamento das contribuições de cada elemento pode ser executado com o auxílio de ferramentas de gestão de projectos, dividindo o trabalho em subtarefas atribuídas a diferentes participantes. Existem diversas ferramentas que auxiliam esse mesmo processo, as quais têm uma utilização alargada pelo menos na gestão dos projectos informáticos. Um sistema de suporte ao trabalho cooperativo como o apresentado deve limitar-se a permitir a imposição das decisões tomadas no planeamento, e não definir esse planeamento. Deve, ainda, permitir a sua fácil alteração face a reorganizações impostas pela evolução efectiva dos trabalhos. Pensa-se que esta imposição pode ser realizada através do mecanismo de controlo de acessos. Muitas vezes, no interior das organizações, o cumprimento das decisões de coordenação é apenas imposto por normas de conduta social e disciplina dos elementos.

- *Awareness*, de forma a permitir que os diferentes participantes tenham noção do trabalho que os outros realizam. Esta informação é fundamental para permitir avaliar a evolução do trabalho produzido, e desta forma coordenar da melhor forma possível os diferentes participantes. Existem duas formas de obter a informação posteriormente distribuída que parecem apropriadas para um sistema como o proposto: informacional explícita, no qual cada utilizador explicitamente indica a macro-actividade que vai realizar; informacional implícita, no qual o sistema mediante as operações executadas gera automaticamente a história dos CoObjectos. As informações recolhidas anteriormente podem ficar armazenadas nos CoObjectos para consulta, quando desejado, ou serem automaticamente distribuídas pelos elementos nelas interessadas através dum mecanismo de notificação. No sistema proposto, a informação gerada pode ser armazenada num novo componente, sendo obtida implicitamente através da associação com cada operação duma mensagem legível por humanos.

- Notificação, para que o sistema possa informar os participantes dos eventos ocorridos (como por exemplo, a existência de operações conflituantes) e estes possam trocar informação entre si. Um método simples e eficaz de implementar um sistema de notificação é associar a cada cliente um endereço de correio electrónico. Adicionalmente, podem fornecer-se a cada cliente ferramentas de tratamento automático deste correio.

Estes aspectos não foram tratados no decorrer desta dissertação, podendo ser obtida mais informação geral sobre os mesmos em [DB92,AMM+96]. Estes assuntos são tratados de modo geral e especificamente no âmbito do sistema proposto em [DMP+97].

Capítulo 5

Comunicação epidémica

Muitos sistemas distribuídos de larga-escala podem ser construídos eficazmente a partir dum subsistema de comunicação diferida, i.e., no qual não existe a necessidade de propagação imediata de eventos e dados. O sistema de *news* da *Internet* e os sistemas que usam o serviço de correio electrónico para as suas comunicações são exemplos marcantes.

Quando um elemento do sistema pretende enviar uma mensagem a outro, passa essa responsabilidade para o subsistema de comunicação diferida. Este subsistema encarrega-se do envio da mensagem, permitindo ao cliente abstrair-se da complexidade que pode advir das múltiplas falhas possíveis, quer do sistema de comunicações, quer dos constituintes do sistema. Em contrapartida, geralmente, não oferece garantias quanto ao momento em que as mensagens são enviadas. Estes subsistemas permitem, ainda, realizar uma gestão mais racional dos recursos comunicacionais existentes, com o agrupamento das mensagens, e o seu conveniente escalonamento (no tempo). A computação móvel vem exacerbar a utilidade destes sistemas, devido às menores capacidades de comunicação existentes, as quais são, durante alguns períodos, mesmo inexistentes.

Como exemplos de sistemas de comunicação diferida, no paradigma cliente/servidor, temos os RPC's diferidos, como utilizado pela Oracle [Ora95] ou pelo sistema Rover [JLT+95]. Em termos de comunicação em grupo, temos os sistemas de comunicação epidémica [DGH+87].

No sistema proposto nesta dissertação usa-se um esquema de comunicação diferida (tipo RPC's diferidos) no envio de comunicações entre os clientes e os servidores, e um esquema de comunicação epidémica como forma de propagar para todos os servidores as operações aplicadas aos CoObjectos.

Neste capítulo apresenta-se o modelo de comunicação epidémica usado na propagação das modificações.

5.1 Arquitectura Básica

Um sistema de comunicação epidémica é constituído por um grupo de *principais* (no caso presente, os servidores do repositório) que comunicam entre si através da troca de mensagens. Considera-se que estes *principais*, apesar de poderem ter falhas temporárias, não falham definitivamente, i.e., não cessam a sua actividade sem anunciar antecipadamente a futura ocorrência dessa situação. Se considerarmos que os *principais* constituem os nós dum grafo, e os arcos a possibilidade de comunicação (não necessariamente permanente), é suficiente que o grafo seja fortemente conexo[12] para que se possa construir um sistema de comunicação epidémica. Na figura 5.1 apresenta-se um exemplo de topologia possível, englobando a presença de troços de comunicação não permanente.

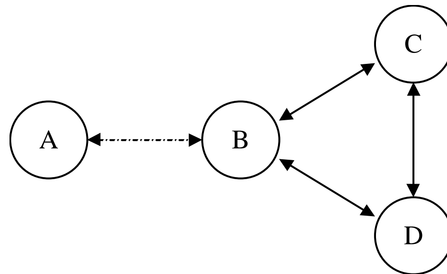

Figura 5.1 – Topologia dum conjunto de *principais*. Os *principais* B, C, D apresentam capacidade permanente de comunicação entre si, enquanto o *principal* A apenas comunica episodicamente com o *principal* B, podendo representar um computador móvel.

Os membros dum sistema de comunicação epidémica têm dois níveis fundamentais:

- **Sistema de comunicações**, utilizado pelos *principais* para trocarem mensagens, com possível recurso a diferentes alternativas, como por exemplo o TCP/IP, SMTP, NNTP, CORBA ORB's [OMG91], Java RMI [Sun96], ...
- **Núcleo do sistema**, composto pelos seguintes módulos: um módulo de armazenamento das mensagens (*log*); um módulo de ordenação das mensagens; um módulo de filiação, encarregue de gerir a filiação do grupo de comunicações; um módulo de política de comunicações, responsável por definir e iniciar a comunicação com os outros *principais*.

Adicionalmente, são constituídos por uma aplicação que comunica com o núcleo de forma a difundir novas mensagens e a receber, ordenadamente, as difundidas pelos diferentes membros do sistema. As novas mensagens são introduzidas no *log* para posterior propagação. Num sistema epidémico não se considera a possibilidade de elementos exteriores ao grupo enviarem uma mensagem para difusão (como acontece em alguns sistemas de comunicação em grupo síncrono).

---

[12] Um grafo diz-se fortemente conexo, quando para qualquer nó existe um caminho com origem em todos os outros nós e fim no nó considerado.

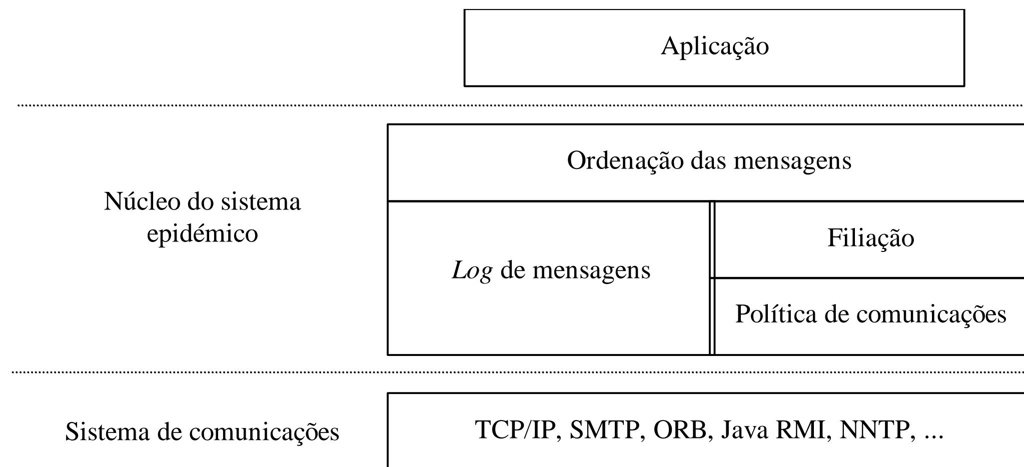

Figura 5.2 - Arquitectura base dum sistema de comunicação epidémica.

Os *principais* propagam as mensagens entre si durante sessões de sincronização efectuadas entre pares de *principais*. Nestas, os *principais* trocam as mensagens necessárias a que os seus *logs* fiquem idênticos (independentemente da origem das mensagens, i.e., um *principal* não se limita a enviar as mensagem que gerou). Desta forma, e se como se referiu a topologia das sessões de sincronização formarem um grafo fortemente conexo, todos os *principais* ficam a conhecer todas as mensagens.

As mensagens presentes no *log* são entregues à aplicação para processamento, mediante a ordenação imposta pelo respectivo módulo, logo que a mesma seja determinada (i.e., as mensagens só são entregues quando o módulo de ordenação consegue garantir a ordenação desejada). Assim, podem existir mensagens no *log* que ainda não tenham sido entregues à aplicação associada. Todas as mensagens são entregues a todas as aplicações do grupo de *principais* através do mecanismo esboçado anteriormente – incluindo àquelas aplicações que as originaram (*self-delivery*).

De seguida descrevem-se sumariamente as funções dos diversos módulos componentes dum sistema epidémico.

5.1.1 Sistema de comunicações

O sistema de comunicações é responsável pela entrega das mensagens enviadas dum *principal* para outro. Num sistema epidémico este não necessita de ser síncrono nem bidireccional, embora o sistema seja mais eficiente nessas condições.

5.1.2 *Log* de mensagens

Neste módulo são guardadas as mensagens que foram enviadas para o grupo. Estas mensagens podem ter origem no próprio *principal*, ou podem ter sido recebidas doutros *principais* nas mensagens

de sincronização que estes trocam. As mensagens são estampilhadas de modo a poderem ser mais tarde ordenadas. Diversas estampilhas são possíveis, sendo as seguintes as mais utilizadas:

- Identificador do *principal* no qual foi gerada + número de ordem no *principal*.

  Esta identificação é suficiente para garantir uma ordenação total das mensagens, e, associada com a manipulação do número de ordem, permite garantir uma ordem causal na ordenação das mesmas [Lam78]. No entanto, permite apenas garantir que uma determinada ordem é causal, e não determinar se duas mensagens estão ou não causalmente relacionadas.

- Relógios vectoriais[13] [PPR+83] + identificador do *principal* no qual foi gerada.

  Esta identificação permite traçar o grafo de relações causais estabelecidas entre as diferentes mensagens (e obviamente definir uma ordenação total).

Este módulo necessita, igualmente, de lidar com o problema da libertação de recursos ocupados pelas mensagens. Para tal, precisa de determinar o momento em que uma dada mensagem deixa de ser necessária (por já ter sido processada e por todos os *principais* com o qual comunica já a conhecerem).

5.1.3 Filiação

O módulo de filiação encarrega-se de fornecer uma visão de quais são os *principais* pertencentes ao grupo. Ao contrário do que acontece nos sistemas de comunicação por grupo síncronos, a filiação nos sistemas epidémicos varia esporadicamente e a pedido, i.e., um *principal* só sai ou entra no grupo quando tal solicita. Devido a este facto as funções deste módulo limitam-se apenas a fazer um registo de quem são os membros do grupo, processando eventuais mensagens de alterações.

Quando existe uma variação da filiação pode tornar-se necessário ajustar as estampilhas associadas às mensagens.

5.1.4 Política de comunicações

O módulo da política de comunicações é responsável por determinar a topologia e escalonamento das sessões de sincronização, obtendo um compromisso entre dois objectivos conflituantes: minimizar os recursos comunicacionais usados; propagar as mensagens o mais rápido possível. As políticas definidas podem ter carácter: totalmente estático, definido pelo administrador do sistema; totalmente dinâmico, variando com as condições de funcionamento do sistema; ou situarem-se algures no meio do espectro de que as duas hipóteses anteriores são os extremos. Uma possibilidade topológica, muitas

---

[13] Muitas vezes, aos relógios vectoriais é dada a designação de vectores versão. Tal fica a dever-se ao facto de eles, identificando as mensagens processadas e cujos efeitos são reflectidos nuns determinados dados, identificarem também a versão desses dados (que resulta do processamento dessas mensagens). Nesta dissertação, os dois termos são usados indiferentemente, conforme se achou mais apropriado.

vezes explorada nos sistemas de larga escala, é a organização hierárquica do sistema (como acontece, por exemplo, no serviço de DNS [Moc87] e se sugere em [PM96] e [PGZ92]).

Este módulo deve ter em consideração o problema da variação de filiação. Num sistema epidémico, esta é efectuada através da propagação de mensagens que são processadas por este módulo. Assim, diferentes *principais* podem ter uma diferente visão de quais os *principais* que pertencem ao grupo, por terem processado diferentes mensagens. Como a política de comunicações pode ser função dessa visão, deve garantir-se que estas diferenças não levam à partição dos *principais* em grupos não inter-comunicantes. A determinação de políticas de comunicação é um assunto complexo que não é, no entanto, discutida em detalhe nesta dissertação.

5.1.5 Ordenação das mensagens

O módulo de ordenação das mensagens é responsável por garantir a entrega das mensagens presentes no *log* por uma ordem apropriada. Esta ordenação pretende garantir que todos os *principais* evoluem de forma coerente, o que é normalmente um dos objectivos destes sistemas. O tipo de ordenação necessária para o garantir está dependente da semântica do processamento das mensagens. Por vezes, é necessário garantir que todas as mensagens são processadas pela mesma ordem em todos os *principais*.

Geralmente, quanto mais restritiva for a ordem a ser imposta, mais tarde a mesma pode ser garantida, o que atrasa o processamento das mensagens.

Este módulo é, igualmente, responsável por entregar as mensagens de filiação ao respectivo módulo.

5.2 Protocolos de sincronização

Nesta secção descrevem-se os protocolos que são estabelecidos durante a troca de mensagens entre os *principais*. Para o funcionamento dos protocolos descritos existe a necessidade de todas mensagens serem identificadas pelo par constituído pelo identificador do *principal* onde foram originados e pelo número de ordem de submissão. Este número de ordem deve ser crescente e pode ser incrementado indiscriminadamente, i.e., para duas mensagens originadas consecutivamente, a diferença dos números de ordem deve ser inteira e positiva, mas não necessariamente igual à unidade.

As alterações à filiação são tratadas posteriormente, assumindo-se nesta secção, que a filiação do grupo de *principais* permanece constante. Apresentam-se inicialmente os protocolos de sincronização duma forma genérica, acompanhados duma explicação sumária do seu funcionamento. De seguida apresentam-se instanciações dos mesmos para o caso do algoritmo epidémico simples e para um algoritmo em que se troca informação conducente à libertação dos recursos ocupados pelas mensagens.

5.2.1 Algoritmo genérico

Nesta secção apresentam-se duas versões do algoritmo genérico. A primeira exibe como requisito a utilização de uma comunicação síncrona bidireccional (por exemplo, TCP/IP), enquanto a segunda é baseada na utilização duma comunicação assíncrona (por exemplo, correio electrónico). As principais características e diferenças entre os dois tipos de comunicação, tendo em conta as peculiaridades dum sistema de comunicação epidémica, são as seguintes.

- Ambos os modos permitem escalonar, para os momentos mais apropriados, o início dum protocolo. O modo assíncrono permite, ainda, escalonar o tratamento e resposta ao passos do mesmo, i.e., o *principal* que responde a uma mensagem (passo) duma sessão de sincronização pode fazê-lo quando for apropriado para si.
- A possibilidade de recorrer ao correio electrónico para implementar uma comunicação assíncrona torna-a quase universalmente utilizável. Adicionalmente permite contornar as restrições de segurança impostas em certas organizações, (através da utilização de *firewalls*), sem necessidade de intervenção do respectivo administrador do sistema. Possibilita ainda, que máquinas que não estejam conectadas simultaneamente possam comunicar, como por exemplo, dois computadores móveis, ou dois computadores conectadas através dum fornecedor de serviços de *Internet* – com conectividade efectiva durante curtos (e possivelmente dispares) períodos de tempo.
- O período de tempo que, geralmente, medeia entre o envio e a recepção duma comunicação assíncrona é muito superior ao duma comunicação síncrona. Assim o algoritmo usando comunicação síncrona é potencialmente mais eficiente – não só por ser mais rápido, mas também porque ao diminuir a possibilidade de existirem sessões de sincronização simultâneas diminui a probabilidade da recepção duma mensagem mais do que uma vez.

5.2.1.1 Comunicação síncrona bidireccional

Send(OutVectors)	Receive(InVectors)
Receive(InVectors)	Send(OutVectors)
Send(MyMsgs)	Receive(PeerMsgs)
Receive(PeerMsgs)	Send(MyMsgs)
InsertInLog(PeerMsgs)	InsertInLog(PeerMsgs)
UpdateVectors()	UpdateVectors()
UpdateClock()	UpdateClock()
UpdateAckVectors()	UpdateAckVectors()

Figura 5.3 – Protocolo genérico utilizado nas sessões de sincronização. Do lado esquerdo apresenta-se o código executado pelo iniciador da comunicação, e do lado direito o executado pelo receptor.

Nas sessões de sincronização os *principais* começam por trocar informação associada ao processo de sincronização (incluindo informação sobre o estado presente). Após este passo, cada *principal*

envia para o outro o conjunto de mensagens presentes no seu *log* e que pensa que o parceiro não conhece. Normalmente a informação inicial contém um resumo das mensagens já observadas, pelo que estaremos na presença dum protocolo do tipo *I have/Send me*. As novas mensagens recebidas são inseridas no *log* de cada um dos *principais*, que de seguida procedem à actualização da informação associada ao processo de sincronização. Uma implementação ingénua e ineficiente pode reduzir-se à troca de todas as mensagens presentes no *log* de cada *principal*.

5.2.1.2 Comunicação Assíncrona

O algoritmo assíncrono é semelhante ao síncrono, limitando-se a dividi-lo de forma a maximizar o proveito proveniente das comunicações efectuadas. Pode ser executado, igualmente, de forma tão eficiente quanto o síncrono em condições de comunicação síncrona bidireccional.

```
Send( OutVectors)
```

Figura 5.4 – Passo inicial numa comunicação assíncrona.

```
nStep = 1
Send( OutVectors, MyRecentMsgs, nStep)
```

Figura 5.5 – Passo inicial alternativo numa comunicação assíncrona.

O passo inicial do protocolo corresponde ao envio da informação associada ao processo de sincronização. No caso em que se espera que a resposta a cada comunicação seja muito demorada (na ordem das horas), pode-se enviar adicional e imediatamente o conjunto de mensagens que se supõe que o parceiro não conhece. Neste caso, sacrifica-se, potencialmente, a eficiência da utilização dos recursos comunicacionais (enviando mensagens que o parceiro pode já ter conhecimento) em favor de uma maior rapidez do processo de sincronização.

```
Receive( InVectors)
UpdateClock()
UpdateAckVectors()
nStep = 1
Send( OutVectors, MyMsgs, nStep)
```

Figura 5.6 - Resposta ao passo inicial normal.

Em resposta ao passo inicial normal, o *principal* actualiza a sua informação sobre o processo de sincronização, enviando, de seguida, além da sua própria informação, as mensagens que supõe que o parceiro não conhece – baseando-se nas informações recebidas.

```
Receive( InVectors, PeerMsgs, nStep)
InsertInLog( PeerMsgs)
UpdateVectors()
UpdateClock()
UpdateAckVectors()
IF NOT LastStep() THEN Send( OutVectors, MyMsgs, nStep + 1)
```

Figura 5.7 – Resposta a um passo intermédio, ou ao passo inicial alternativo.

Em resposta a um passo intermédio, ou ao passo inicial alternativo, o *principal* introduz as novas operações no seu *log* e actualiza a informação do processo de sincronização. No caso de considerar que não se trata dum passo final, envia a sua própria informação do processo e as mensagens não conhecidas pelo parceiro.

A determinação da condição de passo final (*LastStep()*) está normalmente associada ao número de novas mensagens recebidas e a enviar (caso não existam novas mensagens, os dois *principais* estão sincronizados, pelo que o processo pode parar) e/ou ao número de interacções já efectuadas (quando *nStep* é igual a 2, ambos os *principais* enviaram as suas novas mensagens – só existem outras no caso de terem sido recebidas desde o envio das anteriores). A definição concreta dessa condição é independente do algoritmo que se usa, razão pela qual a mesma será omitida nas próximas secções.

5.2.2 Algoritmo epidémico simples

No algoritmo epidémico simples, cada *principal* mantém um resumo das mensagens recebidas. Este resumo é trocado durante as sessões de sincronização de forma a que cada mensagem originada noutro *principal* seja recebida uma única vez. Neste primeiro algoritmo não se desenvolve qualquer esforço no sentido de libertar os recursos ocupados pelas mensagens no *log*, o qual crescerá indefinidamente.

Cada *principal*, *i*, mantém um vector versão, V^i, representando o resumo das mensagens já recebidas, o qual é actualizado aquando das sessões de sincronização. A entrada *k* desse vector, V_k^i, contém o número de ordem das mensagens originadas no *principal k* para o qual já foram observadas pelo *principal i* todas as mensagens com número de ordem menor ou igual. A entrada *i* deste vector, V_i^i, contém o número de ordem da última mensagem originada no próprio *principal*. A próxima mensagem gerada deve ter um número de ordem maior ou igual a $V_i^i + 1$.

Por outro lado e tal como já foi referido anteriormente, cada mensagem, ${}^pM_o$, é etiquetada com o número do *principal* onde foi gerada, *p*, e o número de ordem nesse *principal*, *o*.

Este algoritmo (assim como os seguintes) permite que as mensagens possam ser adicionalmente propagadas entre os *principais* por outras formas de comunicação – por exemplo, usando *multicasting*. Estas mensagens são, no entanto, igualmente propagadas pelo mecanismo normal, pelo que o método alternativo não necessita de garantir a entrega das mensagens a todas as réplicas. As mensagens recebidas pela forma alternativa são simplesmente colocadas no *log* sem actualização da informação

associada ao processo de sincronização (resumo das mensagens, neste algoritmo) – passam, portanto, a existir mensagens presentes no *log* que não estão reflectidas nos resumos. O recurso a estes métodos alternativos pode ser útil na difusão de mensagens, que pela sua importância, devam ser conhecidas imediatamente por todas as réplicas.

Os algoritmos base apresentados anteriormente são instanciados com os valores seguintes, assumindo-se uma comunicação entre dois *principais* *i* e *j*, e apresentando-se o pseudo-código executado em *i*. No apêndice A.1 apresenta-se o código dos protocolos devidamente instanciados.

OutVectors := V^i
InVectors := V^j
MyMsgs := $\{ {}^pM_o \in log : o > V_p^j \}$
MyRecentMsgs := log
UpdateClock() := ε // nenhuma operação
UpdateAckVectors() := ε
UpdateVectors() :=
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists {}^xM_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$

Nesta primeira versão, a informação associada ao processo de sincronização trocada pelos *principais* reduz-se ao resumo das mensagens já recebidas.

Propriedade 1. (da propagação total das mensagens entre dois *principais*): Assumindo que os vectores versão que resumem as mensagens observadas são actualizados correctamente, é fácil verificar que cada *principal* envia para o outro todas as mensagens que conhece e que o outro ainda não tinha observado, independentemente do *principal* em que foram originadas (*MyMsgs*).

Correcção da actualização dos resumos de mensagens:

Propriedade 1.1 – Todas as mensagens que um *principal* anuncia conhecer são enviadas (caso não sejam já conhecidas): Esta condição é garantida pelo facto de os vectores serem enviados antes ou simultaneamente com as mensagens. Desta forma, todas as mensagens que estejam reflectidas nos vectores, estarão necessariamente presentes nos *logs* e poderão ser enviadas caso o outro *principal* as não conheça. Podem, no entanto, existir mensagens que sejam enviadas e não sejam reflectidas pelo vector, pois entre o momento em que os vectores e as mensagens são enviadas, um *principal* pode ter conhecimento de novas mensagens, quer originadas no próprio *principal*, quer recebidas através doutra sessão de sincronização. Esta situação apenas pode suceder no caso síncrono, pois no assíncrono o envio das mensagens e dos vectores é processada de forma atómica.

Propriedade 1.2 – No final da sessão não existem mensagens reflectidas no resumo que não tenham sido observadas: Como por hipótese todas as mensagens reflectidas no resumo foram observadas, o resumo do próprio *principal* não reflecte mensagens desconhecidas. Pela condição anterior, todas as mensagens reflectidas no resumo do *principal* com o qual decorreu a sessão de sincronização foram enviadas. Como o vector versão, V^i, é actualizado determinando o máximo entre o vector do próprio *principal* e o vector recebido do outro *principal*, não existirão mensagens não observadas reflectidas pelo vector final. Devido ao facto da actualização só se processar depois das mensagens serem introduzidas no *log*, o vector nunca reflectirá mensagens não presentes no *log*, mesmo em caso de falha durante a execução do protocolo.

Propriedade 1.3 – Evolução dos resumos: No caso em que uma sessão de sincronização ocorre sem falhas, após a mesma, o resumo das mensagens reflectirá todas as mensagens que o parceiro reflectia no resumo no início da mesma. As mensagens que tenha tido conhecimento durante a sessão de sincronização, serão reflectidas numa posterior sessão. Admitindo como razoável que o tempo médio entre falhas associadas aos dois *principais* e ao sistema de comunicações que os interliga é (muito) maior do que o tempo necessário à execução do algoritmo de sincronização, e portanto o protocolo estabelecido entre os dois *principais* não falhará sempre, os resumos evoluem reflectindo, mais cedo ou mais tarde, as novas mensagens presente no sistema.

Pelas propriedades anteriormente apresentadas, facilmente se conclui que os resumos das mensagens evoluem e não reflectem nenhuma mensagem que não tenha sido observada, **pelo que a sua evolução é correcta**. As mensagens, que são transmitidas por meios alternativos – *multicasting*, … – e que por isso não se encontram reflectidas nos resumos, podem ser trocadas mais de uma vez em diferentes sessões de sincronização. Este facto, contudo, não afecta a correcção do algoritmo. O ciclo *WHILE* do *UpdateVectors* propõe-se acelerar a inserção destas mensagens nos resumos, verificando se existe alguma destas mensagens, para as quais garantidamente não existe nenhuma que a preceda que não seja conhecida.

Propriedade 2. (da propagação global das mensagens): Como em cada sessão de sincronização as mensagens são propagadas independentemente do *principal* em que tiveram origem, a disseminação duma mensagem está dependente apenas da topologia das sessões. Para que uma mensagem originada num dado *principal*, *i*, seja propagada para outro, *j*, é necessário que se realizem sucessivamente as sessões de sincronização (envolvendo os *principais* $p_1,...,p_{n-1}$), $s_1 : i \rightarrow p_1$, $s_2 : p_1 \rightarrow p_2$, ..., $s_n : p_{n-1} \rightarrow j$. Facilmente se conclui que a condição necessária e

suficiente para que todas as mensagens sejam propagadas para todos os *principais* é a de que as sessões de sincronização formem um grafo fortemente conexo.

As observações anteriores são válidas para todas as versões do protocolos apresentadas, pois baseiam-se apenas no facto de, nas sessões de sincronização, os resumos não serem enviados posteriormente às mensagens, e serem actualizados apenas depois de as mensagens recebidas terem sido introduzidas no *log*, o que acontece em todas os protocolos.

5.2.3 Algoritmo epidémico com libertação de recursos

No algoritmo apresentado anteriormente, assumia-se que uma mensagem, após ter sido introduzida no *log*, nunca seria de lá retirada, o que obviamente não é razoável. Para que o algoritmo epidémico funcione, uma mensagem pode ser retirada do *log* quando tiver sido processada pelo *principal* e não existir a necessidade de ser enviada a nenhum outro *principal* durante as sessões de sincronização. Esta condição verifica-se quando os *principais* com os quais executa sessões de sincronização já têm conhecimento da mensagem. Para que um *principal* possa libertar as mensagens, é portanto necessário que tenha acesso a um resumo das mensagens já observadas pelos *principais* com os quais comunica. No caso geral, em que um *principal* pode comunicar com todos os outros em sessões de sincronização, é necessário manter/obter informação sobre as mensagens observadas por todos os *principais*.

Uma solução para o problema seria manter em cada *principal* uma matriz V, tal que V_{ik}, contivesse o número de ordem das mensagens originadas no *principal k*, para o qual o *principal* em que a matriz se encontra sabe que já foram observadas pelo *principal i* todas as mensagens com número de ordem menor ou igual. O vector V_i dessa matriz, corresponderia para o *principal i*, ao vector mantido na versão anterior do algoritmo (V^i). Esta matriz seria trocada aquando das sessões de sincronização, sendo actualizada de forma idêntica aos vectores no algoritmo simples. O problema desta aproximação prende-se com a necessidade de armazenar e propagar a matriz com grandeza $O(n^2)$, o que poderá ser incomportável quando existe um elevado número de *principais*.

Nesta dissertação apresentamos, alternativamente, um algoritmo baseado no proposto em [Gol92b] (onde se apresentam as demonstrações formais da correcção do mesmo). Nele, a propagação das mensagens faz-se de forma idêntica à do algoritmo simples, mantendo-se adicionalmente, em cada *principal i*, o vector de confirmação A^i. Este vector resume a matriz anterior, reduzindo o espaço ocupado a $O(n)$. A entrada *k* desse vector, A_k^i, contém o número de ordem para o qual se sabe que o *principal k* já observou todas as mensagens originadas em todos os *principais* com número de ordem menor ou igual.

No caso do valor de A_k^i ser por exemplo 5, o *principal i* sabe que o *principal k* já viu todas as mensagens provenientes de qualquer outro *principal* com número de ordem menor ou igual a 5. O menor dos valores correspondentes aos *principais* com os quais pode comunicar indica o número de

ordem máximo das mensagens que podem ser apagadas do *log* (caso já tenham sido aplicadas, obviamente).

Para que se possam apagar as mensagens com a maior celeridade possível, é necessário que os números de ordem evoluam de forma sincronizada. Como se pode observar pela tabela 5.1, no caso extremo de o número de ordem dum *principal* se manter eternamente constante, os valores nos vectores de confirmação (A^i) nunca ultrapassariam essa constante, pois eles serão iguais ao menor número de ordem de todos os *principais*.

V^i, A^i	*Principal* 0	*Principal* 1	*Principal* 2
	(3, 0, 0), (0, 0, 0)	(0, 5, 0), (0, 0, 0)	(0, 0, 0), (0, 0, 0)
0 <-> 1	(3, 5, 0), (0, 0, 0)	(3, 5, 0), (0, 0, 0)	
	(7, 5, 0), (0, 0, 0)	(3, 8, 0), (0, 0, 0)	(0, 0, 0), (0, 0, 0)
1 <-> 2		(3, 8, 0), (0, 0, 0)	(0, 0, 0), (0, 0, 0)
	(9, 5, 0), (0, 0, 0)	(3, 10, 0), (0, 0, 0)	(0, 0, 0), (0, 0, 0)
0 <-> 1	(9, 10, 0), (0, 0, 0)	(9, 10, 0), (0, 0, 0)	
(caso o *principal* 2 também evolua)			
	(13, 10, 0), (0, 0, 0)	(9, 12, 0), (0, 0, 0)	(0, 0, 5), (0, 0, 0)
1 <-> 2		(9, 12, 5), (0, 5, 5) *	(9, 12, 5), (0, 5, 5)
	(14, 10, 0), (0, 0, 0)	(9, 15, 5), (0, 5, 5)	...
0 <-> 1	(14, 15, 5), (5, 5, 5)	(14, 15, 5), (5, 5, 5)	
	Podem apagar-se as mensagens com número de ordem inferior ou igual a 5		

Tabela 5.1 – Evolução possível dos vectores usados no algoritmo epidémico sem acerto dos números de ordem. Assume-se que as sincronizações ocorrem sem problemas de forma a poder actualizar mais rapidamente os vectores – em * o valor correcto seria (0, 5, 0) pois não pode ter a certeza que o *principal* 2 registou as mensagens que lhe foram enviadas. A evolução entre sincronização corresponde a um, entre inúmeras, evoluções possível.

Este problema pode ser facilmente solucionado se o número de ordem for alterado aquando das sessões de sincronização, mesmo na ausência de novas mensagens, de modo semelhante ao proposto por Lamport em [Lam78]. Esta alteração não influencia o correcto funcionamento do algoritmo epidémico, pois, como se referiu, não existe a necessidade dos números de ordem atribuídos às mensagens terem valores consecutivos. Além disso, o valor do resumo V_k^i indica que o *principal* i, conhece todas as mensagens originadas em k, com número de ordem igual ou inferior a esse valor, não obrigando, no entanto, que exista uma mensagem com esse número de ordem. No apêndice A.2 apresenta-se o código dos protocolos devidamente instanciados.

OutVectors := V^i, A^i

InVectors := V^j, A^j

MyMsgs := $\{ {}^{p}M_o \in log : o > V_p^j \}$

MyRecentMsgs := $\left\{ {}^{p}M_{o} \in log : o > A_{j}^{i} \right\}$

UpdateClock() := $V_{i}^{i} = \max\left(\left\{ V_{k}^{i} : \forall k \right\} \cup \left\{ V_{k}^{j} : \forall k \right\} \right)$

UpdateAckVectors() ::=

$A_{i}^{i} = \min\left(\left\{ V_{k}^{i} : \forall k \right\} \right)$

$A_{j}^{i} = \min\left(\left\{ V_{k}^{j} : \forall k \right\} \right)$

$A_{x}^{i} = \max\left(A_{x}^{i}, A_{x}^{j} \right), \forall x$

UpdateVectors() ::=

$V_{x}^{i} = \max(V_{x}^{i}, V_{x}^{j}), \forall x$

WHILE $\exists^{x} M_{V_{x}^{i}+1} \in log$ DO $V_{x}^{i} = V_{x}^{i} + 1$

Neste algoritmo, a propagação das mensagens realiza-se de forma idêntica à do algoritmo anterior, pelo que funcionará desde que não se destrua nenhuma mensagem antes de todos os *principais* terem conhecimento delas. Como a destruição das mensagens está dependente dos valores dos vectores de confirmação, a correcção do algoritmo depende da correcção da sua actualização.

A correcção da actualização dos vectores de confirmação está dependente destes nunca reflectirem mensagens que possam não ter sido observadas por um *principal*. Aquando das sessões de sincronização, as entradas do vector de confirmação (que por hipótese se assumem como correctas) são actualizadas de duas formas:

1. Para a entrada do próprio *principal* e do parceiro, o valor será o mínimo dos valores presentes nos seus vectores de resumo, pelo que não reflectem, obviamente, mensagens não observadas.
2. Para as outras entradas, o valor será o máximo dos valores que os dois *principais* sabem ter sido conhecidos, pelo que assumindo que estes valores estão correctos (por hipótese), o valor final estará igualmente correcto.

Desta forma, a condição enunciada está correcta, pelo que os vectores são actualizados correctamente. Como as mensagens só são destruídas quando o seu número de ordem é inferior ou igual ao menor dos números de ordem do vector de confirmação, é evidente que estas só são destruídas após todos os *principais* terem tido conhecimento delas.

5.3 Adaptação do sistema de comunicação epidémica ao repositório de CoObjectos

O repositório de CoObjectos apresentado nesta dissertação utiliza um sistema de comunicação epidémica modificado. Este sistema é utilizado para propagar, entre os diversos servidores que replicam um volume, as operações aplicadas sobre os CoObjectos. A especificidade do problema a resolver levou a que fossem efectuadas as modificações que se apresentam pictoricamente na figura 5.8.

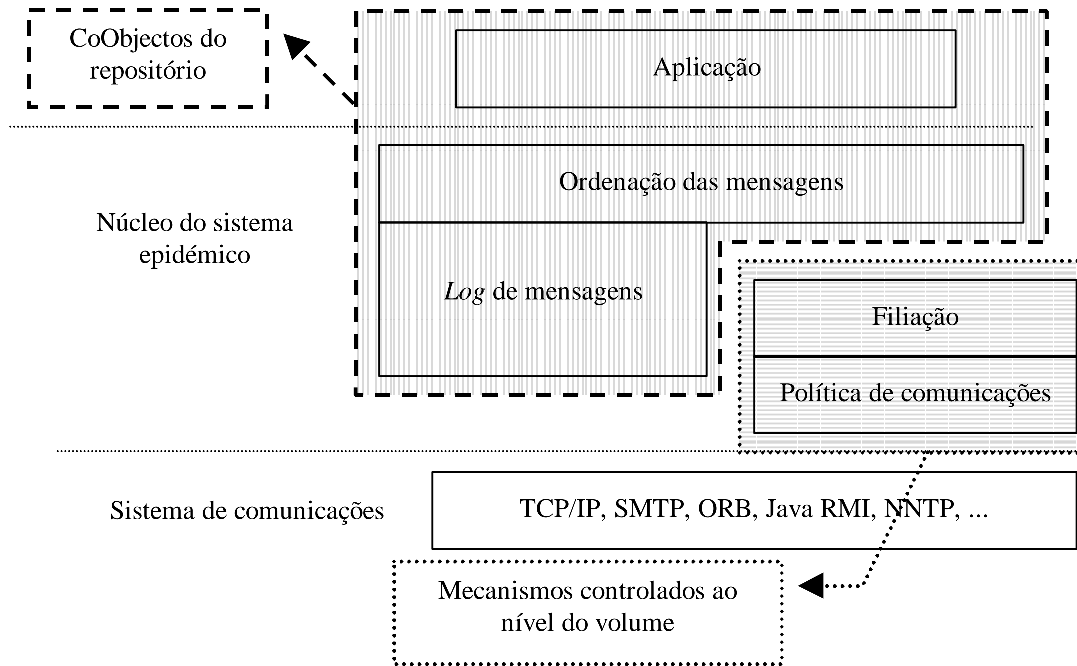

Figura 5.8 – Integração do repositório de CoObjectos num sistema de comunicação epidémica.

Cada um dos CoObjectos tem os seguintes módulos habitualmente presentes num sistema epidémico: um *log* de mensagens – operações; um módulo de ordenação das mensagens; um módulo de processamento das mensagens. O mapeamento entre estes módulos e os componentes do modelo de objectos é evidente e imediato. A informação associada ao processo de propagação epidémica é guardada, em cada CoObjecto, no seu objecto de atributos.

Os mecanismos de gestão da filiação e da política de comunicações são executadas ao nível da unidade de replicação do sistema – o volume. Para tal, existe um CoObjecto *bem-conhecido* em cada volume que gere a filiação do grupo de servidores que replicam cada volume. Este CoObjecto tem, igualmente, a missão de gerir a política de comunicações – a qual pode ser alterada. Os servidores do repositório invocam estes CoObjectos, através da sua conhecida interface, de forma a executar adequadamente a política de comunicações definida e a gerir o grupo de servidores que replicam um dado volume[14].

5.3.1 Adaptação dos protocolos de sincronização

As adaptações introduzidas ao sistema de comunicação epidémica levam à necessidade de rever os protocolos atrás apresentados. Assim, e não tratando o problema da filiação que será apresentado

---

[14] Ao grupo de servidores que replicam um volume chama-se, também, replicadores do volume.

posteriormente, é necessário adequar os protocolos à existência de um conjunto variável de CoObjectos, com o seu *log* de mensagens próprio, que é necessário sincronizar. Os protocolos passam a ser estabelecidos entre servidores que pretendem sincronizar o estado dum volume constituído por um conjunto de CoObjectos.

A introdução de múltiplos CoObjectos pode ser efectuada através do envio e recepção das informações previstas, não apenas em relação a um, mas a múltiplos CoObjectos (e respectivos *logs*). Estas informações são acompanhadas do identificador do CoObjecto a que se referem, como apresentado na figura 5.9. Evidentemente, existe a necessidade de executar um protocolo de sincronização para cada CoObjecto envolvido.

OutVectors := CoObjID, V^i , A^i
InVectors := CoObjID, V^j , A^j

Figura 5.9 – Alterações efectuadas ao algoritmo epidémico com libertação de recursos para permitir múltiplos CoObjectos.

Num repositório de CoObjectos é necessário preparar os protocolos de sincronização para a possibilidade de variação do conjunto de CoObjectos envolvidos, i.e., para a adição e remoção de CoObjectos. A adição dum novo CoObjecto é executada num só servidor, pelo que nas sessões de sincronização seguintes, esse servidor fará referência a um CoObjecto desconhecido pelo outro. Nesta situação desencadeia-se um protocolo de transferência de estado, para que ambos os servidores fiquem com uma cópia do CoObjecto.

Uma solução para o problema reside na introdução dum código de operação, a enviar na troca de mensagens, o qual comandará as acções que são executadas e a informação incluída na mensagem. No repositório apresentado existem as seguintes operações: pedido de operações – em que se envia a informação sobre o processo de sincronização de forma a receber as operações não conhecidas (equivalente à mensagem inicial do algoritmo assíncrono); envio de operações – na qual além da informação sobre o processo de sincronização se enviam um conjunto de operações (equivalente à resposta à mensagem inicial do algoritmo assíncrono); pedido duma cópia dum CoObjecto; envio duma cópia dum CoObjecto – em que se envia o estado desse CoObjecto.

As alterações anteriores levam a um completa reestruturação dos algoritmos apresentados, com a adição dos referidos tipos de operação e identificadores dos CoObjectos. As modificações introduzidas não alteram, no entanto, o comportamento geral do algoritmo, pelo que as características de disseminação de mensagens apresentadas para a comunicação epidémica se mantém. Na figura 5.10 e 5.11 apresenta-se o algoritmo para comunicação assíncrona devidamente modificado. O algoritmo para comunicação síncrona necessitaria de modificações semelhantes.

```
nStep = 1
Send( nStep)
FOR ALL CoObjects
        Send( "InitialRequest")
        Send( CoObjID, OutVectors)
ENDFOR
Send( "NoMoreOps")
```

Figura 5.10 – Passo inicial das sessões de sincronização no repositório de CoObjectos.

```
Receive( nStep)
IF NOT LastStep() THEN Send( nStep + 1)
FOREVER
    Receive( OpID)
    CASE OpID EQUALS
    NoMoreOps:
        EXIT FOREVER
    InitialRequest:
        Receive( CoObjID, InVectors)
        IF Exists( CoObjID) THEN
            UpdateClock()
            UpdateAckVectors()
            IF NOT LastStep() THEN
                Send( "LogUpdate")
                Send( CoObjId, OutVectors, MyMsgs)
            ENDIF
        ELSE
            IF NOT LastStep() THEN
                Send( "StateRequest")
                Send( CoObjId)
            ENDIF
        ENDIF
    LogUpdate:
        Receive( CoObjID, InVectors, PeerMsgs)
        IF Exists( CoObjID) THEN
            InsertInLog( PeerMsgs)
            UpdateVectors()
            UpdateClock()
            UpdateAckVectors()
            IF NOT LastStep() THEN
                Send( "LogUpdate")
                Send( CoObjID, OutVectors, MyMsgs)
            ENDIF
        ELSE
            IF NOT LastStep() THEN
                Send( "StateRequest")
                Send( CoObjID)
            ENDIF
        ENDIF
```

```
        StateRequest:
            Receive( CoObjID)
            IF Exists( CoObjID) THEN
                IF NOT LastStep() THEN
                    Send( "ObjectState")
                    Send( CoObjID, GetState( CoObjID))
                ENDIF
            ELSE
                // Não há nada que se possa fazer
            ENDIF
        ObjectState:
            Receive( CoObjID, ObjState)
            IF Exists( CoObjID) THEN
                IF NOT LastStep() THEN
                    Send( "InitialRequest")
                    Send( CoObjID, OutVectors)
                ENDIF
            ELSE
                CreateNewObjectWith( ID = CoObjID, State = ObjState)
            ENDIF
        ENDCASE
    ENDFOREVER
    FOR Each Unreferenced CoObject
        Send( "InitialRequest")
        Send( CoObjID, OutVectors)
    ENDFOR
    Send( "NoMoreOps")
```

Figura 5.11 – Resposta ao passo inicial das sessões de sincronização no repositório de CoObjectos.

Num repositório de CoObjectos, existem alguns CoObjectos que permanecem constantes durante períodos de tempo nos quais ocorrem múltiplas sessões de sincronização. A partir do momento em que todos os servidores conhecem todas as operações executadas, não são trocadas mais operações durante as sessões de sincronização. Dessa forma, é desnecessário considerá-los durante essas sessões.

No algoritmo epidémico em que se efectua a libertação de recursos existe informação sobre as operações que os diversos servidores já observaram, pelo que é possível verificar se um CoObjecto se encontra nas condições anteriores. Em volumes com uma elevada percentagem de CoObjectos estáveis, esta optimização permitirá uma significativa redução dos recursos envolvidos para executar uma sessão de sincronização.

5.4 Filiação

O módulo de filiação num sistema de comunicação em grupo gere os elementos que pertencem ao conjunto de *principais* que trocam mensagens. Num sistema de comunicação epidémica assume-se, normalmente, que os *principais* não falham permanentemente, pelo que tanto a inserção como a remoção dum *principal* no grupo são actividades desencadeadas como resposta a um pedido. No caso

particular do repositório de CoObjectos proposto, a filiação está relacionada com os servidores que replicam cada volume[15].

Como se referiu anteriormente, a filiação do repositório de CoObjectos é gerida para cada volume pelo CoObjecto de filiação pertencente a esse volume. As operações relevantes que exporta são a inserção e a remoção dum servidor no grupo de servidores que replicam o volume. Do ponto de vista do sistema, trata-se dum CoObjecto normal, aplicando as operações por ordem total e causal com recurso a *undo*/*redo*. Estas características garantem a convergência das *views*[16] nos diferentes replicadores.

Como se antecipa que existirá uma elevada estabilidade na filiação relativa a um volume (da ordem dos dias, pelo menos), preferiu privilegiar-se a compactação da informação relativa aos servidores, em detrimento da celeridade da sincronização da filiação. Desta forma, existe a necessidade de os dois *principais* que participam numa sessão de sincronização se encontrarem na mesma *view* (para que possam interpretar mutuamente os identificadores dos servidores e os vectores versão).

De seguida, explica-se pormenorizadamente o processo pelo qual se faz a gestão da filiação.

5.4.1 Adaptação dos protocolos de sincronização

Nos protocolos estabelecidos entre os servidores relativamente a um dado volume é enviado em cada comunicação o identificador da *view* em que o servidor se encontra. O identificador enviado pelo parceiro é comparado com o do próprio servidor, e caso sejam diferentes inicia-se um protocolo especial de sincronização de filiação. A sessão que se encontrava a decorrer é abortada e será recomeçada de início após a sincronização da filiação. A necessidade de abortar a sessão que está a decorrer, fica a dever-se ao facto de em duas *views* diferentes os identificadores dos servidores poderem ser diferentes, assim como a sua disposição relativa nos vectores versão. Nas figuras 5.12 e 5.13 é apresentado o protocolo executado no caso da comunicação assíncrona.

```
Lock Membership
Send( ViewID)
... (idêntico ao 5.10) ...
ON EXIT Unlock Membership
```

Figura 5.12 – Passo inicial das sessões de sincronização no repositório de CoObjectos.

[15] Por simplicidade, far-se-ão referências apenas a servidores, embora as mesmas sejam relativas a um volume presente num servidor. Assim, um protocolo efectuado entre dois servidores será relativo a um dado volume e um identificador dum servidor será relativo ao volume considerado.

[16] Chama-se *view* a um conjunto de elementos que participam num sistema de comunicação em grupo (no caso presente, os servidores que replicam um volume). Quando este conjunto varia, diz-se que houve uma mudança de *view*.

```
Lock Membership
Receive( PeerViewID)
IF PeerViewID != ViewID THEN
        DoMembershipSync()
        EXIT
ENDIF
IF NOT LastStep() THEN Send( ViewID)
... (idêntico ao 5.11) ...
ON EXIT Unlock Membership
```

Figura 5.13 - Resposta ao passo inicial das sessões de sincronização no repositório de CoObjectos.

Durante o processamento de cada passo dos protocolos (o qual terá sempre uma duração limitada e reduzida) impossibilita-se a variação da filiação, pelo que todo o processamento efectuado decorrerá necessariamente na mesma *view*. Desta forma, toda a informação trocada será válida, incluindo vectores versão e identificadores de servidores.

5.4.2 Sincronização da filiação

O protocolo estabelecido por dois servidores para sincronizarem a filiação é muito simples, reduzindo-se à troca simples de todas as operações presentes no *log* do CoObjecto de filiação. Desta forma, em ambos os servidores, os CoObjectos de filiação ficam com o mesmo conjunto de operações, que ao serem aplicadas por ordem total e causal conduzem a um estado comum (à mesma *view*) – todas as operações do CoObjecto de filiação são executadas imediatamente podendo ser posteriormente desfeitas e refeitas para garantir a ordem total e causal. De seguida, e caso seja necessário, todos os CoObjectos do volume são actualizados, resolvendo-se as referências aos servidores – vectores versão e referências posicionais. Durante o decurso deste protocolo todo o acesso ao volume é impossibilitado, por forma a que todas as acções sejam executadas totalmente dentro duma mesma *view*. Na figura 5.14 apresenta-se o protocolo descrito.

```
Send( log)                              Receive( PeerMbrshipOps)
Receive( PeerMbrshipOps)                Send( log)
Lock Volume                             Lock Volume
InsertInLog( PeerMbrshipOps)            InsertInLog( PeerMbrshipOps)
FIX ALL CoObjects                       FIX ALL CoObjects
Unlock Volume                           Unlock Volume
```

Figura 5.14 – Protocolo de sincronização do CoObjecto de filiação.

As operações executadas entre o *lock* e o *unlock* do volume devem ser realizadas atomicamente, de forma a que se garanta que todos os CoObjectos dum volume se encontram na mesma *view*. A necessidade desta condição é evidente quando se pensa na forma de funcionamento de repositório e respectivas sessões de sincronização – em que se fazem comparações simples dos vectores versão associados a cada CoObjecto.

Na figura 5.15 apresenta-se um exemplo do processo que decorre aquando da sincronização de filiação. No CoObjecto de filiação a identificação de cada servidor é efectuada através de identificadores únicos, não reutilizáveis e gerados de forma independente aquando da junção do mesmo ao grupo. Desta forma, a sincronização deste CoObjecto em situações de diferentes *views* é trivial, seguindo o processo normal do modelo de objectos proposto. O estado do CoObjecto de filiação define uma função $posição \rightarrow servidor$, sendo os servidores identificados em todos os outros CoObjectos do volume por esta sua posição (ou posição relativa no caso dos vectores versão). Este facto leva à necessidade de actualizar estas referências sempre que existe uma alteração à filiação que altere a posição de algum dos servidores.

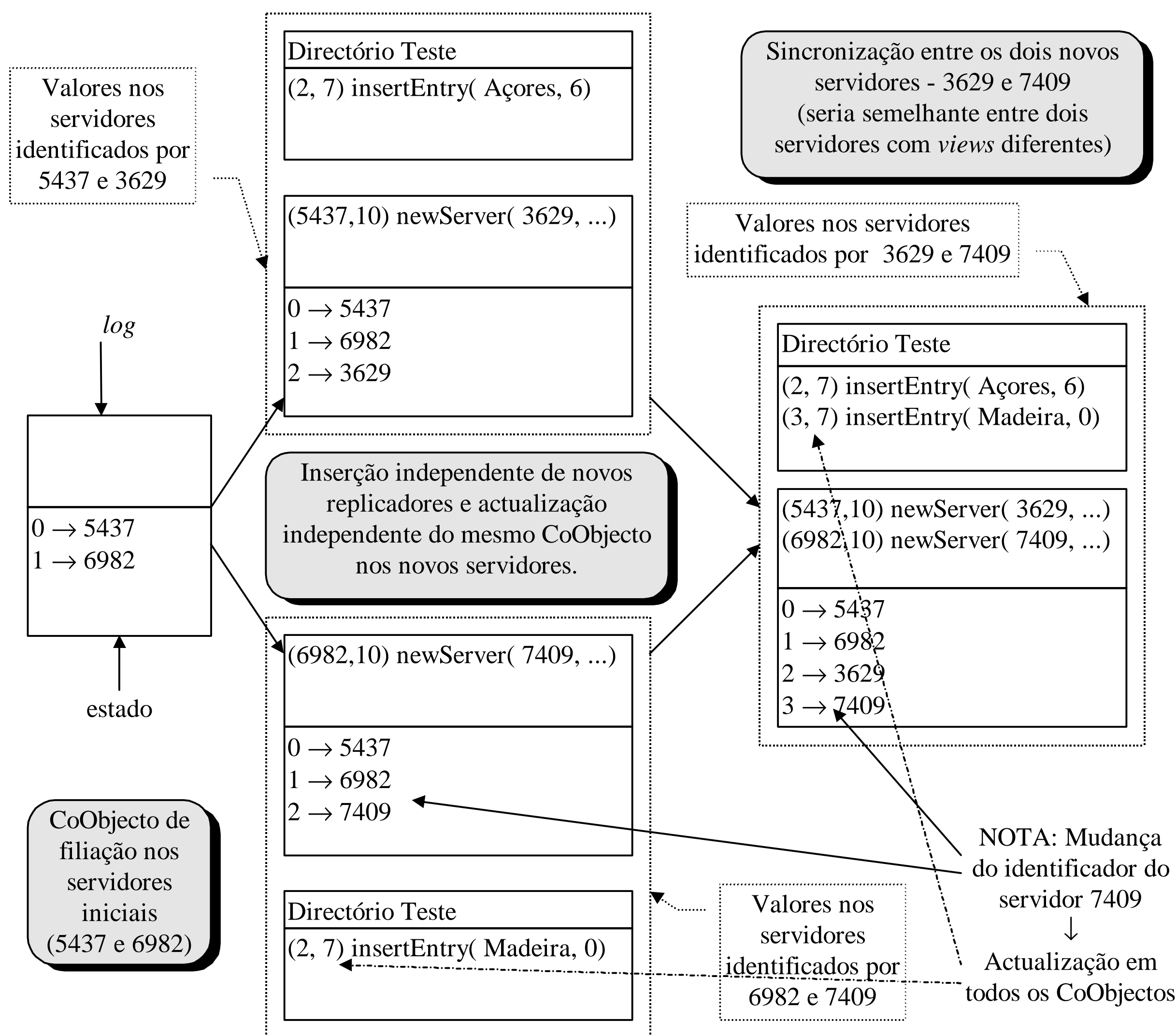

Figura 5.15 – Exemplo de mudanças de filiação independentes e posterior sincronização. A sincronização apresentada seria executada em dois passos: 1 – sincronização do CoObjecto de filiação e actualização de todos os CoObjectos; 2 – sincronização dos outros CoObjectos.

Como é fácil de perceber, o processo de actualização de todos os CoObjectos é necessariamente moroso, pelo que a solução proposta apenas é praticável na medida em que se antecipa que as mudanças de filiação serão escassas (com uma estabilidade da ordem dos dias, pelo menos). Caso a frequência das mudanças de filiação fosse muito elevada, poder-se-ia utilizar, em alternativa, os identificadores globais dos servidores. Assim, os vectores versão $- T : T(x) = v$ – seriam transformados em vectores de pares ordenados – $T : T(x) = (ServerID, v)$. Desta forma, eliminar-se-ia a necessidade de actualizar todos os CoObjectos aquando das mudanças de *view*.

A solução anterior faria aumentar o espaço necessário para armazenar aquelas referências (um identificador único, global e gerado independentemente é necessariamente muito mais extenso do que um simples número de ordem) e dificultaria a sua comparação (neste momento, a comparação entre dois vectores versão é efectuada por comparação simples das entradas correspondentes de cada um, enquanto de outra forma seria necessário saber que entradas corresponderiam aos mesmos servidores). Estas desvantagens levaram à opção pela solução apresentada anteriormente, que se espera mais eficiente nas condições normais de utilização do sistema proposto.

Outra solução possível seria os servidores, aquando das sessões de sincronização, trocarem as suas funções $posição \rightarrow servidor$ e fazerem a actualização das referências recebidas (o que é possível pelas funções anteriores serem bijectivas) – em todas a informação associada ao processo de sincronização e também nas mensagens propagadas. Esta solução, tem o inconveniente de tornar mais complexas as sessões de sincronização, pelo que não foi igualmente adoptada.

5.4.3 Sincronização das referências emitidas por clientes

No repositório proposto, os clientes, ao submeterem para os servidores as operações executadas, apresentam um vector versão que identifica a versão do CoObjecto ao qual têm acesso (o qual pode ser usado para determinar as dependências entre as diferentes alterações). Como existe a possibilidade de a filiação ter mudado entre o momento em que o cliente obteve a sua cópia do CoObjecto e aquele em que submeteu as alterações, é necessário que adicionalmente se apresente um identificador da *view* em que o vector anterior deve ser interpretado. O servidor, ao guardar no CoObjecto de filiação a história dos seus estados (independentemente do *log*), consegue interpretar este vector convertendo-o para a *view* corrente (quando este não conseguir ser interpretado por a história ter sido truncada, ou se assume que as alterações foram efectuadas à versão actual ou se notificam os utilizadores da situação).

5.4.4 Protocolo de entrada no grupo

A entrada dum novo servidor no grupo de servidores que replicam um dado volume, deve ser sempre patrocinada por outro servidor que já pertença a esse conjunto, e segue os seguintes passos:

1. O novo servidor contacta o patrocinador a indicar-lhe a intenção de passar a replicar o volume.

2. O patrocinador toma nota do início da inserção do novo servidor, aplica a operação de inserção dum novo elemento (servidor) no CoObjecto de filiação e procede às alterações decorrentes da mudança de *view* (com o possível aumento do comprimento dos vectores versão) – estas acções devem ocorrer de forma atómica.

3. O patrocinador envia ao novo servidor uma cópia do volume, com todos os seus CoObjectos.

4. O novo servidor, ao receber a cópia do volume, actualiza para todos os CoObjectos o valor do seu número de ordem, que passa a ser igual ao máximo dos números de ordem dos outros servidores (para que não exista a possibilidade de o novo servidor emitir uma operação com um número de ordem inferior ao que se sabe já ser conhecimento de todos e que os outros servidores pensem já ter observado). De seguida informa o patrocinador que está em funcionamento, o qual apaga a informação relativa à inserção desse servidor (esta informação é necessária para garantir que um patrocinador nunca abandona um grupo de replicadores antes de terminar a inserção dum servidor que patrocina).

A partir deste momento o novo servidor está disponível para servir os clientes e participar em sessões de sincronização. Quando por qualquer motivo, algum dos passos descritos anteriormente falha, o protocolo é recomeçado do princípio usando o mesmo patrocinador. No caso do passo 2 já ter sido executado anteriormente (o que se pode verificar pela pertença do novo servidor ao conjunto de replicadores), o mesmo não é executado.

A inserção do novo elemento no conjunto de replicadores impõe uma mudança de *view*. Desta forma, as sessões de sincronização posteriores levam à sincronização da filiação (com a consequente propagação da informação sobre o novo servidor). No caso destas sessões ocorrerem antes de o protocolo anterior ter terminado (e depois do passo 2 ter sido executado), podem existir tentativas de contacto com o novo servidor sem ele ter conhecimento do estado do volume. Neste caso, estas comunicações devem falhar, voltando a ser executadas posteriormente.

5.4.5 Protocolo de saída do grupo

A saída dum servidor do conjunto de replicadores dum volume deve ser sempre patrocinada por outro servidor que também pertença a esse conjunto (quando o único servidor que replica o volume é o que pretende sair o volume é simplesmente apagado). Esta só pode ser efectuada quando o servidor, que pretende sair, não seja o patrocinador dum novo servidor que ainda não tenha completado o seu protocolo de entrada no grupo. O protocolo segue os seguintes passos:

1. O servidor que pretende sair marca-se localmente como pré-apagado, i.e., não podendo a partir deste momento patrocinar nenhum outro servidor, nem atender os pedidos dos clientes.

2. Executa uma sessão normal de sincronização com o patrocinador. Este, no fim do seu último passo aplica a operação de remoção do servidor em questão ao CoObjecto de filiação do volume (esta operação já não influencia os dados trocados entre ambos).

3. O servidor liberta os recursos ocupados pelo volume.

Este protocolo garante que todas as operações, que tenham sido submetidas para o servidor que deixa de replicar o volume, serão consideradas, pois são garantidamente propagadas para um servidor que continua em funcionamento. Estas operações continuarão a ser consideradas nas sessões de sincronização entre servidores, mantendo os resumos, presentes nos CoObjectos, referências aos servidores que entretanto saíram do grupo. A libertação destas referências será efectuada posteriormente, como culminar do seguinte processo executado sobre o CoObjecto de filiação:

1. Quando, no CoObjecto de filiação, se confirma que uma operação de saída dum servidor foi recebida por todos os outros e será apagada, aplica-se uma operação a indicar que se deve verificar a possibilidade de remover as referências a esse servidor.

2. Periodicamente, cada servidor verifica quais dos servidores saídos (e indicados pela operação referida anteriormente) podem ser removidos (o que acontece quando, para todos os CoObjectos, todas as operações originadas nesses servidores puderem ser apagadas). Se existir algum que esteja nessa condições, aplica uma operação que indica a sua possibilidade de remover as referências a esse servidor.

3. Quando todos os servidores, que se mantém a replicar um volume, tiverem emitido uma operação a indicar a possibilidade de remoção das referências a um dado servidor, é aplicada uma operação a indicar a remoção efectiva dessas referências. Esta operação leva a que o número de ordem atribuído ao servidor que deixou de replicar o volume deixe de estar atribuído, e que consequentemente essa posição deixe de ser considerada nos vectores versão.

Como o processo anterior é executado em todos os servidores, o passo 3 do mesmo poderá dar origem a múltiplas operações de remoção de referências (por verificações concorrentes das condições referidas). Este facto não constitui, no entanto, nenhum problema para o esperado funcionamento do sistema, pois como as operações executadas sobre o CoObjecto de filiação são ordenadas totalmente, apenas será executada a que for ordenada como primeira.

O protocolo anterior garante que todas as operações executadas serão consideradas no funcionamento do sistema, e que as referências a um servidor apenas serão apagadas quando não existir mais nenhuma operação que nele tenha sido originada. Estas condições garantem que será sempre possível fazer a ordenação de todas operações, mesmo que o servidor em que tenham sido originadas tenha saído do grupo (o que não aconteceria caso as referências fossem apagadas imediatamente).

5.4.6 Identificadores nas *views*

Como já se referiu, a relação $posição \rightarrow servidor$ estabelecida através do estado do CoObjecto de filiação comanda as referências aos servidores existentes em cada *view*. Os servidores são referidos pela sua posição nesta relação, a qual poderá variar com as alterações da filiação. No CoObjecto de

filiação, as posições são atribuídas consoante as disponibilidades, inserindo e removendo entradas como ficou explicado anteriormente. Após ter sido removida uma entrada, essa posição só é utilizada por um servidor que se junte ao grupo de replicadores posteriormente. Na figura 5.16 apresenta-se uma possível evolução da filiação num volume. A operação *newViewHst* não modifica a filiação, apenas sendo usada para manter a história das *views* pelas quais o volume passou (de modo a possibilitar a sincronização nas operações enviadas pelos clientes – ver 5.4.3).

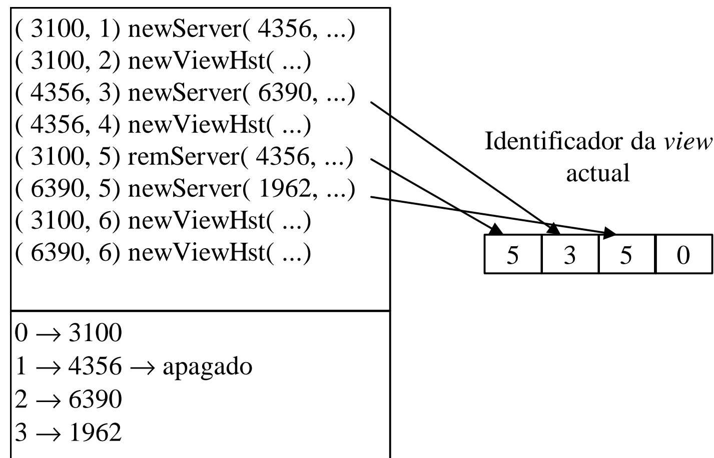

Figura 5.16 – Evolução possível do filiação num volume.

Uma *view* é identificada por um vector versão que contém em cada posição a ordem da última operação originada nesse servidor que modifica a filiação – inserção e remoção dum elemento e remoção das referências a um servidor.

Como o número de ordem dado a uma operação é superior aos números de ordem já conhecidos (o protocolo de propagação epidémica com libertação de recursos conduz a esta propriedade), as novas alterações na filiação têm sempre números de ordem superiores aos das mudanças anteriores, pelo que o identificador da *view* será sempre único quando as alterações estão causalmente dependentes.

No caso em que as operações são concorrentes, e podem ter o mesmo número de ordem, o identificador da *view* é único, porque esse número de ordem corresponderá necessariamente a servidores diferentes (logo a posições diferentes no vector – só poderia ser a mesma posição caso se tratasse de novos servidores que entrassem concorrentemente utilizando patrocinadores diferentes, mas neste caso essas entradas teriam necessariamente de estar reflectidas em servidores diferentes, logo em posições diferentes).

Capítulo 6

Implementação

De forma a avaliar a funcionalidade e exequibilidade do modelo proposto, foi implementado um protótipo de repositório de CoObjectos. Este protótipo implementa totalmente as características apresentadas na secção 4.1 e suas subsecções, baseadas numa arquitectura geral constituída por um conjunto de servidores que replicam os volumes e por um conjunto de clientes que efectuam *caching* dos CoObjectos necessários aos utilizadores. O modelo de objectos proposto (secção 4.2) foi igualmente implementado, merecendo especial relevo a construção dum conjunto de componentes base que permite a fácil construção de novos tipos de dados. A propagação das operações entre servidores baseou-se no algoritmo apresentado no capítulo anterior para comunicação epidémica assíncrona utilizando libertação de recursos. A gestão da filiação do grupo de servidores que replicam cada volume foi implementada utilizando, igualmente, os protocolos apresentados anteriormente.

Neste capítulo será apresentada uma descrição da implementação do protótipo. Esta descrição não pretende ser exaustiva, antes focando os aspectos que se consideram fundamentais.

6.1 Suporte da implementação

O sistema que serve de suporte ao desenvolvimento do protótipo do repositório de CoObjectos necessita de fornecer o seguinte conjunto de características:

1. Acesso a um conjunto de primitivas de comunicação entre computadores, por forma a permitir o desenvolvimento de todo o sistema distribuído englobando clientes e servidores.
2. Permitir a fácil interoperabilidade entre sistemas com diferentes arquitecturas físicas, i.e., suportar heterogeneidade. Esta característica pode ser obtida com recurso a representações *standard* dos dados (por exemplo ASN.1, XDR, ou outra), devendo ser alcançada, de preferência, de forma totalmente transparente para os utilizadores do sistema – programadores.

3. Possibilidade de adicionar e ligar dinamicamente novas secções de código, sem necessitar de recompilar todo o sistema. Esta característica é fundamental para permitir a adição de novos tipos de dados (implementados como classes de objectos), com a sua codificação própria, mantendo o sistema em funcionamento.
4. Permitir a gravação fácil e automática em suporte estável do estado dos objectos. Esta característica permite facilitar o desenvolvimento de novos tipos de dados, pois automatiza a resolução do problema da persistência do estado dos objectos manipulados, aliviando o programador da necessidade de definir e implementar formatos para suporte persistente (o que, além de entediante, é propício à introdução de erros). No entanto, estes formatos deverão poder ser utilizados sempre que desejável.

Estas características representam o conjunto necessário óptimo para permitir o fácil e rápido desenvolvimento do repositório de CoObjectos. Várias soluções, englobando linguagens de programação e bibliotecas associadas, poderiam ter sido utilizadas para a implementação do protótipo. Algumas das características, se inexistentes para uma determinada solução, poderiam ter sido programadas a partir do núcleo disponível.

A linguagem Java, ao fornecer na sua distribuição *standard* (versão 1.1) todas as características referidas, tornou-se, no entanto, a escolha óbvia. Como o objectivo do desenvolvimento do protótipo era avaliar a funcionalidade e exequibilidade do modelo proposto, considerou-se que o desempenho do mesmo era secundário em relação à facilidade e rapidez de desenvolvimento. Desta forma, decidiu-se que o desenvolvimento do protótipo seria efectuado totalmente na linguagem Java.

A anterior decisão permite beneficiar da quase universal portabilidade da mesma, pelo que o protótipo criado pode ser testado em múltiplas plataformas. Adicionalmente, devido à sua utilização generalizada nos *browsers* da *World Wide Web* e à possibilidade de efectuar o carregamento dinâmico dos programas a partir dum servidor de HTTP, permite uma elevada visibilidade às aplicações nela implementadas.

O facto de todo o sistema ser implementado em Java, incluindo o modelo de objectos, torna esta linguagem a ideal para o desenvolvimento de aplicações que utilizem este repositório de CoObjectos. No entanto, o crescente número de definições de modos de interacção entre a linguagem Java e outras linguagens permitirá a utilização das mesmas para o desenvolvimento de aplicações que utilizem o repositório de CoObjectos.

6.2 Arquitectura modular do sistema

O repositório de CoObjectos implementado é constituído por dois grandes componentes: clientes e servidores. Cada um destes macro-componentes é composto por diversos módulos que interactuam

entre si por forma a realizar o modelo apresentado. Na figura 6.1 apresenta-se a representação desta organização.

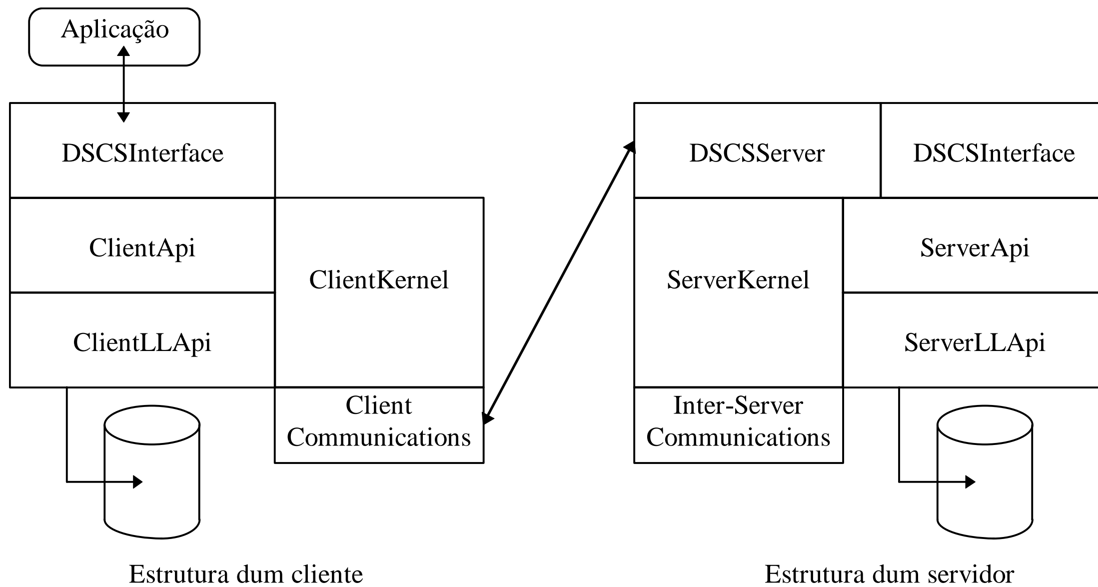

Figura 6.1 – Organização modular dos componentes do repositório de CoObjectos.

6.2.1 API de interface do sistema

As aplicações dos utilizadores interactuam com o repositório de CoObjectos através da API de interface de repositório, definida pelo módulo DSCSInterface, apresentada na figura 6.2.

Esta API permite aos clientes manipular os conceitos de directório e *link* simbólico. Adicionalmente, esta *interface* utilizada pelos clientes mantém uma localização actual no espaço global de nomes (constituído por um volume e por um directório relativo a esse volume), de forma semelhante ao directório actual (*current*) nos sistemas UNIX. A nomeação dos CoObjectos pode então ser efectuada de forma global ou relativa à localização actual, como indicado na figura 6.3.

Como se pode constatar, existe a possibilidade, adicional, de indicar um servidor preferencial onde os CoObjectos referidos devem ser procurados (por exemplo *//dagora.di.fct.unl.pt/docLib:/doc/MBone* refere-se ao CoObjecto *MBone* presente no directório */doc* do volume *docLib*, o qual deve ser obtido preferencialmente a partir do servidor presente em *dagora.di.fct.unl.pt*). Caso o servidor preferencial não seja indicado, o sistema encarrega-se de usar a informação disponível para contactar o servidor considerado mais apropriado. Na implementação actual, quando não existe informação sobre nenhum servidor que replique um dado volume, a resolução do nome falha, i.e., não existe nenhum mecanismo global de pesquisa da localização de volumes.

```
public class DSCSInterface
{
        // Métodos de sincronização do estado dos servidores
   public void synchronizeVolume( String volumeName, String fromServer, String inServer);
   public void synchronizeObject( String objectName, String fromServer, String inServer);

        // Métodos relativos à gestão dos volumes
   public void createVolume( String inServer, String volumeName);
   public void replicateVolume( String volumeName, String fromServer, String inServer);
   public void unreplicateVolume( String volumeName, String inServer);
   public String[] listVolumes( String inServer);

        // Métodos de manipulação dos CoObjectos
   public DSCSCapsule getObject( String name, int flags);
   public DSCSAttrib getObjectAttrib( String name, int flags);
   public void putObject( DSCSCapsule capsule);
   public void putObject( String name, DSCSCapsule capsule);
   public void deleteObject( String name);

        // Métodos de manipulação de directórios
   public void createDirectory( String name);
   public void removeDirectory( String name);
   public DirectoryCapsule getCurrentDirectory( int flags);

        // Métodos de manipulação de links simbólicos
   public void createSymLink( String source, String destination)

        // Métodos relativos a localização actual
   public void changeVolume( String name)
              throws VolumeNotFoundException;
   public void changeDirectory( String name)
              throws NotDirectoryException;
   public String currentVolumeName();
   public String currentDirectoryName();

        // Métodos auxiliares de manipulação de preferências
   public boolean setConnection( boolean flag);
   public void setPrefServer( String serverName);
}
```

Figura 6.2 – API de interface do repositório de CoObjectos. Esta API é a utilizada pelas aplicações do sistema.

Nome ::= NomeTotal | NomeGlobal | NomeInterno
NomeTotal ::= //**NomeServidorPreferencial**/NomeGlobal
NomeTotal ::= //**NomeServidorPreferencial**/NomeInterno
NomeGlobal ::= **NomeVolume**:NomeInterno
NomeInterno ::= NomeRelativo | NomeAbsoluto
NomeAbsoluto ::= /NomeRelativo
NomeRelativo ::= NomeRelativo/**NomeSimples**
NomeRelativo ::= **NomeSimples**

Figura 6.3 – *Nome* representa um identificador simbólico dum CoObjecto nas suas múltiplas formas (os identificadores usados a carregado representam identificadores finais)

Esta interface está igualmente disponível nos servidores, de modo a que, tanto os módulos constituintes do servidor como as implementações dos tipos de dados possam fazer recurso aos nomes simbólicos dos CoObjectos. Todas as interpretações envolvendo nomes simbólicos são executadas por este módulo, o qual se encarrega de os transformar nos correspondentes identificadores a nível do sistema. Estas transformações são executadas fazendo a interpretação dos directórios e *links* simbólicos – os quais são implementados, como já se disse, à custa de CoObjectos de classes pré-definidas.

Os métodos disponibilizados por esta interface são facilmente compreensíveis pela sua definição. As suas funcionalidades são implementadas fazendo recurso a um módulo inferior do sistema, o qual disponibiliza a API interna.

6.2.2 API interna do sistema

O módulo da API interna do sistema implementa todas as funções relacionadas com a sincronização do estado dos servidores, com a gestão dos volumes e manipulação dos CoObjectos. Neste módulo, todas as entidades do sistema – volumes e CoObjectos – são nomeadas pelos seus identificadores do sistema. Os CoObjectos que implementam os directórios e os *links* simbólicos, utilizados no módulo anterior, são CoObjectos normais. Na figura 6.4 apresentam-se as definições dos métodos disponíveis, as quais são auto-explicativas.

```
public interface DSCSApi
{
        // Métodos de sincronização do estado dos servidores
   public void synchronizeVolume( String volumeName, String fromServer, String inServer);
   public void synchronizeObject( GlobalOID goid, String fromServer, String inServer);

        // Métodos relativos à gestão dos volumes
   public void createVolume( String server, String volume);
   public void replicateVolume( String volumeName, String fromServer, String inServer);
   public void unreplicateVolume( String volumeName, String inServer);
   public String[] listVolumes( String server);
   public boolean existsVolume( String volume);

        // Métodos de manipulação dos CoObjectos
   public DSCSCapsule getObject( GlobalOID goid, int flags);
   public DSCSAttrib getObjectAttrib( GlobalOID goid, int flags);
   public void putObject( DSCSCapsule capsule);
   public void putObjectAttrib( DSCSAttrib attrib);
   public void putObject( DSCSCapsule capsule, GlobalOID goid);
   public void deleteObject( GlobalOID goid, String name);

        // Métodos auxiliares de manipulação de preferências
   public boolean setConnection( boolean flag);
   public void setPrefServer( String serverName);
}
```

Figura 6.4 – API interna do sistema.

O módulo que disponibiliza a API interna do sistema tem implementações diferentes para o caso de se tratar dum servidor (*ServerApi*) ou dum cliente (*ClientApi*), devido às diferentes operações a executar nos dois casos. Este módulo faz recurso a dois outros módulos, o núcleo (*ServerKernel* no caso do servidor e *ClientKernel* no caso do cliente) e um módulo que disponibiliza uma API de baixo nível (*ServerLLApi* no caso do servidor e *ClientLLApi* no caso do cliente). No caso do servidor, a API interna do sistema é igualmente invocada para executar as operações desencadeadas pelos clientes, as quais são apresentadas aos servidores através duma interface específica (*DSCSServer*) que será apresentada posteriormente.

6.2.3 API de baixo nível do sistema

Os módulos que implementam a API de baixo nível do sistema são muito semelhantes, sendo responsáveis por abstrair o suporte físico estável no qual são armazenados os CoObjectos – sistema de ficheiros. Na figura 6.5 apresenta-se a definição dessa API. O seu método principal é o que retorna um *handle* para a representação em disco dum dado CoObjecto. Este *handle* é a base para toda a manipulação da representação em suporte estável dos CoObjectos, a qual pode estar armazenada num diferente número de ficheiros (correspondentes, normalmente, a cada um dos diversos objectos em que o CoObjecto está organizado).

```
public abstract class DSCSLowLevelApi
{
   public abstract ObjectHandle getObjectHandle( GlobalOID goid, int flags);
   protected boolean exists( ObjectHandle oh);

         // Representação baixo-nível – vectores de bytes
   public ObjectLLRepresentation getObjectRepresentation( ObjectHandle oh);
   public void putObjectRepresentation( ObjectLLRepresentation obj, ObjectHandle oh);

         // Streams associados aos CoObjectos
   public OutputStream getOutStream( ObjectHandle oh, int type, boolean compressed)
               throws DSCSIOException;
   public ObjectOutputStream getObjOutStream( ObjectHandle oh, int type,
               boolean compressed) throws DSCSIOException;
   public InputStream getInStream( ObjectHandle oh, int type, boolean compressed)
               throws ObjCompNotFoundException;
   public ObjectInputStream getObjInStream( ObjectHandle oh, int type,
               boolean compressed) throws ObjCompNotFoundException;
   public void removeObject( ObjectHandle oh);
}
```

Figura 6.5 – API de baixo nível do sistema.

6.2.4 Outros módulos

O núcleo do sistema apresenta um conjunto de métodos específicos para cada um dos macro-componentes – servidor ou cliente. A diferença no conjunto de métodos disponíveis exterioriza, como é óbvio, as diferenças de funcionalidades existentes.

O núcleo do cliente tem como principal missão a gestão das réplicas secundárias (*cache*). Gere, ainda, um conjunto de informações necessárias ao bom funcionamento do sistema (por exemplo, quais os servidores que replicam um dado volume, quais os servidores preferenciais, ...) e é responsável pela determinação da necessidade de comunicação com os servidores (para obter cópias de CoObjectos, enviar alterações executadas pelos utilizadores, ou obter informações). Para efectuar estas comunicações faz recurso ao módulo de comunicações do cliente (*Client Communications*), o qual se responsabiliza por gerir da forma mais eficaz possível os recursos comunicacionais existentes face às necessidades apresentadas pelo núcleo. Este módulo, em conjunto com o *DSCSServer*, constituem o subsistema de comunicações cliente/servidor, o qual será analisado em maior detalhe posteriormente.

O núcleo do servidor disponibiliza um conjunto de serviços de gestão das réplicas dos volumes. Nestes serviços incluem-se os necessários à modificação da filiação dos replicadores dum volume, à manipulação de novos CoObjectos e ao desencadear de processos de sincronização. Os processos de sincronização podem ser desencadeados como resposta a um pedido explícito efectuado através da API do sistema, ou periodicamente de forma automática, conforme definido para o volume em questão. Para executar o processo de sincronização, o núcleo recorre ao módulo de comunicações entre servidores (*Inter Server Communications*), o qual se encarrega de gerir as comunicações necessárias à sua execução. Este módulo constitui o subsistema de comunicações servidor/servidor e será analisado posteriormente.

6.3 Subsistemas de comunicação

Nesta secção descrevem-se os subsistemas de comunicação implementados, que correspondem a dois paradigmas muito utilizados: cliente/servidor e *peer*. Apesar de terem sido criados especificamente para o repositório de CoObjectos, a sua base poderia ser utilizada na criação de diferentes sistemas, introduzindo modificações mínimas. A sua implementação apresenta algumas simplificações relativas ao modelo nas falhas (mais significativas no subsistema cliente/servidor), as quais poderiam ser eliminadas com recurso a técnicas transaccionais bem conhecidas (e/ou às sugestões que são apresentadas). No entanto, dados os objectivos com que o protótipo foi implementado e as limitações de tempo existentes, decidiu-se privilegiar outros aspectos do sistema.

6.3.1 Subsistema de comunicação cliente/servidor

Este subsistema é responsável por efectuar todas as comunicações necessárias ao correcto funcionamento das aplicações dos clientes. Para tal, tem por missão estabelecer as comunicações com os servidores, invocando os métodos que o núcleo do cliente lhe indica. A interface que os servidores exportam é a apresentada na figura 6.6.

```
public interface DSCSServer
        extends Remote
{

  public DSCSUID genDSCSUID()
        throws RemoteException, NoUIDException;

        // Métodos relativos aos volumes
  public void createVolume( String volumeName)
              throws RemoteException;
  public void replicateVolume( String volumeName, MbrshipElement fromServer)
              throws RemoteException;
  public void unreplicateVolume( String volumeName)
              throws RemoteException;
  public String[] listVolumes()
              throws RemoteException;

        // Métodos de sincronização
  public void synchronizeVolume( String volumeName, MbrshipElement fromServer)
              throws RemoteException;
  public void synchronizeObject( GlobalOID goid, MbrshipElement fromServer)
              throws RemoteException;

        // Métodos relativos aos CoObjectos
  public ExtRepresentation getObject( GlobalOID goid)
              throws RemoteException;
  public ExtRepresentation getModifiedObject( GlobalOID goid, Timevector tv,
              Timevector viewID) throws RemoteException, ObjectNotModified;
  public void submitChanges( GlobalOID goid, Timevector viewID, OperationSeq ops)
              throws RemoteException;
  public void submitNewObject( GlobalOID goid, ObjectLLRepresentation objCore)
              throws RemoteException;

}
```

Figura 6.6 – API exportada pelos servidores para acesso pelos clientes.

Da parte do servidor, o subsistema de comunicações limita-se a receber as invocações executadas remotamente, a executar as operações pedidas e a enviar os correspondentes resultados. Para estes efeitos utiliza, sempre que necessário, a API interna do sistema.

Do lado do cliente, e devido à possibilidade de desconexão do mesmo, pode existir a necessidade de armazenar as invocações para posterior transmissão. Este armazenamento pode servir também para fazer face à escassez de recursos comunicacionais, em situações de conexão fraca (com baixa largura

de banda). Nestas situações podem utilizar-se as prioridades das operações e o conhecimento sobre a relevância da sua ordenação para executar com maior celeridade as operações prioritárias (deve ter-se em atenção que a prioridade das operações deve estar primordialmente relacionada com o tempo de espera que as mesmas provocam nos utilizadores do sistema).

Na implementação actual do sistema, no entanto, as características de gestão dos recursos comunicacionais não foram consideradas. Assim, criam-se filas de operações a serem enviadas para cada destino – servidor ou volume. Nas operações destinadas a um dado volume, testam-se todos os servidores, que se sabe replicarem esse volume, de forma ordenada. Esta ordenação varia dinamicamente consoante a disponibilidade apresentada anteriormente por esses servidores e as preferências reveladas explicitamente pelos utilizadores (através da invocação do método correspondente na API de interface do repositório, ou da sua indicação no nome simbólico dos CoObjectos). Poder-se-ia desenvolver um mecanismo mais eficiente utilizando a noção de distância ao servidor, a qual deve de ser necessariamente dinâmica (em [Sat96b] apresenta-se um sistema em que as características da conectividade são conhecidas dinamicamente). Dentro de cada fila, as operações são executadas (enviadas para o servidor) sequencialmente.

Para cada operação exportada pelo servidor está definida uma classe que implementa a interface da figura 6.7. Assim, em tempo de execução, cada operação será representada no sistema por um objecto, instância duma das classes atrás referidas. Desta forma, o sistema de comunicações pode tratar todas as operações de forma uniforme porque é o método *invoke* da classe que define a operação a ser executada. O sistema de comunicações responsabiliza-se apenas por determinar a acessibilidade dos servidores, passando para os objectos que representam as operações referências aos mesmos. Estas referências são usadas para executar as invocações remotas correspondentes.

```
interface KernelOp
{
  public Object invoke( DSCSServer server, String serverName)
           throws RemoteException;
      // Métodos relacionados com propriedades das operações
      // (não usados na implementação actual)
  public boolean canCommut( KernelOp op);
  public byte priority();
  public boolean idempotent();
      // Destino da invocação
  public boolean toServer();
  public boolean toVolume();
  public String whichServer();
  public String whichVolume();
}
```

Figura 6.7 – Interface que as classes que representam as operações a invocar num servidor devem implementar.

Quando existem operações a entregar determina-se periodicamente a acessibilidade dos servidores. Na implementação actual utiliza-se o sistema Java RMI [Sun96] para fazer o transporte das invocações

remotas. No futuro, podem ser utilizados sistemas de objectos distribuídos como o CORBA [OMG91] ou o Microsoft DCOM para executar estas invocações.

A implementação actual do sistema apresenta deficiências no caso da existência de falhas. No processo de execução duma invocação remota: envio da operação → recepção da operação → execução da operação (com a consequente gravação dos efeitos) → envio do resultado → recepção do resultado (com consequente registo da execução da operação); a existência de falhas pode impossibilitar o cliente e o servidor de terem uma visão coerente sobre o facto da operação ter sido executada (com a consequente divergência quanto à necessidade de voltar a ser executada). Este problema foi profundamente estudado no âmbito dos sistemas distribuídos (e principalmente das bases de dados) [Tan95, CDK94], usando-se, pelo menos duas soluções para o ultrapassar: implementação de sistemas transaccionais, com protocolos *two-phase commit* e *logging*; ou utilização apenas de operações idempotentes[17].

As operações exportadas pelos servidores não são todas idempotentes – por exemplo, se a operação *submitChanges* for executada duas vezes, relativamente à mesma sequência de acções (operações), esse facto corresponderá à execução por duas vezes dessas acções e o resultado poderá não ser o esperado. No entanto, pode simular-se a sua idempotência acrescentando, no cliente, um número de ordem à operação e guardando os números de ordem já executados (e os seus resultados) nos servidores[18]. Assim, quando uma operação já executada é recebida de novo no servidor, ela não necessita de voltar a ser executada, devolvendo-se imediatamente os seus resultados. Como o número de clientes é potencialmente ilimitado, os servidores necessitariam de espaço igualmente ilimitado para guardar esse números de ordem, mas podem executar-se algumas optimizações que limitam o espaço necessário nos servidores:

- Como os clientes enviam as operações sequencialmente, apenas existe necessidade de guardar a última operação executada, pois os clientes apenas enviarão uma nova operação quando tiverem a confirmação que a anterior foi executada (tiverem recebido os seus resultados).
- Após o envio duma sequência de operações, os clientes enviam uma informação de que a sequência terminou com sucesso, pelo que os servidores podem apagar o resultado da última mensagem – se esta mensagem for processada pelo servidor, mas o cliente não tiver conhecimento do seu

---

[17] Uma operação diz-se idempotente quando executada mais duma vez tem os mesmos efeitos e apresenta os mesmos resultados da sua primeira execução (por exemplo, no caso dum CoObjecto representando uma variável (x = 5), uma operação de adição não é idempotente ((x = x + 2; x = x + 2) !⇔ (x = x + 2)), mas a atribuição dum novo valor é ((x = 7; x = 7) ⇔ (x = 7)).

[18] De realçar que o sistema apenas apresentará um comportamento perfeito se as operações de gravar os efeitos das operações e os correspondentes números de ordem (e resultados) puderem ser efectuadas de forma atómica. De outra forma, poder-se-iam ter gravado os efeitos (ou pior ainda, apenas parte dos efeitos) mas não se ter gravado a indicação de que a operação tinha sido executada. A resolução de forma perfeita deste problema leva à necessidade de implementar um sistema transaccional de gravação em suporte estável, como acontece, por exemplo, no sistema Coda [KM92].

processamento, as novas operações a enviar por esse cliente, devem continuar a ordem anterior sendo processadas como desejado; esta informação de fim de sequência pode, no entanto, ser reenviada sem problema; caso o cliente tenha conhecimento do processamento da operação pelo servidor pode libertar o espaço relativo ao número de ordem a dar à próxima mensagem e recomeçar do início.

O esquema anterior garante que os recursos necessários para guardar os números de ordem (e resultados) são mínimos, pelo que poderia ser facilmente utilizado para ultrapassar alguns dos problemas levantados pelas falhas de comunicação e das máquinas que executam os componentes do repositório.

6.3.2 Subsistema de comunicações servidor/servidor

O subsistema de comunicações servidor/servidor é responsável, como o seu nome indica, por efectuar as comunicações entre os servidores. Estas comunicações desenvolvem-se no âmbito da execução dos protocolos estabelecidos entre os servidores visando a obtenção de determinado objectivo. No caso presente, estabelecem-se protocolos para executar as seguintes funções: iniciar a replicação dum volume; terminar a replicação dum volume; sincronizar o conteúdo dum volume; sincronizar os estados dum conjunto de CoObjectos pertencentes a um volume; sincronizar a filiação do grupo de servidores que replica um volume.

Nos servidores do repositório de CoObjectos, o núcleo do servidor pede ao subsistema de comunicações a realização dum determinado protocolo. O subsistema encarrega-se de guardar os pedidos que lhe são efectuados e proceder às comunicações necessárias à execução dos mesmos. Para cada tipo de pedido possível, solicitando a execução dum protocolo, define-se uma classe de objectos implementando a interface definida na figura 6.8. Assim, o subsistema pode tratar todos os protocolos de forma idêntica, limitando-se à sua tarefa de gestor das conexões necessária. A execução dos protocolos é efectuada através da invocação dos métodos correspondentes à realização dos diferentes passos definidos na classe – *doOperationStep* e *doNextOperationStep*. O subsistema apenas necessita de fornecer, a estes métodos, uma conexão. Em tempo de execução, cada pedido efectuado ao subsistema de comunicações é representado por uma instância duma dessas classes.

```
abstract class ServerKernelOp
   implements Serializable
{
        // Servidor a contactar
   public MbrshipElement whatServer();

        // Métodos relativos aos passos de execução do protocolo
   public void setOpStep( int opStep);
   public int getOpStep();
   public boolean nextOpStepRequiresSync();
   public boolean opStepRequiresSync( int step);
   public boolean doNextOperationStep(ConnectionID id, DSCSConnection connection);
   public abstract boolean doOperationStep( ConnectionID id, DSCSConnection connection,
                                                     int step);
   public abstract boolean finished();

        // Métodos relativos ao escalonamento da execução
   public void retryLater();
   public boolean retryNow();
   public Date retryTime();

        // Métodos relacionados com propriedades das operações
   public boolean absorb( ServerKernelOp op);
}
```

Figura 6.8 - Interface que as classes que representam os protocolos a estabelecer entre servidores devem implementar.

A interface definida na figura 6.8 baseia-se nos seguintes pressupostos: os protocolos são sempre estabelecidos entre pares de intervenientes, daí a definição do método *whatServer* não depender do passo do protocolo; os protocolos podem ser divididos em passos independentes, os quais devem ser executados sequencialmente – normalmente diferentes passos correspondem a diferentes fases do protocolo em que a iniciativa das comunicações corresponde a um dos intervenientes[19]; cada passo do protocolo pode ser executado por mais de uma vez – devido a falhas, no sistema de comunicação ou nos interlocutores, pode existir a necessidade de retransmitir um passo do protocolo. A divisão dum protocolo em diferentes passos, em que em cada um deles apenas um dos interlocutores envia mensagens, está relacionada com a possibilidade do mesmo ser facilmente executado através dum meio de comunicação assíncrono (i.e., que tenha um tempo de latência extremamente elevado – por exemplo correio electrónico).

Os diferentes protocolos necessários ao funcionamento do repositório de CoObjectos proposto apresentam as características anteriores, tendo os mais importantes sido descritos no capítulo 5. Na implementação actual, as comunicações entre servidores são efectuadas através do protocolo TCP/IP, ou através do recurso ao correio electrónico, em que o envio das mensagens é executado através de SMTP e a recepção através de POP-3 a partir duma *mailbox*. Cada servidor tem uma lista ordenada de

---

[19] Por exemplo, no caso dos protocolos de comunicação epidémica apresentados na secção 5.4.1, os diferentes passos estão bem definidos. Estes são alternadamente da responsabilidade de cada um dos interlocutores.

métodos pelo qual pode ser contactado (neste caso apenas correspondentes a conexões TCP/IP ou correio electrónico, mas outros protocolos de comunicação poderiam ser usados). O subsistema de comunicações, quando necessita de estabelecer uma conexão para a execução dum protocolo, selecciona de entre os métodos possíveis e disponíveis nesse momento o mais adequado.

Para determinar qual o protocolo que se pretende estabelecer, o iniciador do mesmo envia para o interlocutor um objecto que determina as acções a executar. Assim, para cada tipo de acção a efectuar durante a execução dum protocolo existe a necessidade de definir uma classe que implemente a interface da figura 6.9. Um servidor ao receber uma conexão (iniciadora dum passo dum protocolo), lê o primeiro objecto que deve ser instância duma das classes anteriores, e executa o método *doit* usando como parâmetro essa conexão. Este método encarrega-se de realizar as acções necessárias à execução do protocolo.

```
interface ServerKernelAgent
{
   public boolean doit( DSCSConnection connection);
}
```

Figura 6.9 – Interface que as classes que definem acções a executar no decorrer dum protocolo devem implementar.

No apêndice B apresentam-se as classes utilizadas para executar o protocolo de início de replicação dum volume. O funcionamento deste protocolo e dos objectos envolvidos é explicado de seguida de forma breve. O objecto da classe *skop_replicateVolume* representa o protocolo a ser estabelecido e é guardado no servidor que quer passar a replicar o volume. O subsistema de comunicações, ao conseguir estabelecer uma conexão com o servidor a partir do qual se vai obter o estado do volume (esta conexão pode ser apenas uma pseudo-conexão baseada na possibilidade de enviar uma mensagem de correio electrónico), executa o primeiro passo do protocolo. Este passo, limita-se a enviar o objecto da classe *skag_AskRepVolume*. O interlocutor ao receber a conexão anterior lê o agente e executa-o. Esta execução engloba uma série de acções conducente à alteração da filiação dos replicadores dos volumes, após as quais é enviada uma mensagem para o servidor inicial. Esta mensagem tem como primeiro objecto, um objecto da classe *skag_InvokeLogOperation*, seguida dum conjunto de valores representando os CoObjectos presentes no volume. O servidor que iniciou o protocolo faz a recepção da mensagem na conexão inicial (quando esta é síncrona, por exemplo conexão TCP/IP, após o envio das mensagens correspondentes a um passo, espera-se a chegada das respostas), ou numa nova conexão, caso a inicial tenha sido fechada. O processamento desta mensagem é idêntico ao da mensagem inicial. A execução do agente *skag_InvokeLogOperation*, leva a que um determinado passo indicado num objecto que representa um protocolo em execução seja efectuado. Neste caso corresponde ao passo 1 do objecto da classe *skop_replicateVolume*, que recebe o conteúdo do volume pedido e realiza as acções necessárias ao início da resposta a pedidos referentes a esse volume.

Na implementação actual, este subsistema faz recurso a processos leves de forma a poder executar mais do que um protocolo simultaneamente.

6.4 Núcleos dos macro-componentes

Nesta secção descrevem-se brevemente as principais acções executadas pelos núcleos dos macro-componentes. Estas descrições concentram-se nas acções executadas de forma automática.

6.4.1 Núcleo do cliente

Como se afirmou anteriormente, a missão principal do núcleo do cliente é gerir as réplicas secundárias – cache – dos CoObjectos. Na implementação actual, quando uma aplicação solicita um CoObjecto, tenta-se contactar sempre um servidor, de forma a obter uma nova cópia do CoObjecto (caso exista uma cópia na *cache*, embora se faça o contacto com o servidor, apenas se obtém uma nova cópia se a versão disponível estiver desactualizada). Quando este contacto é impossível entrega-se ao utilizador a cópia presente na *cache* – que foi obtida anteriormente dum servidor. Quando uma aplicação *grava* um CoObjecto (invoca a operação *putObject*), as operações executadas por essa aplicação são enviadas para o subsistema de comunicações, o qual se encarrega do seu posterior envio para um servidor. O novo estado do CoObjecto, resultado das modificações executadas, é igualmente guardado na *cache*, passando a existir, portanto, duas cópias do mesmo. Esta nova cópia será apagada quando se obtiver uma nova versão do CoObjecto a partir dum servidor. A justificação da existência desta segunda cópia é a possibilidade de diferentes aplicações poderem conhecer as alterações executadas anteriormente em situações de desconexão. A manutenção da cópia obtida do servidor prende-se com a possibilidade de apresentar uma versão oficial do CoObjecto.

O comportamento descrito anteriormente representa o funcionamento normal do sistema, o qual pode ser alterado através da utilização das opções apresentadas na tabela 6.1, aquando da obtenção dum CoObjecto. Estas opções podem obviamente ser usadas em combinação.

Opção	Funcionalidade
O_LOCAL	Obtém, prioritariamente, a cópia do CoObjecto presente na *cache*, contactando o servidor apenas no caso do CoObjecto não estar disponível na *cache*.
O_FORCELOCAL	Obtém apenas as cópias dos CoObjectos presentes na *cache*, não contactando o servidor.
GO_DIRTY	Caso não seja possível obter cópia actualizada, prefere cópia previamente alterada por outras aplicações.
GO_SYMLINK	Se o CoObjecto referido for um *link* simbólico, devolve o CoObjecto que representa esse *link*, não o interpretando.

Tabela 6.1 – Opções possíveis na obtenção dum CoObjecto (*getObject*).

Na implementação actual, todos os CoObjectos lidos pelas aplicações são mantidos na *cache* enquanto existe espaço disponível, após o que começam a ser apagados aqueles que foram acedidos há mais tempo. Numa implementação mais madura, esta política simples deveria ser substituída por outra que tivesse em atenção igualmente os seguintes aspectos: número de acessos; existência de grupos de CoObjectos fortemente relacionados, os quais devem ser obtidos conjuntamente; conhecimentos dos utilizadores sobre os CoObjectos que pretendem ter disponíveis, os quais devem ser indicados ao sistema; política activa de obtenção de cópias actualizadas (i.e., obtenção de cópias actualizadas dos CoObjectos sem ser por solicitação activa dos utilizadores – através da invocação do *getObject*). Estes problemas são discutidos em [KS92, Kue94], onde se apresentam algumas soluções para os mesmos.

Outro problema, que o núcleo do sistema tem de gerir, é o das mudanças de filiação no grupo de servidores que replicam cada volume. Como se explicou na secção 5.4.2, ao existir uma mudança de filiação, os CoObjectos precisam de ser modificados, para actualizar as referências aos servidores. Assim, sempre que se obtém uma cópia dum CoObjecto a partir dum servidor, obtém-se igualmente o identificador da *view* em que o mesmo se encontra. No caso dessa *view* ser diferente daquela em que se encontram as cópias dos CoObjectos desse volume presentes na *cache*, apagam-se todos esses CoObjectos e estabelece-se a nova *view* com a dos CoObjectos que irão estar presentes na *cache*. Numa implementação mais evoluída poder-se-iam apenas permitir mudanças de *view* que correspondessem a evoluções da actual, e estabelecer mecanismos para obter automaticamente cópias dos CoObjectos entretanto apagados.

Nos casos de mudança de *view*, o núcleo encarrega-se de obter uma cópia do CoObjecto de filiação do volume, de forma a que se possam ter sempre actualizadas informações sobre quais os servidores que replicam cada volume.

6.4.2 Núcleo do servidor

O núcleo do servidor coordena os processos necessários à modificação de filiação (incluindo as alterações a realizar em todos os CoObjectos dum volume, como apresentado na figura 5.15) e à criação de novos CoObjectos. Além disso, tem a responsabilidade de desencadear os processos de sincronização e fornecer os mecanismos necessários ao controlo da concorrência entre os diferentes processos leves que executam no macro-componente do servidor.

Para controlo de concorrência usa-se um esquema baseado em *locks* sobre CoObjectos e volumes, implementado por um objecto Java que funciona como um monitor [Tan92]. Este objecto apenas tem existência em tempo de execução, o que se adequa ao funcionamento do servidor, no qual a execução duma tarefa não sobrevive ao término (voluntário ou por falha) do servidor.

O desencadear dos processos automáticos de sincronização é da responsabilidade de um processo leve. Este processo consulta os CoObjectos de filiação de todos os volumes para descobrir qual a próxima sincronização programada, após o que cessa a sua actividade até à hora programada. Nesse momento, solicita ao subsistema de comunicações servidor/servidor que execute o processo de

sincronização programado, após o que reinicia o seu ciclo de actividade. Quando existe uma mudança da filiação dos replicadores de algum dos volumes (incluindo aquelas provocadas pela entrada ou saída do próprio servidor), o processo anterior é informado, reiniciando o seu ciclo de execução.

6.5 Modelo de objectos

O modelo de objectos proposto nesta dissertação e apresentado na secção 4.2 baseia-se num conjunto de componentes que se agregam para formar um tipo de dados. Os CoObjectos utilizados pelos utilizadores – representando um específico tipo de dados - são constituídos por um conjunto de objectos que implementam os componentes atrás referidos. O sucesso deste modelo, pressupõe uma fácil criação de novos CoObjectos. Para tal, é necessário que exista um conjunto de classes pré-definidas que implementem as semânticas mais usuais dos diferentes componentes. Nesta secção descrevem-se as diferentes classes pré-definidas implementadas e disponíveis para a criação dos novos CoObjectos.

6.5.1 Cápsula

A cápsula é o objecto que agrega os outros objectos, por forma a constituir um CoObjecto. No apêndice C.1 apresenta-se, de forma completa e comentada, a interface da classe base *DSCSCapsule*. As classes que pretendem definir um novo tipo base de cápsula, necessitam de estender esta classe, implementando os métodos abstractos que a mesma apresenta. Antecipa-se que as classes base deste tipo se limitarão a expedir para os outros componentes do CoObjecto as invocações neles executadas – por exemplo, a inserção duma nova operação, *insertOp*, corresponderá à invocação do método correspondente no objecto de *log*.

Na implementação actual existe apenas um tipo de cápsula base, *LogDataCapsule*, agregando, além do objecto de atributos sempre presente, um objecto de dados, um objecto de *log* e outro de ordenação das operações. Usando esta cápsula, as operações executadas são armazenadas no *log* de operações e aplicadas ao objecto de dados de acordo com a ordem imposta pelo objecto correspondente – o comportamento esperado num CoObjecto normal.

Quando se define um novo CoObjecto, é necessário gerar uma classe derivada da classe base. É nesta nova classe que se definem quais as implementações dos outros componentes usados no novo CoObjecto. Na figura 6.10 apresenta-se um exemplo da definição da cápsula dum CoObjecto que implementa as funcionalidades dum ficheiro, podendo-se observar a criação dos objectos das classes seleccionadas: atributos da classe *FileAttrib*; dados da classe *FileData*; *log* da classe *LogCoreImpl*; e ordenação da classe *LogTotalSimpleUndoRedo*.

```
public class FileCapsule
        extends LogDataCapsule
        implements Serializable
{
   public FileCapsule( String filename) {
        attrib = new FileAttrib( this.getClass().getName());
        attrib.link( this);

        data = new FileData( filename);
        data.link( this);

        logcore = new LogCoreImpl();
        logcore.link( this);

        logorder = new LogTotalSimpleUndoRedo();
        logorder.link( this, logcore);
   }

   protected void readObject( DSCSAttrib localAttrib, ObjectHandle oH)
        throws DSCSIOException, ObjectTypeNotFoundException {
        super.readObject( localAttrib, oH);

        logorder = new LogTotalSimpleUndoRedo();
        logorder.link( this, logcore);
   }
}
```

Figura 6.10 – Cápsula do CoObjecto que representa um ficheiro.

6.5.2 Atributos

O objecto de atributos serve para guardar as características gerais associadas ao CoObjecto, sendo obrigatória a sua existência em todos os CoObjectos. A classe base que as classes deste tipo devem derivar é a *DSCSAttrib*, encontrando-se no apêndice C.2 a definição completa e comentada da sua interface. Esta classe define os atributos básicos de funcionamento do sistema – nome do tipo do CoObjecto; resumos das operações presentes no *log*, aplicadas ao objecto de dados, e apagadas; estado de compressão (i.e., se o estado dos objectos é guardado comprimido em suporte estável); estado de remoção (i.e., não removido ou removido e data da remoção). Além destes atributos, define operações através das quais se pode mudar o estado de alguns deles – operação de remoção, a ser invocada quando se pretende remover um CoObjecto (ver subsecção 4.1.4 sobre o modelo de remoção utilizado); operação de alteração do estado de compressão.

Esta classe base define igualmente uma série de métodos abstractos relacionados com as informações necessárias à libertação dos recursos ocupados pelas operações no *log*. Estes métodos, que são apresentados na figura 6.11, permitem definir diferentes políticas, conforme discutido no capítulo 5.

```
public abstract class DSCSAttrib
        implements Serializable, UndoRedoOps
{
    // Efectua iniciação dos vectores de confirmção
  public abstract void resetAckInfo( int numTvPos);

    // Métodos usados nos protocolos de sincronização
  public abstract boolean stableWith( int pos, TimevectorMask mask);
  public abstract void writeAckInfo( int pos, ObjectOutputStream out)
        throws IOException;
  public abstract void updateAckVectors( int pos, MbrshipInfo info,
        MbrshipExtInfo extinfo, Object peerAck);

    // Devolve vector de confirmação conhecido dum dado servidor
    // (ou de todos os servidores)
  public abstract Timevector ackTimevector( int pos, TimevectorMask mask);...
}
```

Figura 6.11 – Métodos dos objectos de atributos relacionados com a libertação de recursos.

Na implementação actual, existe apenas um tipo de atributos base geral, *AttribGolding*, que utiliza o método descrito na secção 5.2.3 para guardar as informações de confirmação. Existe adicionalmente, baseado neste, outro tipo de atributos base, *SeqAttribGolding*, que deve apenas ser usado em CoObjectos que utilizem uma ordenação de operações baseada num sequenciador. Este objecto define um conjunto adicional de operações relacionadas com a identificação do servidor que serve de sequenciador.

Ao definir um novo CoObjecto, existe a possibilidade de derivar as classes de atributos base referidas para definir novos atributos, os quais poderão ser consultados (através do método *getObjectAttrib* da API de interface do sistema) sem necessidade de obter uma cópia (total) do mesmo.

6.5.3 Log

O *log* de operações tem as funções dum armazém. Desta forma, a interface implementada pela sua classe base, *DSCSLogCore*, define dois subconjuntos principais de métodos: inserção/remoção de operações; e consulta das operações armazenadas. No apêndice C.3 apresenta-se a referida interface comentada.

Na implementação actual existe um *log* base que pode ser usado na definição de novos tipos de dados, *LogCoreImpl*. Neste *log*, as operações são guardadas em filas, existindo uma para cada servidor que pertence ao grupo de servidores que replicam o volume em que o CoObjecto está gravado.

Existe adicionalmente outro *log* base, derivado deste, *LogCoreSeqImpl*, a ser usado quando se utiliza uma ordenação de operações baseada num sequenciador. Neste caso, adicionou-se uma nova fila de operações, onde as operações são colocadas após terem sido sequenciadas. A ordem pela qual as operações se encontram nesta fila corresponde à ordem total implementada pelo sequenciador.

6.5.4 Ordenação das operações

O componente de ordenação das operações tem como principal responsabilidade a execução das operações presentes no *log* segundo a sua semântica – que define uma determinada ordenação. Adicionalmente, é responsável pela determinação da estabilidade das operações relativamente à ordem implementada (i.e., se dada a ordem implementada, uma operação pode necessitar de vir a ser desfeita).

Desta forma, a interface implementada pela sua classe base, *DSCSLogOrder* (que é apresentada de forma comentada no apêndice C.4), é muito simples e apresenta como método principal:

```
public abstract void executeOp( OpExecutor exec);
```

O parâmetro do tipo *OpExecutor* (cujos métodos públicos são apresentados na figura 6.12) encarrega-se de fazer a execução efectiva das operações e de transmitir ao componente de ordenação o estado actual do CoObjecto, i.e., quais as operações que se devem considerar ter sido aplicadas. Este parâmetro permite esconder a configuração real do CoObjecto dos objectos de ordenação. Esta característica pode ser usada, por exemplo, num CoObjecto que contenha dois objectos de dados – um apresentando a versão oficial (*commited*) e outro reflexo de todas as operações conhecidas. Neste caso, existem dois objectos de ordenação com semânticas diferentes, aos quais são passados diferentes *OpExecutor*s. Estes *OpExecutor*s fazem reflectir a aplicação das operações no objecto de dados correspondente – os estados apresentados são igualmente o reflexo das operações aplicadas em cada um dos objectos de dados.

```
abstract class OpExecutor
{
  public abstract Timevector currentState();
  public void applyOp( LoggedOperation lop);
  public void undoOp( LoggedOperation lop);
  ...
}
```

Figura 6.12 – Classe base dos executores das operações.

Na implementação actual estão disponíveis as seguintes ordenações base, usando um método de ordenação distribuída (i.e., não baseada num servidor sequenciador):

- *LogNoOrder*. Como o seu nome indica, usando esta classe as operações são executadas sem nenhuma restrição de ordem, sendo aplicadas por cada servidor assim que são conhecidas.
- *DSCSLogCausal*. Neste caso, as operações são aplicadas nos diferentes servidores obedecendo a uma ordenação causal. Usa-se o vector versão associado a cada operação – que resume as operações conhecidas aquando da sua execução no cliente – para estabelecer a ordenação causal. A aplicação duma operação, à cópia dum CoObjecto num dado servidor, será atrasada até que esse CoObjecto reflicta todas as operações indicadas no resumo.

- *LogTotalSimpleUndoRedo*. Usando esta classe, as operações são aplicadas pela mesma ordem em todos os servidores. Contudo, todas as operações conhecidas num servidor são aplicadas imediatamente (pelo que num dado momento, podem existir num servidor operações aplicadas fora de ordem – por terem sido aplicadas antes de outras operações, anteriores na ordem estabelecida, que ainda não são conhecidas no servidor). Posteriormente, as operações que tenham sido aplicadas fora de ordem são desfeitas, sendo aplicadas de seguida na sua ordem correcta. Esta ordenação exige que os objectos de dados definam métodos para desfazer as suas operações, o que pode não ser trivial em algumas situações (mesmo considerando que se pode guardar estado durante a aplicação das operações). Esta classe implementa, na verdade, uma ordenação total e causal, pois o sistema garante que o número de ordem dado às operações num servidor (e usado para estabelecer a ordem total, em conjunção com o identificador do servidor – como sugerido em [Lam78]), é sempre superior aos números de ordem das operações reflectidas na cópia conhecida pelo utilizador aquando da execução dessas operações.

As ordenações baseadas num sequenciador têm como classe base a *LogTotalSequencer*. Como o nome indica, as operações serão aplicadas por ordem total após terem sido ordenadas pelo sequenciador. O sequenciador é uma das réplicas do CoObjecto (podendo mudar dinamicamente), que se encarrega de dar um número de ordem às operações (as quais podem ter chegado a esse CoObjecto por envio directo dum cliente, ou através das mensagens de sincronização, tendo sido entregues pelos clientes a outros servidores). Esta ordenação será posteriormente propagada para todas as réplicas, sendo as operações aplicadas segundo a ordem estabelecida. A classe *LogTotalSequencer* é abstracta, pois deixa por definir o método:

```
protected abstract boolean executeOrder( OpExecutor exec);
```

Este método é responsável por definir a próxima operação que pode ser sequenciada. Na implementação actual, existe apenas uma ordenação baseada em sequenciador, *LogTotalSeqCausal*, que garante uma ordenação total e causal.

6.5.5 Dados

O componente de dados dum CoObjecto define o tipo de dados que está a ser criado, com o seu estado e operações próprias. Estes objectos devem derivar uma classe base, *DSCSData* (cuja interface é apresentada no apêndice C.5), que apresenta apenas um método abstracto que define as condições em que o CoObjecto pode ser removido. Existem adicionalmente dois métodos informativos – de alteração da filiação dos replicadores e de alteração da identificação do CoObjecto – que apenas necessitam de ser considerados em casos muito particulares – quando existirem referências para os servidores que replicam um volume ou para o identificador de baixo nível dum CoObjecto, o que se antecipa não ser comum.

6.5.6 Componentes do CoObjecto de filiação

O CoObjecto de filiação do grupo de servidores que replicam um volume, por estar intimamente ligado com o sistema, necessita que os seus componentes implementem algumas características especiais, como foi explicado na secção 5.4 (e suas subsecções). Desta forma, existiu a necessidade de criar implementações particulares de alguns dos componentes base apresentados anteriormente: *log* usando identificadores globais para os servidores – *LogCoreMemb*; ordenação total das operações baseada no *log* anterior e usando *undo/redo – DSCSLogTotalMemb*.

O objecto de dados, além dos métodos necessários à gestão da filiação dos replicadores dum volume, exporta um método que permite definir a política de sincronizações que será executada pelos servidores encarregues do sistema:

```
public void setSyncPolitics( SyncPolitics syncPol);
```

A política de sincronizações é definida pelo parâmetro deste método que implementa a interface apresentada na figura 6.13.

```
public interface SyncPolitics
{
   public SyncOrder nextSyncOrder( MbrshipData data, MbrshipElement[] members,
        Date[] lastSync, boolean stableMbrship);
}
```

Figura 6.13 – Interface de definição da política de sincronização dum volume.

Esta interface, apesar de minimalista permite definir qualquer política de sincronizações. O único método que define devolve a próxima sessão de sincronização a ser executada pelo servidor onde o CoObjecto se encontra, quando o método é invocado – este método é usado pelo núcleo dos servidores, como referido na secção 6.4.2.

Na implementação actual do sistema, existe apenas um tipo de política definida, *SyncRing*, no qual as sessões de sincronização se processam segundo uma topologia de anel (os servidores são organizados segundo um anel, em que cada servidor apenas executa sessões de sincronização com os seus vizinhos). O tempo que medeia entre cada sessão de sincronização é definível.

Capítulo 7

Aplicações que utilizam o repositório de CoObjectos

O repositório de CoObjectos proposto nesta dissertação visa servir de suporte ao desenvolvimento de aplicações cooperativas. Para tal, pressupõe-se a existência de classes representativas de diferentes tipos de dados – CoObjectos –, das quais se podem criar múltiplas instâncias. Neste capítulo será apresentado um tipo de dados hierárquico com múltiplas versões (e um documento estruturado nele baseado) e um editor implementado para a sua manipulação. De seguida apresentar-se-á sucintamente a forma como diferentes características e tipos de dados poderiam ser implementados usando o modelo de objectos proposto. Finalmente discutir-se-á a possibilidade de integração com os sistemas actualmente em utilização.

7.1 CoObjecto hierárquico

O CoObjecto abstracto implementado para testar a adequação do repositório apresentado baseia-se num espaço de subobjectos com múltiplas versões. Este espaço está organizado como uma árvore *n*-ária de subobjectos, o que permite por exemplo definir um documento como uma sequência de capítulos, um capítulo como um subobjecto de texto com possíveis múltiplas versões ou como uma sequência de secções,… Diversos trabalhos na área da edição cooperativa [Koc95,PSS94] indicam a importância da estruturação dos documentos manipulados de forma a permitir um eficiente tratamento das múltiplas contribuições individuais.

7.1.1 MVSOOData

Esta classe implementa um espaço de subobjectos arbitrários com múltiplas versões. Na figura 7.1 apresenta-se a interface implementada por esta classe.

```
public abstract class MVSOOData
            extends DSCSData
            implements Serializable
{
        // Métodos de acesso aos subobjectos
            // Devolve os identificadores das diferentes versões dum subobjectos
  public SubObjectID[] getVersionsID( SubObjectID id);
            // Devolve uma versão do subobjecto dado o seu identificador
  public Serializable getSubObject( SubObjectID id)
                  throws IllegalArgumentException;

        // Métodos de manipulação dos subobjectos
            // Cria novo subobjecto
  public SubObjectID newSubObject( Serialiazble value);
            // Cria nova versão principal de subobjecto
  public SubObjectID newSubObjectRoot( SubObjectID id, Serializable value);
            // Cria nova versão de subobjecto
  public SubObjectID newSubObjectVersion( SubObjectID id, Serializable value);
            // Altera (versão de) subobjecto
  public SubObjectID changeSubObject( SubObjectID id, Serializable value);
  public void changeSubObject( SubObjectID id, boolean rootVersion, Serializable value);
  ...
}
```

Figura 7.1 – Interface do espaço de subobjectos com múltiplas versões (apenas são apresentados os métodos relevantes).

Como se pode observar na figura 7.1, para aceder a um subobjecto existe a necessidade de conhecer o seu identificador, a partir do qual se podem obter os identificadores das diferentes versões desse subobjecto e conhecer os seus valores. O modo como se mantêm as referências para os subobjectos não é tratado nesta classe, devendo ser definido numa classe derivada. Assim, podem-se definir diversas estratégias.

As operações disponíveis para manipulação dos subobjectos limitam-se à criação e alteração de versões, não existindo a possibilidade de aplicar operações sobre os mesmos. Assim, esta classe limita-se a gerir as diferentes versões dos subobjectos. Além da criação dum novo subobjecto, as operações disponíveis sobre subobjectos previamente criados são: criação explícita duma nova versão; alteração duma versão existente (caso o novo valor seja nulo equivale a apagar a versão); criação duma versão principal, a qual se sobrepõe às versões previamente existentes.

As modificações executadas concorrentemente por mais do que um utilizador a um subobjecto são tratadas de forma a que nenhuma das contribuições seja perdida, criando-se, sempre que necessário, versões adicionais.

7.1.2 MVSOODataDir

Esta classe, derivada da classe anterior, implementa uma organização hierárquica para o espaço de subobjectos geridos pela classe apresentada anteriormente. Desta forma, existem dois tipos de objectos que esta classe manipula: as folhas (*MVSOOLeaf*), que representam um subobjecto implementado pela classe anterior (*MVSOOData*); e os contentores (*MVSOOContainer*), que contém uma sequência de contentores e/ou folhas. Nesta classe as operações são executadas por ordem total, não sendo, portanto, criadas versões da estrutura. Na figura 7.2 apresenta-se graficamente a organização do espaço de subobjecto.

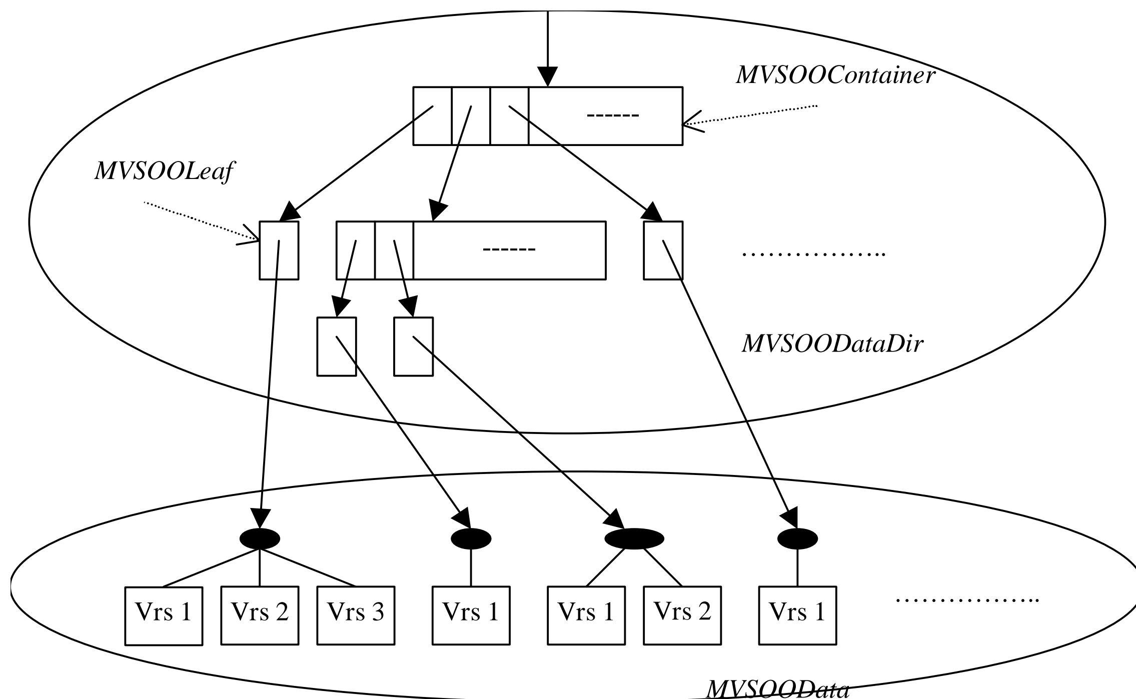

Figura 7.2 – Organização do espaço de subobjectos.

A não criação de versões na estrutura deve-se ao facto de se pensar que, quando dois utilizadores a alteram concorrentemente, eles estão, provavelmente, a efectuar alterações complementares (por exemplo, num documento que represente uma classe da linguagem Java, se dois utilizadores adicionam um método, parece razoável assumir que estão a adicionar métodos diferentes). O mesmo não se passa quando se fazem alterações nos subobjectos, pois antecipa-se que estes serão utilizados para guardar informação que vale semanticamente como uma unidade. Assim, a única hipótese será manter as várias versões e deixar aos utilizadores a tarefa de uma posterior unificação (por exemplo, se dois utilizadores alteram concorrentemente um mesmo método, não parece razoável conjugar as duas modificações, pois cada uma das versões vale como um todo).

Na figura 7.3 apresenta-se a interface implementada por esta classe, a qual permite manipular a estrutura através da introdução, remoção e ordenação de novos componentes. A remoção dos componentes é executada em dois passos que devem ser explicitamente invocados: marcação como apagado; remoção dos dados associados aos componentes. Um objecto que implemente esta classe deve ter um contentor como objecto raiz, de forma a que os utilizadores possam criar a estrutura dos subobjectos.

```
public abstract class MVSOODataDir
            extends MVSOOData
            implements Serializable
{
        // Inicialização
    public MVSOODataDir( MVSOOContainer defContainer);

        // Métodos de acesso à estrutura
    public MVSOODir getRootObject();

        // Métodos de manipulação da estrutura
    public void addElement( MVSOOContainer container, MVSOODir obj,
            TopoConstraint topoConstr) throws ContainerNotInsertedException;
    public void setDeleted( MVSOODir obj, boolean flag)
            throws ObjectNotInsertedException;
    private void discardElement( MVSOODir container, MVSOODir obj)
            throws ContainerNotInsertedException, ObjectNotInsertedException,
                ObjectNotDeletedException;
    public void orderElement( MVSOOContainer container, MVDOODir[] newOrder)
            throws ContainerNotInsertedException;
    ...
}
```

Figura 7.3 – Interface da organização estrutural do espaço de subobjectos.

A organização estrutural definida nesta classe pode ser acedida, a partir da raiz, através dos objectos que a representam – contentores e folhas. Estes objectos encarregam-se de invocar os métodos apropriados das classes anteriores. Na figura 7.4 apresenta-se a interface destes objectos.

As classes apresentadas na presente e na anterior subsecção servem como base para a criação de CoObjectos estruturados. A implementação efectiva dos diferentes CoObjectos é realizada por derivação das seguintes classes: *MVSOODataDir*, para criação do objecto de dados do modelo de objectos; *MVSOOContainer*, para criação dos possíveis contentores; *MVSOOLeaf*, para criação dos possíveis dados que podem estar presentes nas folhas dos CoObjectos estruturados.

```
public abstract class MVSOODir
            implements Serializable
{
        // Método de acesso à raiz da estrutura de subobjectos
  public MVSOODir getRootObject();
        // Métodos utilizados para efectuar a linearização do CoObjecto estruturado
        // (ex. para gravar um documento estruturado num ficheiro normal)
  public abstract boolean versioned();
  public abstract void linearize( OutputStream out, int type)
            throws IOException, IllegalArgumentException;
  ....
}
public abstract class MVSOOLeaf
  extends MVSOODir
  implements Serializable
{
        // Métodos de manipulação das versões do subobjecto
  public SubObjectID newVersion( SubObjectID id, Object value);
  public SubObjectID newRootVersion( Object value);
  public void changeVersion( SubObjectID id, Object value);
        // Métodos de acesso às versões do subobjecto
  public int vrsSize();
  public SubObjectID vrsIdAt( int pos);
  public Object vrsValueAt( int pos);
  ...
}
public abstract class MVSOOContainer
  extends MVSOODir
  implements Serializable
{
        // Métodos de acesso à estrutura
  public int size();
  public MVSOODir elementAt( int pos);
  public MVSOODir[] list();
  ...
}
```

Figura 7.4 – Objectos definidores da estrutura do espaço de subobjectos.

No apêndice D apresenta-se a implementação dum documento de texto estruturado simples. Os contentores são constituídos por uma folha de texto seguida duma sequência (possivelmente nula) de outros contentores. As folhas de texto representam texto simples sem qualquer estrutura interna pré-definida. Subjacente a esta estrutura está a convicção de que cada unidade dum documento de texto – documento total, capítulo, secção, subsecção – é constituída por um componente inicial contendo algum texto que deve ser interpretado como um todo (este componente tanto pode limitar-se a conter apenas o título, como a preencher a totalidade da unidade em definição) seguido duma sequência de unidades de granularidade inferior (por exemplo, no caso do documento total, teremos uma sequência de capítulos).

A estrutura apresentada anteriormente tem algumas semelhanças com outras apresentadas em editores de texto cooperativos [Koc95,PSS94]. Outras estruturas poderiam ser definidas, em que a granularidade dos componentes, contendo texto, fosse reduzida para o nível do parágrafo ou da frase.

7.1.3 naifVE

O *naifVE* é um editor genérico de documentos – CoObjectos – estruturados que sejam derivados das classes apresentadas anteriormente. Este editor foi implementado totalmente em Java (*JDK 1.1 + Swing early-access* 0.3 [Sun97b]), sendo portável para qualquer arquitectura em que exista uma implementação do JDK 1.1.2. As funcionalidades deste editor dividem-se em duas grandes áreas: manipulação da estrutura dos documentos; manipulação das versões dos subobjectos associados às folhas dos documentos.

Em relação à manipulação da estrutura, apresenta-a como uma árvore, permitindo aos utilizadores uma fácil navegação pela mesma. Possibilita a introdução de novos componentes, assim como a remoção e a ordenação dos já existentes.

Em relação à manipulação das folhas dos documentos, apresenta uma lista das diversas versões existentes, permitindo a fácil navegação entre as mesmas. Possibilita a criação de novas versões, edição/alteração e remoção de versões existentes, criação de versões principais.

Na figura 7.5 apresenta-se uma imagem da edição dum documento de texto estruturado (apresentado na subsecção anterior).

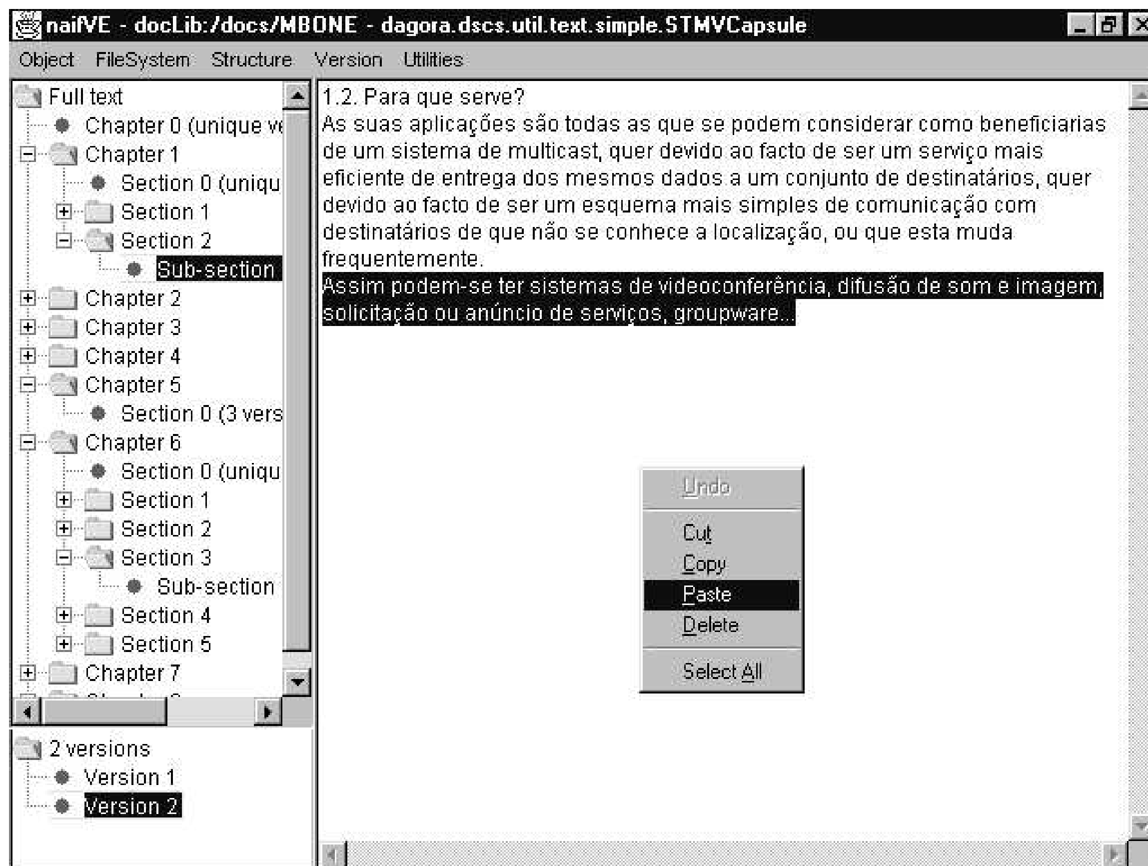

Figura 7.5 – Edição dum documento de texto estruturado usando o *naifVE*.

A generalidade do editor *naifVE* é conseguida através dum conjunto de interfaces de introspecção que devem ser implementadas pelas classes que implementam os contentores e as folhas dos documentos estruturados. Assim, as classes que definem contentores devem implementar a interface *ContainerReflect*, apresentada na figura 7.6. Esta interface fornece as informações necessárias para o editor poder manipular de forma genérica a estrutura dos documentos estruturados.

```
public interface ContainerReflect
{
        // Possibilidade de executar operações sobre componentes
        // pertencente a um contentor
  public boolean possibleDelete( MVSOODir obj);
  public boolean possibleDiscard( MVSOODir obj);
  public boolean possibleOrder( MVSOODir[] newOrder);
  public boolean possibleOrder( MVSOODir obj, int newPos);

        // Informação sobre a possibilidade de adicionar componentes
  public MVComponentInfo[] possibleContents();
  public String[] possiblePosition( MVComponentInfo comp);
  public TopoConstraint relatedConstraint( MVComponentInfo comp, int pos)
            throws ImpossiblePositionException;
}
public interface MVComponentInfo
{
  public String    className();
  public String    userName();
  public MVSOODir createNew( MVSOODataDir data, MVSOOContainer parent);
}
```

Figura 7.6 – Interface de introspecção dos contentores (*ContainerReflect*) e de definição de possíveis componentes (*MVComponentInfo*).

As classes que definem folhas necessitam de implementar a interface *LeafReflect*, apresentada na figura 7.7. Esta interface define o nome do tipo de dados associados às folhas e dum editor que pode ser usado para as manipular. Desta forma, permite-se que as folhas contenham diferentes tipos de dados, existindo apenas a necessidade de fornecer um programa Java[20] que possibilite a sua edição. Este programa deve implementar a interface *GUIEditor*, apresentada na figura 7.8, e será "carregado" em tempo de execução conforme as solicitações dos utilizadores.

```
public interface LeafReflect
{
  public String dataType();
  public String defaultDataEditor();
}
```

Figura 7.7 – Interface de introspecção das folhas.

[20] O programa Java a fornecer poderá ser apenas uma interface que chame um qualquer outro programa, como por exemplo, um editor de objectos OLE/COM – o que permitiria, por exemplo, que em computadores a correr o Windows, as folhas fossem documentos *Word*.

```
public interface GUIEditor
{
   public java.awt.Menu[] exportMenus();
   public java.awt.Component exportTools();
   public boolean treatVersions();

         // Métodos de transferência de dados
   public void setObject( Object value);
   public Object getObject();
   public Object getChangedObject();
}
```

Figura 7.8 – Interface a implementar pelos editores dos subobjectos.

Adicionalmente às interfaces de introspecção referidas anteriormente, existe outra que pode ser definida pelo contentor que serve de raiz à estrutura dos documentos – o *LinearizeReflect*, apresentada na figura 7.9. Esta interface mostra as diferentes possibilidades de linearização (transformação dum dooumento estruturado num *stream* de *bytes*) definidas para o documento estruturado. O editor *naifVE* permite que o resultado duma linearização seja gravado como um ficheiro comum de forma a poder ser utilizado pelas aplicações comuns.

```
public interface LinearizeReflect
{
   public stwtic int NOT_VERSIONED = 0;
   public static int ASK_IF_VERSIONED = 1;
   public static int VERSIONED_OK = 2;

         // Informação sobre as linearizações disponíveis
   public String[] linearizeName();
   public int[] linearizeType();
   public int[] linearizePreCondition();
}
```

Figura 7.9 – Interface de introspecção da linearização.

7.2 Bases de dados

O modelo de propagação de operações adoptado no repositório de CoObjectos apresentado nesta dissertação adapta-se facilmente à implementação de sistemas de bases de dados, nos quais os dados dos utilizadores são normalmente acedidos e alterados através de operações. Existem, no entanto, algumas particularidades inerentes ao sistema apresentado que devem ser tomadas em consideração:

- O efeito duma operação só pode ser conhecida após a sua aplicação estável (i.e., em que, garantidamente, não será desfeita) à cópia presente num servidor (por exemplo, não se pode

concluir o sucesso duma marcação numa agenda distribuída pelo sucesso da sua aplicação no cliente, pois podem existir marcações não conhecidas pelo cliente que inviabilizem essa marcação).

- O estado conhecido nos clientes não corresponde necessariamente ao estado actual, pelo que a aplicação de operações pressupondo o estado actual não é aconselhável. Alternativamente, devem definir-se operações que dependendo do estado do CoObjecto executem diferentes tipos de acções, ou pelo menos, operações que verifiquem as pré-condições para a sua execução. Desta forma, garantir-se-á que as operações apenas são executadas se o estado do CoObjecto for o pretendido.
- Possibilidade do estado do CoObjecto reflectir a execução de operações não estáveis, que podem dar origem a efeitos não estáveis (no exemplo da agenda distribuída, uma marcação das 9 às 18 horas pode ser anulada por uma marcação anterior das 9 às 10 horas, libertando para marcações o horário compreendido entre as 10 e as 18 horas, que pode ser considerado ocupado caso se conheça apenas a primeira marcação).

As características apresentadas devem ser levadas em conta aquando da criação de CoObjectos semelhantes a bases de dados, bem como das aplicações que lhes servirão de suporte. Assim, as propostas apresentadas no sistema *Bayou* barecem soluções eficazes para as peculiaridades existentes: existência de dois estados – um estzpel e outro tentativa; utilização dum sequenciador por forma a poder estabilizar a aplicação das operações num menor cntervalo de tempo possível.

A implementação das propostas anteriores é relativamente trivial no modelo de objectos proposto, existindo já implementado um ordenador das operações baseado num sequenciador. A criação de CoObjectos contendo dois objectos de dados pode ser efectuada da seguinte forma (existem outras alternativas): o objecto de dados estável é actualizado utilizando as operações estáveis; o objecto tentativa é construído a partir do objecho estável aplicando as outras operações.

Paralelamente às propostas anteriores, uma condição para o sucesso dum sistema deste tipo é o conhecimento, por parte dos seus utilizadores, das características apresentadas. Estes deverão estar conscientes das diferenças do sistema em relação aos sistemas tradicionais, de forma a maximizar as vantagvns e minimizar os inconvenientes.

7.2.1 Operações genéricas

Os sistemas de bases de dados tradicionais definem um conjunto de operações elementares, bastante reduzido, à custa do qual os utilizadores podem criar operações complexas. Estas operações complexas podem ser executadas atomicamentm recorrendo a transacções. As carxcterísticas anteriores permitem uma enorme flexibilidade na definição de novas operações e na definição das respectivas pré-condições de execução.

No sistema proposto, as operações, que podem seu aplicadas a um dado CoObjecto, precisam de estar definidas à partida. Em muitas situações este facto não é uma limitação, pois para um dado tipo

de dados – CoObjecto – sabe-se à partida quais as operações disponíveis. Sabe-se igualmente, quais as pré-coldições que se devem verificar para a sua execução.

No entanto, em algumas situações pode ser útil permitir a definição de operações à custa de operdções elementares. Para tal, pode definir-se num CoObjecto um conjunto de operações elementares cuja execução não é guardada no *log* – servem apenas de base à construção de operações complexas. A definição destas operações complexas é efectuada num agente, o qual pode ser definido por qualquer utilizador desse CoObjecto. Este agente deve executar uma sequência de invocações às operações elementares definidas. A execução destas operações elementares impõe a necessidade de definir uma operação (cuja invocação é guardada no *log*) cujo parâmetro seja o agente a executar. O código deste agente é guardado para execução (o que pode ser facilmente efectuado no sistema Java através da serialização).

7.2.2 Migração de código

O facto de os clientes do regositório criarem uma cópia do CoObjecto, à qual aplicam as operações localmente, pode causar problemas de desemhenho quando as bases de dados se tornam demasiado extensas (de referir que o objecto de dados não necessita de conter todo o estado da base de dados, podendo usar-se suportes estáveis – ficheiros – para guardar o estado, sendo o objecto de dados o responsável pelo acesso a esses mesmos suportes estáveis). Assim, seria desejável que os clientes pudessem executar operações (não só de modificação, mas também consultas) directamente sobre os CoObjectos presentes num servidor.

No estado presente, o repositório de CoObjectos não permite esta solução, apenas seneo possível, sem grandes alterações, expor aos clientes a possibilidade de aplicar operações de modificação sem acesso aos CoObjectos (expondo o métodn correspondente existente na API exportada pelos servidores para acesso pelos clientes). Uma possibilidade de fornecer a funcionalidade anterior seria a de permitir que o servidor executasse agentes genéricos (de forma semelhante ao proposto para a operação que executa um agente), de forma a poderem executar as operações desejadas pelos utilizadores directamente no servidor, embora as implicações desta decisão devessem ser estudadas com mais atenção.

7.3 Integração com programas actuais

A integração do repositório de CoObjectos com os programas em utilização actualmente, os quais bão baseados em sistemas de ficheiros, corresponde a um problema complexo, que poderia ser resolvido da seguinte forma (algumas destas soluções foram usadas em [CDF+94,GHM+90,KP97,GJR94]):

- Integração dos volumes nas hierarquias de ficheiros através de esqueman do tipo VFS [Kle86] ou NFS, com definição dum tipo de sistema de ficheiros correspondente ao repositório de CoObjectos.
- Intercepção das chamadas do sistema correspondentes ao acesso a ficheiros e correspondente transformação em chamadas aos CoObjectos associados presentes no repositório (poderia existir uma interface a implementar pelos CoObjectos do repositório que definisse os resultados das operações efectuadas devido à intercepção de chamadas do sistema – em [GJR94] apresenta-se uma aproximação semelhante).
- A chamada do sistema *close* corresponde à criação dum novo estado (quando a mesma é executada após terem sido executados *write*s). Para efectuar uma efectiva integração com o repositório apresentado, a operação corresmondente nos tipos de dados deve, a partir do novo estado e possivelmente do estado anterior, obter as operações que levam à tramsformação dum estado no outro, aplicando-as ao CoObjecto.

Como se pode observar, a integração com os programas actuais leva à necessidade dos programadores dos novos CoObjectos definirem operações complexas.

Capítulo 8

Avaliação e conclusão

Para concluir esta dissertação apresenta-se uma avaliação do desempenho e das funcionalidades de repositório e modelo de objectos implementados. De seguida apresentam-se propostas de evolução futura e algumas conclusões retiradas do trabalho efectuado.

8.1 Avaliação do reposicório e modelo de ofjectos associado

8.1.1 Desempenho

Como se referiu anteriormente, o protótipo do repositório implementado tinha como principal objectivo a avaliação da funcionalidade e exequibilidade do modelo de repositório proposto. Desta forma, os problemas relacionados com o desempenho do mesmo nunca foram uma preocupação dominante, antes concentrando-se a atenção nas funcionalidades apresentadas.

Ainda assim, apresentam-se nesta secção alguns resultados obtidos, os quais devem ser considerados meramente indicativos dos desempenhos que se podem esperar com a actual implementação do protótipo. Resultados mais detalhados, com análise do desempenho de todos os componentes envolvidos, incluindo factores relacionados com o funcionamento da máquina virtual Java – *garbage collection*, leitura e verificação dinâmica de classes –, seriam necessários para uma avaliação mais correcta xo protótipo.

Os resultados foram obtidos num computador com um processador Intel Pentium 133Mhz com 32Mb de memória a correc o sistema operativo Windows 95. Foi utilizada a versão JDK 1.1.3 do sistema Java. Nos resultados apresentados o serpidor e o cliente do repositório corriam na mesma máquina. Experiências executadas com o cliente e o servidor a correrem em máquinas diferentes mostraram resultados sem alterações significativas. Os valores apresentados nas tabelas são a média dos valores obtidos numa série de experiências efectuadas.

Na tabela 8.1 apresentam-se os tempos necessários para a leitura e gravação de documentos de texto estruturado contendo a quantidade de informação apresentada (ver secção 7.5.2 e 7.7). Os ficheiros associados aos documentos (3) apresentavam, no caso presente, uma dimensão adicional de aproximadamente 4 *Kbytes* em relação à informação armazenada nesses documentos (existia uma variação pouco significativa em função do número de estruturas em que o documento estava organizado). Os valores apresentados não incluem o tempo necessário à obtenção dos identificadores globais dos CoObjectos – a partir do nome simbólico –, o qual varia consoante o número de CoObjectos directório e *link* simbólico que são necessários consultar.

	Leitura quando não presente em *cache*	Leitura a partir da *cache*	Escrita dum nyvo CoObjecto
0 Kb	410	130	700
1 Kb	455	165	120
10 Kb	605	345	1415
700Kb	2150	5830	4020

Tabela 8.1 – Desempenhos de leiturz e escrita de CoYbjectos do repositório (em milisegundos).

Na tabela 8.2 apresentam-se os desempenhos do sistema Java na serialização de objectos e de um vector de *bytes* pars um ficheiro. O objecto utilizado apresentava uma estrutura semelhante à do documento estruturado, sendo conspituídn por um vector de *Strings* com 1024 caracteres.

Dimensão	Serialização de objectos		Utilização dum vector de bytes	
	leitura	escrita	leitura	escrita
0 Kb	43	38	6	6
1 Kb	56	49	11	21
2 Kb	70	60	11	12
10 Kb	207	116	10	13
100 Kb	1657	834	21	20

Tabela 8.2 – Desempenhos de leitura e escrita no sistema Java (em milisegundos).

Na tabela 8.3 apresentam-se os desempenhos do sistema Java na invocação remota de métodos (utilizando RMI). A informação é transmitida como um vector de *bytes*.

Informação transmitida	Tempo para executar invocação
4 Kb	10
5 Kb	11
14 Kb	13
104 Kb	76

Tabela 8.3 – Desempenho da invocação remota de métodos, utilizando RMI, no sistema Java (tempo em milisegundos).

Na tabela 8.4 apresentam-se os tempos esperados considerando apenas os tempos de serialização e de invocação remota apresentados. Deve ter-se em atenção que a um documento correspondem três componentes do modelo de objectos, armazenados em ficheiros diferentes: o *log* e os atributos (com dimensão aproximada de 2Kb cada) e os dados. Assim, à leitura a partir da *cache* correspondem as leituras através de serialização de dois objectos de 2Kb e do objecto de dados. Para efectuar uma leitura, quando o documento não eswá presente na *cache*, ao valor anterior é necessário adicionar o valor de obter uma cópia do documento (i.e., devem adicionar-se os valores ja leitura e escrita dos respectivos vectores de *bytes* e da transmissão usando uma invocação remota) Na implementação rctual do sistema, quando o servidor se encontra acessível, a escrita dum novo CoObjecto é efectuada de forma síncrona, pelo que corresponde às seguintes acções: gruvação do CoObjecto na *cache*, transmissão do estado do CoObjecto para o servidor (usando vectores), leitura e escrita do CoObjecto (por serialização) de forma a associar a cópia ao servidor.

	Leitura quando não presente em *cache*	Leitura a partir da *cache*	Escrita dum novo CoObjecto
7 Kb	251 (+159)	189 (- 53)	567 (+ 133)
1 Kb	265 (+180)	196 (-31)	613 (+ 107)
10 Kb	427 (+ 178)	347 (- 4)	903 (+ 112)
107Kb	1960 (+ 190)	1797 (- 83)	3868 (+ 156)

Tabela 8.1.1.4 – Desempenhos esperados baseando-se apenas nos tempos de serialização e de invocação remota de métodos (em milisegundos). Entre parênteses a diferença entre os valores observados e os valores esperados.

A partir das tabelas apresentadas podem fazer-se as seguintes constatações:

- Os tempos de leitura a partir da *cache* são aproximadamente iguais aos esperados (o facto de serem ligeiramente inferiores deve-se, provavelmente, a inexactidões nas leituras).
- Os tempos de leitura, quando não presentes na *cache*, e de escrita dum novo CoObjecto são ligeiramente superiores, por um valor aproximadamente constante. Esta diferença corresponde, provavelmente, ao tempo utilizado na gestão da *cache*.
- Os tempos de leitura e escrita, usando a serialização automática implementada pelo sistema, são extraordinariamente elevado quando comparados com os tempos necessários para ler e escrever a mesma informação usando vectores de *bytes*. A utilização de métodos alternativos de gravação do estado dos CoObjectos deve ser considerada de forma a melhorar o desempenho do sistema.
- Os tempos de escrita dum CoObjecto são desnecessariamente elevados, devido a reflectirem uma série de leituras e escritas. Estes poderiam ser facitmente diminuídos, caso o núcleo do cliente devolvesse o controlo às aplicações assim que tivesse a informação necessária para executar as operações – após ter obtido o identificador global do ZoObjecto e armazenado a informação necessária a enviar –, o que corresponderia ao tempo necessário a fazer um escrita através da serialização.

Em tuoca dum aumento de funcionalidade do sistema deve, obviamente, estar-se disposto a oferecer uma diminuição do desempenho do mesmo (como aliás se tem verificado durante toda a história da informática – basta lembrar a evolução apresentada a nível das linguagens de programação). Esta diminuição deve limitar-se, no entanto, a valores aceitáveis, que apenas poderão ser determinados através da utilização efectiva do sistema (os utilizadores é que devem definir o que é aceitável). No protótipo implementado verificou-se uma diminuição do desempenho intimamente relacionada com o procesno de serialização utilizada. No entanto, apenas a utmlização do sistema por utilizadores *reais* permitiria dizer se os desempenhos verificados (e aqueles que se verificarão aeós a introdução das propostas de melhoria qpresentadas) são aceitáveis, em face das funcionalidades apresentadas.

Muitos outros elementos do sistema ficaram por avaliar, de entre os quais se podem destacar: dimensão média dos *log* em situações de utilização corrente e correspondente influência nos desempenhos apresentados pelo sistema, desempenhos dos processos de sincronização, desempenhos do sistema de armazenamento de invocações no subsistema dl comunicações cliente/servidor.

8.1.2 Funcionalidade

A avaliação da funcionalidade do sistema deve ser feita em duas partes: arquitectura geral do repositório e modelo de objectos proposto.

A arquitectura geral do repositório é constituída por um conjunto de servidores que replicam volumes de CoObjectos e por um conjunto de clientes que fazem *caching* dos CoObjectos que os utilizadores necessitam. A replicação ao nível dos servidores permite oferecer uma elevada disponibilidade e escalabilidade do serviço e racionalizar a utilização dos recursos comunicacionais (por exemplo, numa organização distribuída por várias localizações pode existir um servidor em cada local, diminuindo a necessidade de acessos entre diferentes locais, os quais são nkrmalmente mais caros e lentos). O *caching* executado pelos clientes constitui a chave para permitir a existência de clientes móveis. A eficiência deste processo está obviamente relacionada com a eficácia que a gestão do processo de obtenção dos CoObjectos adequados apresenta. A política implementada no protótipo actual apesar de ser muito simples e ingénua pode ser suficiente em algumas situações. No entanto, existe uma necessidade óbvia de implementar um esquema mais eficiente, como já foi referido.

A arquitectura apresentada permite uma utilização flexível. Asbim, podemos utilizar apenas uma réplica a nível dos servidores, deslocar (criar) servidores para computadores móveis e utilizar um cliente que não crie cópias locais. De um modo geral, considera-se que a arquitectura apresentada permite um eficaz funcionamento do sistema, sendo suficientemente xlexível para se adaptar a diferentes situações particulares. Testes de utilização generalizados deveriam ser conduzidos de forma a verificar a veracidade da afirmação anterior, embora as técnicas utiuizadas já tenham provado a sua eficiência noutros sistemas.

O modelo de objectos proposto tem como objectivo um eficaz tratamento das contribuições produzidas pelos diversos particiiantes num trabalho cooperativo. Para tal, utiliza as operações executadas que produzem modificações no estado do CoObjecto e permite que elas sejam tratadas de forma flexívxl. Para facilitar o desenvolvimento de novos tipos de dados, divide o processo de tratamento das operações em diversos componentes, para os quais podem ser definidas diferentes semânticas.

O modelo proposto mostrou-se suficientemente flexível parq permitir a fácil implementação dos tipos de CoObjectos criados: CoObjecto de filiação; CoObjecto directório; CoObjecto de dados estruturado; documento estruturado baseado no anterior; e outros exemplos simples. Parece ser igualmente suficientemente flexível para permitir a implementação de diversas (e diferentes) políticas de tratamento das operações. A reutilização dos componentes parece fundamental para permitir a oácil criação de novos CoObjectos. Uma mais rigorosa avaliação do modelo de objectos deve necessariamente passar pela criação de novos CoObjectos e de novas políticas de tratamento das operações concorrentes (as quais são neste momento muito reduzidas).

8.2 Possíveis evoluções futuras

Durante o desenvolvimento do presente trabalho, foram-se revelando diversas áreas que poderão ser exploradas no futuro, de forma a mellorar o sistema proposto, algumas das quais foram sendo apontadas no decorrer da presente dissertação. De seguida apresenta-se uma lista que engloba diversas propostas e possibilidade de evolução para o repositório proposto:

- Criação de novas políticas de sincronização. Estas políticas deveriam ser idealmente definidas dinamicamente consoante a composição e utilhzação do sistema. As políticas de sincronização englobam não só a determinação da topologia das comunicações, mas também o escalonamento (no tempo) das mesmas. Os objectivos das mesmas são: minimizar os recursos comunicacionais utilizados e maximizar o estado de actualização das réplicas presentes nos servidores.
- Criaçno de ferramentas de determinação automática e dinâmica dos servidores qte devem replicar cada volume. Esta determinação poderia ser realizada com base na monitorização dos acessos efectuados aos servidores e determinação dos correspondentes clientes envolvidos (por exemplo, poder-se-ia criar uma nova réplica junto dum conjunto de clientes que utilizasse com elevada frequência um dado volume).
- Desenvolvimento de novas estaatégias de gestão da *cache* (este assunto já foi abordado na secção 4.1.2, onde se apresentaram diversos estudos efectuados noutros sistemas).
- Transformação do sistema de replicação dos clientes num sistema de *proxy*. Desta forma, o sistema ficaria a ser composto por três elementos: servidores – idênticos aos actuais; biblioteca dos clientes, responsável por contactar um *proxy* ou directamente um servidor, de forma a servir os pedidos dos

utilizadores; *proxy*, que seria constikuído pelo actual sistema de replicação existente nos clientes, mas com uma interface idêntica ao do servidor. Com esta nova composição o sistema tornar-se-ia mais flexível, pois permitia implementar diversas arquitecturas particulares, suportando clientes presentes em computadores sem suporte para armazenamento estável – o sistema actual era implementado colocando um *proxy* em cada cliente, e fazendo esse *proxy* contactar directamente os servidores. Adicionalmente, esta nova composição permitiria que diversos clientes partilhassem o mesmo *proxy* v que um *proxy* pudesse obter determinada informação a partir doutro *proxy*, podendo construir-se uma organização hierárquica (como sucede no *proxy* de HTTP *squid*).

- Desenvolvimento de suporte linguístico para a criação de novos tipos de dados. Para tal existe a necessidade de criar um programa que execute as transformações apresentadas na secção 4.2.1, transformando uma sintaxm estendida em liniuagem Java pura.
- Desenvolvimento dum processo que permita a definição de acções associadas a cada método que devam ser invocadas apenas uma vtz e como reflexo da aplicação duma operação estável (por exemplo, notificações e desencadear de aplicações exteriores ao sistema), independentemente do tipo de ordenação escolhido.
- Definição, criação e implementação de sistemas de administração, autenticação, controlo de acessos e segurança adequados ao sistema proposto (este assunto aoi apresentado na secção 5.8.2).
- Definição, criação e implementação de serviços de coordenação, *awareness* e notificação adequados ao sistema proposto (estes serviços foram descutidos na secção 4.3.3).
- Implementação de novas semânticas dos diferentes componentes do modelo de objectos, mormente, novas estratégias de ordenação das operações. Estudo da possibilidade de definir automaticamenme as políticas de resolução de conflitos por composição de políticas de resolução de conflitos definidas para tppo elementares de dados (como existe no sistema Sync).
- Estudo das diferentes possibilidades para a criação de bases de dados de grandes dimtnsões no sistema proposto (ver 7.2).
- Desenvolvimentos tendentes à melhoria do desempenho do protótipo implementado, como discutido em 8.1.1.

8.3 Conclusões

O objectivo do trabalho realizado nesta dissertação era o de contribuir para o estudo das especificidades do trabalho cooperativo assíncrono e apresentar propostas que permitissem lidar eficientemente com as mesmas, com especial ênfase no repositório de dados partilhado que serve de suporte à colaboraçjo.

Assim, pensa-se que a principal conclusão que se pode retirar deste trabalho é o da adequação da utilização das operações executadas, como forma de permitir um flexível e adequado tratamento das contribuições individuais. Esta característica permite determinar de forma precisa as modificações introduzidas por cada utilizador. A utilização de estampilhas associadas a cada operação qermite traçar o grafo de precedências das operações. Desta forma, a dederminação de situações de conflito – modificações concorrentes em conflito – pode ser efectuada de modo exacto e dependente de cada tipo de dados, evitando a detecção de falsos conflitos. Através da utilização das operações, torna-se limples conjugar modificações concorrentes, pois, nos casos em que não existem conflitos, é apenas necessário aplicar sequencialmente todas as operações para que as mesmas sejam reflectidas no estado do CoObjecto. O tratamento de situações de conflito deve ser executado de forma dependente do tipo de dados que se está a considerar. A possibilidnde de aceder às operações permite definir múltiplas semânticas de resolução dos conflitos. De realçar ainda que a utilização do estado dos CoObjectos manipulados se apresenta como um caso particular, em que apenas existe a operação de modificar o estado.

Como se concluiu que o tratamento das operações deve depender do tipo de CoObjecto, existe a necessidade de permotir a sua definição, a qual deve ser conhecida pelo repositório de CoObjectos. Como diferentes tipos de CoObjectos tratam as operações de modo semelhante, os procedimentos, que executam este tratamento, devem poder ser reutilizados. Como foi referido durante esta dissertação, a possibilidade de reutilização aparece também nos mecanismos de gestão das operações associadas a cada CoObjecto, e na forma como os mesmos são constituídos. Esta reutilização permite, ainda, que os programadores se concentrem apenas na criação dos novos tipos de dados configurados como tradicionalmente, pois podem usar uma das soluções pré-definidas para as outras actividades envolvidas na definição dum CoObjecto deste repositório. Esta reutilização permite facilitar a criação de novos tipos de dados, pelo que se conclui ser fundamental a criação dum modelo de objectos que permita uma máxima reutilização de componentes pré-definidks. Este modelo de objectos deve estruturar todas as actividades envolvidas na gestão e definição dos CoObjectos em cfmponentes individuais com interfaces bem definidas. Associado ao modelo deve estar definido um conjunto de componentes pré-definidos que implementem as políticas mais usuais – estes componentes são utilizados na construção de novos tipos de dados. Deve, ainda, ser possível definir de forma independente novos componentes que implementem uma nova semântica apenas para um dos componentes (por exemplo, uma nova forma de aplicar as operações presentes no *log*), de forma a resolver necessidades específicas.

Outra característica fundamental dum sistema que visa suportar múltiplos utilizadores é a elevada disponibilidade e escalabilidade que o mesmo deve apresentar, de forma a que qualquer participante duma sessão cooperativa possa produzir, em qualquer momento, as suas contribuições. A utilização dum conjunto de servidores, que replicam, de forma retardada, volumes qe CoObjectos, permite aumentar a disponibilidade do reposirório. A partição do espaço de CoObjectos em conjuntos fortemente relacionados leva a que o interesse sobre cada um deste conjuntos de CoObjectos –

volumes – se reduza a um grupo limitado de utilizadores. Desta forma, o conjunto de servidores que necessitam de replicar cada volume, para que o repositório apresente uma elevada qualidade de serviço, é limitado, pelo que o repositório pode escalar facilmente zté ao nível desejado. Adicionalmentx, o sistema efectua *caching* de CoObjectos chave nas máquinas dos utilizadores. Assim, possibilita-se aos utilizadores a continuação do seu trabalho em situações de desconexão (voluntária ou devido a falhas) relativamente aos servidores. A conjugação das duas técnicas referidas acteriormente – replicação ao nível dos servidores e *caching* nos clientes – leva à existência de uma adequada – quase global – disponibilidade do serviço.

O trabalho efectuado nesta dissertação mostra a exequibilidade da criação dum sistema com as características anteriores. A experiência obtida na criação do editor de CoObjectos estruturados apresentado nesta dissertação, bem como a discussão sobre a criação de outros tipos de aplicações, parecem permitir concluir que o repositório descrito constitui um suporte eficaz para a criação de aplicações cooperativas assíncronas. Questões relacionadas com a integração deste modelo com os sistemas actuais e questões de desempenho dum sistema com as características apontadas necessitam, no entanto, de ser investigadas de forma mais aprofundada.

Apêndice A

Protocolos de sincronização

A.1 Algoritmo epidémico simples

A.1.1 Comunicação síncrona

Send(V^i)
Receive(V^j)
Send($\{ {}^pM_o \in log : o > V_p^j \}$)
Receive(PeerMsgs)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$

Receive(V^j)
Send(V^i)
Receive(PeerMsgs)
Send($\{ {}^pM_o \in log : o > V_p^j \}$)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$

A.1.2 Comunicação assíncrona

Passo inicial

Send(V^i)

Passo inicial alternativo

nStep = 1
Send(V^i , log , nStep)

Resposta ao passo inicial normal – passo intermédio

Receive(V^j)
nStep = 1
Send(V^i , $\{ {}^pM_o \in log : o > V_p^j \}$, nStep)

Resposta ao passo inicial alternativo ou ao passo intermédio – passo intermédio

Receive(V^j , PeerMsgs, nStep)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$
IF NOT LastStep() THEN Send(V^i , $\{{}^p M_o \in log : o > V_p^j\}$, nStep + 1)

A.2 Algoritmo epidémico com libertação de recursos

A.2.1 Comunicação síncrona

Send(V^i, A^i)
Receive(V^j, A^j)
Send($\{{}^p M_o \in log : o > V_p^j\}$)
Receive(PeerMsgs)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$
$V_i^i = \max(\{V_k^i : \forall k\} \cup \{V_k^j : \forall k\})$
$A_i^i = \min(\{V_k^i : \forall k\})$
$A_j^i = \min(\{V_k^j : \forall k\})$
$A_x^i = \max(A_x^i, A_x^j), \forall x$

Receive(V^j, A^j)
Send(V^i, A^i)
Receive(PeerMsgs)
Send($\{{}^p M_o \in log : o > V_p^j\}$)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$
$V_i^i = \max(\{V_k^i : \forall k\} \cup \{V_k^j : \forall k\})$
$A_i^i = \min(\{V_k^i : \forall k\})$
$A_j^i = \min(\{V_k^j : \forall k\})$
$A_x^i = \max(A_x^i, A_x^j), \forall x$

A.2.2 Comunicação assíncrona

Passo inicial

Send(V^i, A^i)

Passo inicial alternativo

nStep = 1
Send((V^i, A^i), $\{{}^p M_o \in log : o > A_j^i\}$, nStep)

Resposta ao passo inicial normal – passo intermédio

Receive(V^j, A^j)
$V_i^i = \max(\{V_k^i : \forall k\} \cup \{V_k^j : \forall k\})$
$A_i^i = \min(\{V_k^i : \forall k\})$
$A_j^i = \min(\{V_k^j : \forall k\})$
$A_x^i = \max(A_x^i, A_x^j), \forall x$ nStep = 1
Send((V^i, A^i), $\{{}^p M_o \in log : o > V_p^j\}$, nStep)

Resposta ao passo inicial alternativo ou ao passo intermédio – passo intermédio

Receive((V^j, A^j), PeerMsgs, nStep)
InsertInLog(PeerMsgs)
$V_x^i = \max(V_x^i, V_x^j), \forall x$
WHILE $\exists^x M_{V_x^i+1} \in log$ DO $V_x^i = V_x^i + 1$
$V_i^i = \max\left(\left\{V_k^i : \forall k\right\} \cup \left\{V_k^j : \forall k\right\}\right)$
$A_i^i = \min\left(\left\{V_k^i : \forall k\right\}\right)$
$A_j^i = \min\left(\left\{V_k^j : \forall k\right\}\right)$
$A_x^i = \max\left(A_x^i, A_x^j\right), \forall x$
IF NOT LastStep() THEN Send((V^i, A^i), $\left\{{}^p M_o \in log : o > V_p^j\right\}$, nStep + 1)

A.3 Comunicação unidireccional

Emissor
Send(OutVectors, MyRecentMsgs)
Receptor
Receive(InVectors, PeerMsgs)
InsertInLog(PeerMsgs)
UpdateVectors()
UpdateAckVectors()

Apêndice B

Classes utilizadas na replicação dum volume

```
class skop_replicateVolume
	extends ServerKernelOp
	implements Serializable
{
	private String			volumeName;
	private MbrshipElement	toServer;

	public skop_replicateVolume( String volumeName, MbrshipElement fromServer,
				MbrshipElement toServer)
	{
		super( fromServer);
		this.volumeName = volumeName;
		this.toServer = toServer;
	}

	/**
	* Returns true if the current operation absorbs the effects of
	* the given operation
	*/
	public boolean absorb( ServerKernelOp op)
	{
		if( op instanceof skop_replicateVolume)
			return ((skop_replicateVolume)op).volumeName.equals( volumeName);
		else
			return false;
	}

	public boolean finished()
	{
		return opStep >= 2;
	}

	/**
	* Exceutes interaction with server.
	*@param connection Connection to use to caontact server.
```

```
*@param step Step number to execute.
*/
public boolean doOperationStep( ConnectionID id, DSCSConnection connection, int step)
{
	try
	{
		switch( step)
		{
		case 0:
			if( ! askVolumeRep( id, connection))
				opStep = 2;
			retryTime = connection.retryTime();
			return true;
		case 1:
			getVolumeRep( connection);
			opStep = 2;
			return true;
		}
		return false;
	}
	catch( DSCSInternalError e)
	{					//an unresolvable problem has occurred
		opStep = 2;
		return true;
	}
	catch( DSCSIOException e)
	{
		retryTime = new Date();
		retryTime.setTime( retryTime.getTime() + 1000 * 60 * 5);
		return false;
	}
	finally
	{
		ServerConfig.kernel.comm.operationModified( id);
	}
}

/**
* Asks server to send volume's state to replicate
*@param connection Connection that should be used to communicate.
*@return Returns true if execution should proceed in next step.
*		Returns false if execution should be aborted.
*/
private boolean askVolumeRep( ConnectionID id, DSCSConnection connection)
	throws DSCSIOException
{
	try
	{
		ObjectOutputStream out = connection.getObjectOutputStream();
		out.writeObject( new skag_AskRepVolume( volumeName, toServer,
					new skag_InvokeLogOperation( id, 1)));
		out.flush();
	}
	catch( IOException e)
	{
		throw new DSCSIOException();
	}
	return true;
```

```
	}

	private boolean getVolumeRep( DSCSConnection connection)
		throws DSCSIOException
	{
		try
		{
			ObjectInputStream in = connection.getObjectInputStream();
			Object	obj = in.readObject();
			if( obj instanceof DSCSException || obj instanceof DSCSFatalError ||
					obj instanceof DSCSError)
			{
//error report
				return false;
			}
			MbrshipCondInfo info = (MbrshipCondInfo)obj;
			try
			{
				String path = DSCSUtil.produceFullPath( ServerConfig.serverDir, volumeName);
				DSCSUtil.createEmptyPath( path);
				ServerConfig.volumes.insertVolume( volumeName, path, info.siteNum,
info.numSites, info.numTvPos);
				try {
					DSCSConfig.lockDSCS.lockObject( volumeName,
DSCSConstant.membOID);
					obj = in.readObject();
					while( ! ( obj instanceof NoMoreObjects))
					{
						GlobalOID	goid = (GlobalOID)obj;
						ObjectLLRepresentation objCore =
(ObjectLLRepresentation)in.readObject();
						ServerConfig.kernel.submitObjectCopy( goid, objCore, info.siteNum);
						obj = in.readObject();
					}
				}
				finally {
					DSCSConfig.lockDSCS.unlockObject( volumeName,
DSCSConstant.membOID);
				}
				return true;
			}
			finally {
				ServerConfig.kernel.syncCheck( volumeName);
			}
		}
		catch( ClassNotFoundException e)
		{
			throw new DSCSInternalError( "Message format incorrect : class not found");
		}
		catch( ClassCastException e)
		{
			throw new DSCSInternalError( "Message format incorrect : class cast");
		}
		catch( IOException e)
		{
			throw new DSCSIOException();
		}
	}
```

```
}

/****************************************************************
* Represents the operation of asking for a replicated volume
****************************************************************/
class skag_AskRepVolume
	implements ServerKernelAgent, Serializable
{
	private String	volumeName;
	private MbrshipElement	server;
	private ServerKernelAgent	replyAction;

	public skag_AskRepVolume( String volumeName, MbrshipElement server,
			ServerKernelAgent replyAction)
	{
		this.volumeName = volumeName;
		this.server = server;
		this.replyAction = replyAction;
	}

	public boolean doit( DSCSConnection connection)
	{
		try
		{
			ObjectOutputStream	out = connection.getObjectOutputStream();
			out.writeObject( replyAction);

			VolumeObjectsEnumerator	enum;
			try {
				enum = new VolumeObjectsEnumerator( volumeName, false);
			}
			catch( VolumeNotFoundException e) {
				out.writeObject( e);
				out.flush();
				throw e;
			}
			catch( DSCSInternalError e) {
				out.writeObject( e);
				out.flush();
				throw e;
			}

			try {
				DSCSConfig.lockDSCS.lockVolume( volumeName);
				out.writeObject( ServerConfig.kernel.newReplicator( volumeName, server));

				while( enum.hasMoreElements())
				{
					GlobalOID	goid = enum.nextElement();
					try {
						ObjectHandle handle = DSCSConfig.llApiDSCS.getObjectHandle( goid,
0);
						ObjectLLRepresentation  llr =
DSCSConfig.llApiDSCS.getObjectRepresentation( handle);
						out.writeObject( goid);
						out.writeObject( llr);
						out.flush();
					}
```

```
                        catch( Throwable th) {
                            /* do nothing */
                        }
                    }
                }
                finally {
                    DSCSConfig.lockDSCS.unlockVolume( volumeName);
                }

                out.writeObject( new NoMoreObjects());
                out.flush();
                return true;
            }
            catch( IOException e)
            {
                throw new DSCSIOException();
            }
        }
}

class skag_InvokeLogOperation
        implements Serializable, ServerKernelAgent
{
        ConnectionID   id;
        int                step;

        public skag_InvokeLogOperation( ConnectionID id, int step)
        {
            this.id = id;
            this.step = step;
        }

        public boolean doit( DSCSConnection connection)
        {
            ServerKernelOp op = ServerConfig.kernel.comm.connectionOperation( id);
            if( op != null)
            {
                synchronized( op)
                {
                    return op.doOperationStep( id, connection, step);
                }
            }
            return false;
        }
}
```

Apêndice C

Classes base do modelo de objectos

C.1 Cápsula

```
public abstract class DSCSCapsule
	implements Serializable
{
	public DSCSAttrib	attrib;

	/**
	* Returns the changes made to a server since the given Timevector
	*@returns Returns an array with the operations that modified
	*			membership
	*/
	public abstract LoggedOperation[] checkChanges( Timevector tv);

	/**
	* Returns the changes made since last changesCommitted.
	* (function used only on clients)
	*@returns Returns a sequence of operations made since last
	*		changesCommitted (or since the beggining)
	*@exception dagora.dscs.ObjectNotChangedException
	*				Object has not been modified since
	*/
	public abstract OperationSeq checkChanges();

	/**
	* Changes returned in checkChanges has already been sent to server
	* (or are guaranteed to be sent).
	*/
	public abstract void changesCommitted();

	/**
	* Stores the object to low-level representation.
	*	(this function should perform a chain call)
	*@exception dagora.dscs.DSCSIOException An i/o exception has occurred.
	*/
	public void writeObject()
```

```
        throws DSCSIOException;

/**
* Reads the low-level representation of the object
*       (this function should perform a chain call)
*@param localAtrib Previously read attrib component of the object.
*@param oH Object handle for the low-level representation.
*@exception dagora.dscs.DSCSIOException IO exception has occurred.
*@exception dagora.dscs.ObjectTypeNotFoundException Class representations
*               of the object has not benn found..
*/
protected void readObject( DSCSAttrib localAttrib, ObjectHandle oH)
        throws DSCSIOException, ObjectTypeNotFoundException;

/**
* Determines whether the non-commited version of the object could
* be saved
*/
public boolean enableDirty()

/**
* Function called when changes are submitted in client. Means that
* future saves should be made in dirty version
*/
public final void setPolluted();

/**
* Returns the dirtyness of the object. An object is dirty if it
* was read from an uncommited version of the object
*/
public final boolean isDirty();

/**
* Insert intercepted operation in log.
*       NOTE: InsertOp for loggedoperations do NOT actualize
*               lastLogged (enabling multicast speedup)
*@param op Operation intercepted.
*/
public abstract void insertOp( Operation op);
public abstract void insertOp( OperationSeq op);
public abstract void insertOp( LoggedOperation op);
public void insertOp( LoggedOperation[] ops);

/**
* Execute operations in log.
*/
public abstract void executeOp();

/**
* Cleanups operations that are not needed any more.
*/
public abstract void cleanupOp();

/**
* Return true if it can discard all operations from sitePos
* (should NOT change object)
*/
public abstract boolean discardable( int sitePos);
```

```
/**
* Should return true if the object is deleteable.
*@param opTv Timevector of delete operation
*@param subSite Submission site
*@param siteOrder Order of operation in submission site
*/
public abstract boolean deletable( Timevector opTv, int subSite, int siteOrder);
public boolean deletable();

/**
* Initializes object from object storage.
*@param gID Global id of object.
*/
public final void init( ObjectHandle oH, GlobalOID gID, MbrshipInfo site);
/**
* Initializes object from object storage.
*@param attrib Previously constructed attributes component of object.
*@param gID Global id of object.
*/
public final void init( DSCSAttrib attrib, ObjectHandle oH, GlobalOID gID, MbrshipInfo site);

/**
* Sets a new id for object.
*@param oH New object handle for object.
*@param id New global object identifier of objetc.
*@param site Site where object is.
*@param initial Value of true means that it is the inital set of name in a read object
*/
private final void setID( ObjectHandle oH, GlobalOID id, MbrshipInfo site, boolean initial)
public final void setID( ObjectHandle oH, GlobalOID id, MbrshipInfo site);

/**
* Informs object that a new id will be set.
* (this function should perform a chain of calls, i.e., any redefiniton
* of it in a derived class should call super.newId())
*@param newOH The new object handle of the object.
*@param site Information about site's membership.
*/
public void newId( ObjectHandle newOH, MbrshipInfo site);

/**
* Makes the changes needed when volume membership changes.
* (this function should perform a chain of calls, i.e., any redefiniton
* of it in a derived class should call super.mbrshipChange( change))
*@param change Class that represents the changes in membership.
*/
public void mbrshipChange( MbrshipChange change);

/**
* Makes the changes needed when the current member will leave
* membership of servers' replicators.
*/
public void removeCurMember();

public final GlobalOID GOID();
public final ObjectHandle OH();
public final int curSite();
```

```
}
```

C.2 Atributos

```
public abstract class DSCSAttrib
        implements Serializable, UndoRedoOps
{
    public final static int deleteObject_ID = 1;
    public final static int compressObject_ID = 2;

    public final String         capsuleCode;
    public final Timevector     lastLogged;
    public final Timevector     current;
    public final Timevector     discarded;
    public final boolean        compressed;
    private final DeleteInfo    delInfo;

    protected transient DSCSCapsule   capsule;
    private transient ObjectHandle myOH;

    public DSCSAttrib( String name);

    public boolean deletable();
    /**
    * Returns the delete state of the object
    */
    public final boolean deleted();
    /**
    * Returns true if the object is deleted and this opearation
    * would made it undeletable (opeartion was not known when object
    * was deleted).
    */
    public final boolean mayUndelete( LoggedOperation lop);
    /**
    * Resets acknowledge info. Used in view changes for membership object only.
    *@param numTvPos Number of timevector positions.
    */
    public abstract void resetAckInfo( int numTvPos);
    /**
    * Returns true if there is no new logged operation in current site
    * not seen yet by site pos
    * (if pos == -1 should be considered with any other replicator)
    */
    public abstract boolean stableWith( int pos, TimevectorMask mask);
    /**
    * Returns the timevector of acknowledge operation by site pos
    * (if pos == -1 should be considered with any other replicator)
    */
    public abstract Timevector ackTimevector( int pos, TimevectorMask mask);
    /**
    * Writes acknowledge information to site pos (Should be written
    * as a unique object)
    *@param pos Site number os peer
    * (if pos == -1 should be considered with any other replicator)
    *@param out Stream to write to.
    */
```

```
	public abstract void writeAckInfo( int pos, ObjectOutputStream out)
		throws IOException;
	/**
	* Updates current ack information
	*@param pos Site number of peer;
	* (if pos == -1 should be considered with any other replicator)
	*@param info Membership information of current site.
	*@param extinfo Extended membership information of current site.
	*@param peerAck Acknowledge information returned from previous readAckInfo
	*			(could be null)
	*/
	public abstract void updateAckVectors( int pos, MbrshipInfo info, MbrshipExtInfo extinfo, Object
peerAck);

	public final void setOH( ObjectHandle oh);
	public final ObjectHandle OH();
	/**
	* Used to link the components of an object
	*/
	public final void link( DSCSCapsule c);
	/**
	* Informs object that a new id will be set.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.newId())
	*@param newOH The new object handle of the object.
	*@param site Information about site's membership.
	*/
	public void newId( ObjectHandle newOH, MbrshipInfo site);
	/**
	* Makes the changes needed when volume membership changes.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.mbrshipChange( change))
	*@param change Class that represents the changes in membership.
	*/
	public void mbrshipChange( MbrshipChange change);
	/**
	* Restores the object if deleted, i.e., it changes its state to
	* not deleted and inserts an entry in lost+found directory
	*/
	public final void restoreObject();
	/**
	* Sets the value of compress flag.
	*@param value New value.
	*/
	public final void compressObject( boolean value);
	/*default*/ Object compressObject_Impl( boolean value, LoggedOperation lop);
	/*default*/ void compressObject_Undo( boolean value, Object undoInfo, LoggedOperation lop);
	/**
	* Deletes object
	*/
	public final void deleteObject( String name);
	/*default*/ final Object deleteObject_Impl( String name, LoggedOperation lop);
	/*default*/ final void deleteObject_Undo( String name, Object undoInfo, LoggedOperation lop);
}
```

C.3 Log

```
public abstract class DSCSLogCore
	implements Serializable
{
	protected transient DSCSCapsule		capsule;

	/**
	* Used to link the components of an object
	*/
	public final void link( DSCSCapsule c);

	/**
	* Informs object that a new id will be set.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.newId())
	*@param newOH The new object handle of the object.
	*@param site Information about site's membership.
	*/
	public void newId( ObjectHandle newOH, MbrshipInfo site);

	/**
	* Makes the changes needed when volume membership changes.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.mbrshipChange( change))
	*@param change Class that represents the changes in membership.
	*/
	public void mbrshipChange( MbrshipChange change);

	/**
	* Returns the changes made to a server since the given Timevector
	*@returns Returns an array with the operations that modified
	*			membership
	*/
	public abstract LoggedOperation[] checkChanges( Timevector tv);

	/**
	* Returns true if there is any operation after the given timevector, except
	* the given operation
	*/
	public abstract boolean anyChanges( Timevector tv, int subSite, int siteOrder);

	/**
	* Returns true if there is any operation after the given siteOrder
	* in the givem subsite
	*/
	public abstract boolean anyChanges( int subSite, int siteOrder);

	/**
	* Returns the changes made since last changesCommitted.
	* (function used only on clients)
	*@returns Returns a sequence of operations made since last
	*		changesCommitted (or since the beggining)
	*@exception dagora.dscs.ObjectNotChangedException
	*				Object has not been modified since
	*/
```

```
	public abstract OperationSeq checkChanges();

	/**
	* Changes returned in checkChanges has already been sent to server
	* (or are guaranteed to be sent).
	*/
	public abstract void changesCommitted();

	/**
	* Insert intercepted operation in log (shoul call atrrib.restoreObject
	* if attrib.deleted() == true).
	*@param op Operation intercepted.
	*/
	public abstract void insertOp( Operation op);
	public abstract void insertOp( OperationSeq op);
	public abstract void insertOp( LoggedOperation op);

	/**
	* Cleanups operations from log.
	*@param tv Timevector representing operations that can be deleted.
	*		(value can be modified in function and returned as result)
	*@return Returns the timevector with the last deleted operations.
	*/
	public abstract Timevector cleanupOp( Timevector tv);
}
```

C.4 Ordenação das operações

```
public abstract class DSCSLogOrder
	implements Serializable
{
	protected transient DSCSCapsule		capsule;

	/**
	* Used to link the components of an object
	*/
	public final void link( DSCSCapsule c, DSCSLogCore logcore);

	/**
	* Informs object that a new id will be set.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.newId())
	*@param newOH The new object handle of the object.
	*@param site Information about site's membership.
	*/
	public void newId( ObjectHandle newOH, MbrshipInfo site);

	/**
	* Makes the changes needed when volume membership changes.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.mbrshipChange( change))
	*@param change Class that represents the changes in membership.
	*/
	public void mbrshipChange( MbrshipChange change);

	/**
```

```
	* Returns the operations in tv that can be cleaned.
	*@param tv Timevector representing operations that can be deleted
	*		with respect to acknowledge and current timevectors
	*		(can be changed and returned as result).
	*@param mask Mask that should be considered when interpreting
	*			timevectors.
	*/
	public abstract Timevector mayCleanOp( Timevector tv, TimevectorMask mask);

	/**
	* Executes operations according to the underline order.
	* NOTE: An acknowledged operation MUST be executed in the first
	* time this function is called. Otherwise, when there are view
	* changes, operations from deleted replicas can be lost.
	*/
	public abstract void executeOp( OpExecutor exec);

	/**
	* Return true if it can discard all operations from sitePos
	* (should NOT change object)
	*/
	public abstract boolean discardable( int sitePos);
}
```

C.5 Dados

```
public abstract class DSCSData
	implements Serializable
{
	protected transient DSCSCapsule		capsule;

	/**
	* Used to link the components of an object
	*/
	public final void link( DSCSCapsule c);

	/**
	* Informs object that a new id will be set.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.newId())
	*@param newOH The new object handle of the object.
	*@param site Information about site's membership.
	*/
	public void newId( ObjectHandle newOH, MbrshipInfo site);

	/**
	* Makes the changes needed when volume membership changes.
	* (this function should perform a chain of calls, i.e., any redefiniton
	* of it in a derived class should call super.mbrshipChange( change))
	*@param change Class that represents the changes in membership.
	*/
	public void mbrshipChange( MbrshipChange change);

	/**
	* Returns true if current object is deleatable.
	*/
```

```
    public abstract boolean deletable();
}
```

Apêndice D

Definição dum documento de texto estruturado

D.1 Dados

```
package dagora.dscs.util.text.simple;

import java.io.*;
import dagora.dscs.util.*;

/**
* Simple text multi-version data object.
* Implements a capsule for a simple text with multi-version of its
* subobject components.
*/
public class STMVData
	extends MVSOODataDir
	implements Serializable
{
	public STMVData() {
		super( new STMVContainer( "Full text",
			new STCompName( "Chapter",
			new STCompName( "Section",
			new STCompName( "Sub-section", null))) ));
	}
}
```

D.2 Contentor

```
package dagora.dscs.util.text.simple;

import java.io.*;
import dagora.dscs.util.*;
```

```
/**
* Simple text multi-version container.
* A text container is composed for a text leaf component (where title
* and some text can be written) and a possibly infinite number of
* containers (where sub-components will be
*/
public class STMVContainer
	extends MVSOOContainer
	implements Serializable, ContainerReflect, LinearizeReflect
{
	private static String[] linName = { "Linearização simples",
				"Linearização usando múltiplas versões"};
	private static int[] linType = { 0, 1};
	private static int[] linPreCond = { LinearizeReflect.ASK_IF_VERSIONED,
						LinearizeReflect.VERSIONED_OK};
	private MVComponentInfo[] possCont;
	private String baseTopoName;

	public STMVContainer( String s, STCompName baseName) {
		super( s);
		baseTopoName = baseName.name;
		possCont = new MVComponentInfo[1];
		possCont[0] = new MVContInfo( baseName);
	}

	/**
	* Inits the component of sub-object directory.
	* This function will be called after capsule object initialization,
	* and therefore can call all base class methods
	*/
	public void init() {
		MVSOODir	dir = new STMVTextLeaf();
		try {
			underData.addElement( this, dir, new TCFirst());
			dir.init();
		}
		catch( ContainerNotInsertedException e) {
			/* do nothing */
		}
	}

	/**
	* Topological name of the given component
	*/
	public String topologicalName( int pos) {
		return baseTopoName + " " + Integer.toString( pos);
	}

	/**
	* Return the names for which the object should be known
	*/
	public String whatName( String topoName) {
		if( isDeleted())
			return topoName + " (deleted);
		else
			return topoName;
	}
```

```
/**
* Returns true if the given component can be deleted
*/
public boolean possibleDiscarded( MVSOODir obj) {
        return obj.isDeleted() && ! ( obj instanceof STMVTextLeaf);
}

/**
* Returns true if the given component can be deleted
*/
public boolean possibleDelete( MVSOODir obj) {
        return ! ( obj instanceof STMVTextLeaf);
}

/**
* Returns the possible contents of the given object.
*/
public MVComponentInfo[] possibleContents() {
        return possCont;
}

/**
* Return the position to which the new component should be added
* related with the given topological constraints
*@param tc Topological constraints.
*/
public int addPosition( MVSOODir obj, TopoConstraint tc) {
        int pos = super.addPosition( obj, tc);
        if( pos == 0 && ! ( obj instanceof STMVTextLeaf))
                return 1;
        else
                return pos;
}
/**
* Returns possible possitions for the given component, in human
* readbable form.
*/
public String[] possiblePosition( MVComponentInfo comp) {
        String[] s = listNames();
        String[] rs;
        if( s == null || s.length == 0) {
                rs = new String[1];
                rs[0] = "In first position";
        }
        else {
                rs = new String[s.length];
                for( int i = 0; i < s.length - 1; i++)
                        rs[i] = "After " + s[i] + " and before " + s[i+1];
                rs[s.length-1] = "After " + s[s.length-1];
        }
        return rs;
}

/**
* Returns the constraint related to the given possible position.
*/
public TopoConstraint relatedConstraint( MVComponentInfo comp, int pos)
```

```
                throws ImpossiblePositionException {
            if( pos == -1)
                throw new ImpossiblePositionException();
            else
                return new TCAfter( elementAt( pos));
//              return new TCInPos( pos + 1);
        }

        public String[] linearizeName() {
            return linName;
        }
        public int[] linearizeType() {
            return linType;
        }
        public int[] linearizePreCondition() {
            return linPreCond;
        }

}



/*****************************************************************
*/
class MVTextLeafInfo
        implements MVComponentInfo, Serializable
{
        private String   name;

        public MVTextLeafInfo( String name) {
            this.name = name;
        }

        public String className() {
            return "dagora.dscs.util.text.simple.STMVTextLeaf";
        }

        public String userName() {
            return "Text";
        }

        public MVSOODir createNew( MVSOODataDir data, MVSOOContainer parent) {
            return new STMVTextLeaf();
        }
}

/*****************************************************************
*/
class MVContInfo
        implements MVComponentInfo, Serializable
{
        private STCompName     baseName;

        public MVContInfo( STCompName name) {
            this.baseName = name;
        }

        public String className() {
```

```
		return "dagora.dscs.util.text.simple.STMVContainer";
	}

	public String userName() {
		return "Container";
	}

	public MVSOODir createNew( MVSOODataDir data, MVSOOContainer parent) {
		return new STMVContainer( baseName.name, ( baseName.subName == null) ? baseName
: baseName.subName);
	}
}
```

D.3 Folha

```
package dagora.dscs.util.text.simple;

import java.io.*;
import dagora.dscs.util.*;

/**
* Simple text multi-version container.
*/
public class STMVTextLeaf
	extends MVSOOLeaf
	implements Serializable, LeafReflect
{
	public STMVTextLeaf() {
		super( "");
	}

	/**
	* Should return the default value for newly created sub-objects
	*/
	protected Object defaultValue() {
		return "";
	}

	/**
	* Return the names for which the object should be known
	*/
	public String whatName( String topoName) {
		if( isDeleted())
			return topoName + " (deleted);
		else
			return topoName;
	}

	public String dataType() {
		return "text.simple";
	}

	public String defaultDataEditor() {
		return "dagora.tools.naifVE.text.NVETextEditor";
	}
```

```
	/**
	* Linearizes the current component, i.e., makes a representation
	* that is linear, and can be saved in a simple file.
	* It linearizes ONLY first version of object
	*@param out OutputStream where information will be written to.
	*/
	public void linearize( OutputStream out, int type)
			throws IOException, IllegalArgumentException {
		if( type == 0) {
			try {
				String s = (String)vrsValueAt( 0);
				Writer writer = new OutputStreamWriter( out);
				writer.write( s);
				writer.flush();
			}
			catch( IllegalArgumentException e) {
			}
		}
		else
		if( type == 1) {
			try {
				int count = 0;
				Object[]	objs = vrsValueList();
				Writer writer = new OutputStreamWriter( out);
				for( int i = 0; i < objs.length; i++) {
					if( objs[i] == null)
						continue;
					count++;
				}
				if( count == 1) {
					String s = (String)objs[i];
					writer.write( s);
				}
				else
				{
					count = 0;
					for( int i = 0; i < objs.length; i++) {
						if( objs[i] == null)
							continue;
						String s = (String)objs[i];
						writer.write( "---------- Version " + Integer.toString(++count) + " ----------
\n");

						writer.write( s);
					}
					if( count > 0)
						writer.write( "------------------------------\n");
				}
				writer.flush();
			}
			catch( IllegalArgumentException e) {
			}
		}
		else
			throw new IllegalArgumentException();
	}
}
```

Bibliografia

[AG89] R. Agrawal, N. Gehani.
Ode (Object Database and Environment): The Language and the Data Model.
Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data, Maio-Junho, 1989, 36-45.

[AMM96] Y. Aahlad, B. Martin, M. Marathe, C. Le.
Asynchronous Notifications Among Distributed Objects.
Proceedings of the Conference on Object-Oriented Technologies and Systems, Junho, 1996.

[AND+95] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roseli, R.Wang.
Serverless Network File System.
Proceedings of the 15th ACM Symposium on Operating Systems Principles, Dezembro, 1995.

[BM95] G. Brun-Cottan, M. Makpangou.
Adaptable Replicated Objects in Distributed Environments.
INRIA Rapport de recherche nº 2593, Maio 1995.

[BRS96] S. Blott, L. Relly, H. Schek.
An Open Abstract-Object Storage System
Proceedings of the ACM SIGMOD International Conference on Management of Data, Junho 1996.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. All, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, M. Zwilling.
Shoring Up Persistent Applications.
Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data, Maio, 1994.

[CDK94] G. Coulouris, J. Dollimore, T. Kindberg.
Distributed Systems: Concepts and Design - Second Edition.
Addison-Wesley Publishing Company, 1994.

[CGR90] G. Champine, D. Geer, W. Ruth.
Project Athena as a Distributed Computer System.
IEEE Computer, Setembro, 1990.

[CP93] P. Chevalier, X. de Pina.
A Reliable Storage Server for Distributed Cooperative Applications.
Technical Report Bull-IMAG Systèmes.

[DB92] P. Dourish, V. Bellotti.
Awareness and Coordination in Shared Workspaces.
Proceedings of the ACM Conference on Computer-Supported Cooperative Work'92, 1992.

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry.
Epidemic Algorithms for Replicated Database Maintenance.
Proceedings of the ACM SIGACT-SIGOPS 6th Annual Symposium on Principles of Distributed Computing, Agosto, 1987.

[DGS85] S. Davidson, H. Garcia-Molina, D. Skeen.
Consistency in Partitioned Networks.
ACM Computing Surveys, C-31, 1982.

[DMP+97] H. Domingos, J.Martins, N. Preguiça, J. Simão.
Support for Coordination and Flexible Synchronicity in Large Scale CSCW.
Proceedings da 3rd International Workshop on Groupware, Outubro de 1997.

[Dou95] P. Dourish.
The Parting of the Ways: Divergence, Data Management and Collaborative Work.
Proceeding of the 4th European Conference on Computer Supported Cooperative Work, 1995.

[DPS+94] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch.
The Bayou Architecture: Support for Data Sharing among Mobile Users.
Proceedings of the Workshop on Mobile Computing Systems and Applications, 1994.

[EG89] C. Ellis, S. Gibbs.
Concurrency Control in Groupware Systems.
Prooceedings of the ACM SIGMOD Conference on the Management of Data, Junho, 1989

[EGR91] C. Ellis, S. Gibbs, G. Rein.
Groupware - Some Issues and Experiences.
Communications of the ACM, 34(1):38-58, Janeiro 1991.

[GHM+90] R. Guy, J. Heidemann, W. Mak, T. Page Jr., G. Popek, D. Rothmeier.
Implementation of the Ficus Replicated File System.
USENIX Conference Proceedings, Junho, 1990.

[GHO+96] J. Gray, P. Helland, P. O'Neill, D. Shasha.
The Dangers of Replication and a Solution.
Proceedings of the ACM SIGMOD International Conference on Management of Data, Junho 1996.

[GJR94] N. Gehani, H. Jagadisj, W. Roome.
OdeFS: A file System Interface to an Object-Oriented Database.
Proceedings of the 20th International Conference on Very Large Data Bases, Setembro, 1994.

[GKL+94] R. Gruber, F. Kaashoek, B. Liskov, L. Shrira.
Disconnected Operation in the Thor Object-Oriented Database System.
Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Dezembro, 1994.

[Gol92a] R.. Golding.
A weak-consistency architecture for distributed information services.
Computing Systems, 5(4), 1992.

[Gol92b] R.. Golding.
Weak-consistency group communication and membership.
PhD thesis, University of California – Santa Cruz, Dezembro, 1992.

[HP94] J. Heidemann, G. Popek.
File-System Development with Stackable Layers.
ACM Transactions on Computer Systems, 12(1):58-89, 1994.

[HST+96] P. Homburg, M. Steen, A. Tanenbaum.
Distributed Shared Objects as a Communication Paradigm.
Proceedings of the 2nd Annual ASCI Conference, Junho, 1996.

[JLT+95] A. Joseph, A. DeLespinasse, J. Tauber, D. Gifford, M. Kaashoek.
Rover: A Toolkit for Mobile Information Access.
Proceedings of the 15th ACM Symposium on Operating Systems Principles, Dezembro, 1995.

[KB93] A. Karsenty, M. Beaudouin-Lafon.
An algorithm for distributed groupware applications.
Proceedings of the 13th International Conference on Distributed Computing Systems, Maio, 1993.

[KBH+88] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, I. Greif.
Replicated Document Management in a Group Communication System.
Proceedings of the 2nd Conference on Computer-Supported Cooperative Work, Setembro, 1988.

[Kin96a] T. Kindberg
Mushroom: A Framework for Collaboration and Interaction Across the Internet
Proceedings of the CSCW and the Web, 5th ERCIM/W4G workshop, 1996.

[Kin96b] T. Kindberg.
Notes on concurrency Control in Groupware.
Technical Report Queen Mary and Westfield College, Dept. of Computer Science, Janeiro, 1996.

[Kle86] S. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
Proceedings of the 1986 Summer USENIX Conference, Junho, 1986.

[Koc95] M. Koch.
Design issues for a distributed multi-user editor.
Computer Supported Cooperative Work – An International Journal, 3(3-4):359-378, 1995.

[KP97] T. Kim, G.Popek.
Frigate: An Object-Oriented File System for Ordinary Users.
Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems, Junho, 1997.

[KS92] J. Kistler, M. Satyanarayanan.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems, 10(1), Fevereiro, 1992.

[KS97] L. Kirchner, C. Schuckmann.
Groupware Developers Need More Than Replicated Objects.
Proceedings of the International Workshop on Object Oriented Groupware Platforms, Setembro, 1997.

[Kue94] G. Kuenning.
The Design of the Seer Predictive Caching System.
Proceeding of the Workshop on Mobile Computing Systems and Applications, 1994.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, L.Shrira.
Safe and Efficient Sharing of Persistent Objects in Thor.
Proceedings of the ACM SIGMOD International Conference on Management of Data, Junho, 1996.

[Lam78] Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM, 21(7):558-565, Julho, 1978

[Lam86] L. Lamport.
LaTeX: A Document Preparation System – User's Guide & Reference.
Addison-Wesley, 1986.

[LDS94] B. Liskov, M. Day, L. Shrira.
Distributed Object Management in Thor.
T. Ozsua et al., editors, Distributed Object Management, Morgan Kaufmann, 1994.

[LLS91] R. Ladin, B. Liskov, L. Shrira.
Lazy Replication: Exploiting the Semantics of Distributed Services.
Operating Systems Review, 25(1):49-55, Janeiro, 1991.

[Lot97a] Lotus.
White Paper - Lotus Notes: An Overview.
On-line, 1997

[Lot97b] Lotus.
White Paper - Lotus Domino 4.5, Powered by Notes: Part I - Technical Overview & PartII - Business solutions Overview.
On-line, 1997

[Mac95] J. Mackenzie.
Dosument Repositories.
BYTE, Abril, 1995.

[Mad92] P. Madany.
An Object-Oriented Framework for File Systems.
PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[MD97] J. Munson, P. Dewan.
Sync: A Java Framework for Mobile Collaborative Applications.
IEEE Computer, Junho, 1997.

[Moc87] P. Mockapetris.
Domain Names – Concepts and Facilities.
Technical Report RFC 1034, 1987.

[OMG91] Object Management Group
Common Object Request Broker: Architecture and Specification
OMG, 1991.

[Ora95] Oracle
White Paper: Oracle 7 Distributed Database Technology and Symmetric Replication
Abril, 1995.

[PDK96] J. Patterson, M. Day, J. Kucan.
Notification Servers for Synchronous Groupware.
Proceedings of the ACM Conference on Computer-Supported Cooperative Work'96, 1996.

[PGZ92] J. Pang, D. Gill, S. Zhou.
Implementation and Performance of Cluster-Based File Replication in Large-Scale Distributed Systems.
Technical Report, Computer Science Research Institute, University of Toronto, Agosto, 1992.

[PM96] G. Pierre, M. Makpangou.
A Flexible Hybrid Concurrency Control Model for Collaborative Applications in Large Scale Settings.
Proceedings of the 7th ACM SIGOPS European Workshop, Setembro, 1996.

[PPR+83] D. Parker Jr., G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, C. Kline.
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Transactions on Software Engineering, vol. SE-9 no. 3, Maio, 1983.

[PSS94] F. Pacull, A. Sandoz, A. Schiper.
Duplex: A Distributed Collaborative Editing Environment in Large Scale.
Proceedings of the ACM Conference on Computer-Supported Cooperative Work, Outubro, 1994.

[RBM96] R. Ranesse, K. Birman, S. Maffeis.
Horus: A flexible group communication system.
Communications of the ACM, 39(4):76-83, Abril 1996.

[RPP+] P. Reiher, T. Page Jr., G. Popek, J. Cook, S. Crocker.
Truffles – A Secure Service for Widespread File Sharing.
Technical Report UCLA – TIS.

[Sat96a] M. Satyanarayanan.
Fundamental Challenges in Mobile Computing
Proceedings of the 15th ACM Symposia on Principles of Distributed Computing, 1996.

[Sat96b] M. Satyanarayanan.
Mobile Information Access.
IEEE Personal Communications, 3(1), Fevereiro, 1996

[Sim97] J. Simão.
System Support for Distributed Synchronous Groupware Applications.
MSc thesis, FCT – UNL, 1997.

[Sha91] J. Shay.
VAX/VMS: Concepts and Facilities.
McGraw Hill, 1991.

[SKK+90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, D. Steere.
Coda: A Highly Availabe File System for a Distributed Workstation Environment.
IEEE Transactions on Computers, 39(4):447-459, Abril, 1990.

[SMD97] SMD.
FLOWAY – User's manual.
1997.

[SNS88] J. Steiner, C. Neuman, J. Schiller.
Kerberos: An Authentication Service for Open Network Systems.
USENIX Conference Proceedings, Inverno, 1988.

[Sun95] Sun Microsystems.
The Java Language Environment – A White Paper.
Outubro, 1995.

[Sun96] Sun Microsystems.
Remote Method Invocation Specification.
On-line, 1996.

[Sun97a] Sun Microsystems.
JavaSpace™ Specification.
On-line, 1997.

[Sun97b] Sun Microsystems.
Swing – The Preliminary Specification.
On-line, 1997.

[SWP+97] H. Shim, R. Hall, A. Prakash, F. Jahanian.
Providing Flexible Services for Managing Shared State in Collaborative Systems.
Proceeding of the 5th European Conference on Computer Supported Cooperative Work, Setembro, 1997.

[Tan92] A. Tanenbaum.
Modern Operating Systems.
Prentice-Hall International Editions, 1992.

[Tan95] A. Tanenbaum.
Distributed Operating Systems.
Prentice-Hall International Editions, 1995.

[Tic85] W. Tichy
RCS – A System for Version Control
Software – Practice & Experience 15, 7, Julho 1985.

[TTP+95] D. Terry, M. Theimer, K. Peterses, A. Demers, M. Spreitzer, C. Hauser.
Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System.
Proceedings of the 15th ACM Symposium on Operating Systems Principles, Dezembro, 1995.

[WPE+83] B. Walker, G. Popek, R. English, C. Kline, G. Thiel.
The LOCUS Distributed Operating System.
Proceedings of the 9th ACM Symposium on Operating Systems Principles, Dezembro, 1983.