Supporting Groupware in Mobile Environments

Nuno Preguiça, J. Legatheaux Martins
Henrique J. Domingos and Sérgio Duarte

Technical report 4-2002 DI-FCT-UNL

Departmento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal

{nmp,jalm,hj,smd}@di.fct.unl.pt

Supporting Groupware in Mobile Environments*

Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, Sérgio Duarte
Departamento de Informática
Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa
Quinta da Torre, 2825-114 Monte da Caparica, Portugal
{nmp, jalm, hj, smd}@di.fct.unl.pt

Abstract

This technical report describes a replicated object store designed to support typically asynchronous collaborative activities in mobile environments. The system allows the definition of type-specific data management solutions using an object framework that includes awareness management. High data availability is provided in mobile environments relying on server replication and client caching and exploring ad hoc networks established among clients. Partial caching allows operation on mobile devices with reduced resources. *Blind* invocation allows users to produce new contributions on data that is not locally available, during disconnected operation. The system also supports the integration of synchronous sessions in the overall asynchronous activity.

Keywords

Mobile computing, data management, asynchronous groupware, synchronous groupware, integration.

INTRODUCTION

The use of portable computers and wireless networks has increased dramatically in the last years. Portable computers range from small personal digital assistants (PDAs) with limited resource to powerful notebooks that can be used as desktop replacements. These devices can connect to a fixed network while docked or using wireless networks. Furthermore, they can establish ad hoc wireless networks to communicate among them directly (e.g. using Bluetooth [5] or Wireless Ethernet [25]).

In this technical report, we present a new version of the DAgora distributed storage system (named DOORS [2]), focusing on the new mechanisms introduced to provide a better support for users working on mobile computers. This system is designed to support typically asynchronous collaborative activities. However, synchronous sessions can be integrated in the overall asynchronous activity. In the context of the DAgora project [1] we have also addressed other aspects that are important for groupware, such as the coordination of collaborative activities [3] and event dissemination [4] (e.g. to propagate awareness information).

In asynchronous groupware, users usually collaborate accessing and modifying information without immediate knowledge about the actions of other users (either because users work at different times or simply because they do not have access to each other's actions). Our overall approach to support these applications is based on the combination of two elements. First, a highly available replicated object store that allows users to produce their contribution without restrictions (besides coordination and access control restrictions). These characteristics maximize the chance for collaboration. Second, an object framework that addresses most aspects related with data sharing, including concurrency control and awareness management. New data types, to be used in collaborative applications, can be easily created, according to this framework, reusing

* This work has been partially support by FCT/MCT.

the pre-defined components with adequate characteristics (we call the data objects manipulated by the system and structured according to this framework coobjects – from collaborative objects).

This technical report details several mechanisms introduced to improve the support for mobile computing: partial caching; *blind* invocation to overcome cache misses; and the exploration of ad hoc networks among mobile users. Additionally, we describe the support to integrate synchronous session in the overall asynchronous activity.

Sometimes, users only need to access a subset of data to produce their contributions – e.g. in the collaborative editing of a structured document, a user may only need to access some of the sections. The partial caching mechanism explores this property maintaining only partial copies of coobjects. This mechanism improves the support for mobile computers with limited storage and/or bandwidth.

During periods of disconnection, data management systems usually prevent any access to data not locally cached. However, users may still want to produce contributions that affect the unavailable data – e.g. a user may submit a new version for some section in a shared document. These updates need to be merged with the current data state. However, awareness information of groupware activities may allow good contribution. We have implemented a *blind* invocation mechanism that allows the execution of operations over data not locally cached ("replacement" objects can be used to observe the expected result of operations).

Mobile clients may sometimes connect with each other using ad hoc wireless networks (or any other form of communication). During these periods, clients may update their caches directly (without server intervention). Furthermore, clients may establish synchronous sessions among them to modify coobjects that are being typically asynchronously manipulated. The new object framework supports the manipulation of coobjects in synchronous sessions. This mechanism allows the combination of synchronous and asynchronous work session using the DOORS system.

The remainder of this technical report is organized as follows. Section 2 discusses requirements and design choices. Section 3 presents an overview of the system. Section 4 details the mechanisms implemented to support groupware in mobile environments. Section 5 describes the support for synchronous sessions. Section 6 discusses related work and section 7 concludes the technical report with some final remarks.

REQUIREMENTS AND DESIGN CHOICES

In this section we present the requirements and design choices that lead to DOORS. Some of these requirements have already been discussed in an earlier version of the system [2]. In this technical report, we briefly outline the most important aspects already addressed, and focus on the new requirements. To illustrate some of the requirements we will use examples from two applications: a group calendar and a multi-user editing tool. Similar situations can be found in other typical asynchronous groupware applications.

In a group calendar, the appointments from a group of users are maintained in a shared calendar. Users should be allowed to request the introduction of new appointments independently, even during disconnection. These new appointments should be considered tentative [8] until they are committed using some form of automatic global agreement. It should also be possible for users to request a new appointment even if they can not locally access the shared calendar. If requested, users should be notified when their requests are committed or aborted.

A multi-user editing tool allows a group of users to collaboratively edit some structured document. Concurrent asynchronous modifications should be adequately merged maintaining syntactic consistency [7]. Users should be allowed to make their contributions independently. Cache misses should be reported but should not prevent users from making their contributions. Users should be allowed to manipulate the document during synchronous sessions. Awareness information about document evolution should be maintained with the document to be presented to users.

Basic requirements

We start our discussion reviewing some requirements that had already been discussed in [2]. However, as they are fundamental in our approach to support asynchronous groupware they are briefly presented here.

From the previous brief descriptions, we note that users collaborate through the access and modification of shared data. Therefore, one fundamental requirement to maximize the chance for collaboration is to allow users to access and modify shared data without restrictions. In DOORS, data is replicated by groups of servers to mask networks failures/partitions and server failures. Mobile clients cache data to mask disconnections. High read and write availability is obtained through a "read any/write any" model of data access that allows all clients to modify data independently.

This optimistic approach leads to the need of handling divergent streams of activity (caused by independent concurrent updates executed by different users). In most systems, concurrent updates are handled using a single customizable strategy (e.g., in Bayou [8], the following strategy is always used: updates are executed in the same order in all replicas; when a conflict is found – using the user-defined condition – a new update is generated – using the user-defined conflict-resolution function – and executed). However, multiple strategies have been proposed in literature (e.g. using of undo-redo techniques [14], operational transformations [23,24], searching the best solution using semantic information [15]). It seems that no single strategy is adequate for all situations. Instead, different groups of applications call for different strategies. In DOORS, we allow the use of different strategies in different applications – coobjects define its own reconciliation/concurrency-control strategy.

Awareness has been identified as important for the success of collaborative activities because individual contributions may be improved by the understanding of the activities of the whole group [6,21]. In DOORS, we have designed an integrated mechanism for handling awareness information relative to the evolution of the shared data. Each update can produce awareness information when it is processed in the server (and its definite result is obtained). Each coobject may include support to present this awareness information using a shared feedback style [6] (when awareness messages are maintained with the coobjects and presented in applications) and/or a notification style (where users receives these messages directly, for example, as short messages in their mobile phones/pagers). Users can customize the awareness support specifying which messages they are interested on.

Another important requirement is to ease the development of new groupware applications with different characteristics. To support flexible type-specific solutions, DOORS delegates on coobjects most of the aspects related with data sharing (including concurrency control and awareness management). We have defined a data management object framework that decomposes each coobject in several components, each one responsible for a different aspect of object "operation". Using this framework, programmers can design a new global solution reusing the most adequate solution (or implementing a new one) for each aspect identified in the framework. This approach allows programmers to use different strategies in different applications. Furthermore, complex distributed systems algorithms can be used in a simple way. A new version of the object framework has been introduced in this new version of our system.

Partial caching

As it has been mentioned, high data availability is a fundamental requirement to maximize the chance for collaboration in asynchronous groupware applications. Mobile computers need to rely on local data copies to provide such availability. The granularity of caching may influence the ability to support *small* mobile devices. Several approaches have been used in different distributed storage systems.

In Coda [16], as well as in other distributed file-systems, the unit of caching is the complete file. This approach allows a simple and clean file access semantics – as the file is the unit of caching, either it is possible to access the whole file or nothing at all (in a *cache miss*). However, when files are very large this approach may be a problem for mobile computers with limited storage resources. The users may also consider excessive the time needed to obtain the full data copy from a server over slow wireless networks. As, in some

situations, users only need to access a fraction of the data, partial caching can be used to adequately support mobile computers with different resources.

In object-based systems, applications usually manipulate complex graphs of small objects ([10] mentions the average size of 100 bytes). To support disconnected operation, it is possible to cache only a subset of all objects. However, the determination of which objects must be cached is very complex because objects are usually small and highly interconnected – even when objects are clustered in pages of objects this process works at the level of objects [10].

In DOORS, we have introduced a strategy that can be seen as a tradeoff between those two. Programmers should define a coobject as a set of interconnected sub-objects. Each sub-object can still be a complex object and be implemented as a composition of many small objects. However, a sub-object is a unit by itself for caching purposes – e.g., a section of a structured document can be a sub-object. Furthermore, it is possible to invoke operations on each sub-object independently. This approach enables the partial caching of a coobject, while allowing users to continue to work in the cached parts – e.g., a user can cache only the sections she wants to modify and edit them while disconnected. We detail our partial caching approach in section 4.

Blind invocation

In distributed storage systems, cache misses usually prevent users to continue using data. However, in some situations, users may still produce useful work despite the unavailability of some data. For example, in a shared calendar, a user may request the scheduling of new appointments even when he cannot access the calendar. In a structured document where each section may have multiple versions, a user may want to create a new version despite he cannot access the current versions. These updates must be integrated in the current state of objects, in the severs.

In groupware applications, the coordination and awareness information associated with the collaborative process may maximize the chance to produce new good-quality contributions in these situations. For example, if the users had previous access to the awareness information related with data evolution, they should have an (almost) up-to-date knowledge about the current data state (although the data is unavailable through the data management system). If (formal or informal) coordination rules set the responsibility of creating some section in a shared document to some user, he will tend to know which updates he should execute in the context of this task, even when "cache misses" occur.

In DOORS, we have introduced the following mechanisms to allow users to continue their work in these situations, thus minimizing the effects of cache misses. First, users can execute operations over sub-objects that are not locally cached. These operations are logged in the clients and later propagated to the servers, where they are integrated in the current data state using the reconciliation strategy defined for the coobject. Second, a sub-object not locally available can be instantiated in "replacement" mode. Therefore, besides being able to invoke operations, it is also possible to observe the expected results of the executed operations. We detail these mechanisms in section 4.

Ad hoc communication and synchronous collaboration

Mobile computers may, sometimes, communicate with each other inexpensively using ad hoc peer-to-peer wireless networks. Distributed data management systems may explore these communications to update the caches in mobile computers: new objects or new versions of existing objects may be obtained directly from their peers (i.e. without contacting any server). In DOORS, we have introduced this feature, allowing clients to obtain new coobjects/sub-objects and/or new updates to coobjects from other mobile clients.

Mobile users can engage in synchronous collaborative sessions using these ad hoc wireless networks. During these synchronous sessions, users may want to update data objects that are being typically updated in an asynchronous way. For example, a group of users may be producing a document asynchronously. However, at some moments, they can engage in synchronous sessions to make some important modifications and co-

ordinate their work – e.g., to decide the structure of the document or merge multiple versions of some sections. It should be noticed that users can also engage in this kind of interactions using their desktop computers – mobile computers and wireless communications only add the (possible) convenience of allowing (some) users to meet together in the same (physical) place without requiring any additional network infrastructure.

In DOORS, we have introduced the possibility to manipulate coobjects in synchronous sessions. To this end, the adaptation component of the object framework can be used to maintain the state of several local copies of the same coobject (synchronously) synchronized. In section 5 we detail this mechanism and discuss the integration of synchronous sessions in the overall asynchronous activity (a multi-synchronous editing tools is presented).

SYSTEM OVERVIEW

In the previous section we have discussed the main requirements addressed in our system. In this section, we present an overview of the system and of the associated object framework. The partial caching and the *blind* invocation mechanisms and the integration of synchronous sessions will be detailed in the following sections.

Architecture and working model

DOORS is a distributed object store based on a "extended client/replicated server" architecture. It manages coobjects – objects structured according to the DOORS object framework. A coobject represents a *data type* designed to be shared by multiple users in an asynchronous way, such as a structured document, a shared calendar or a shared spreadsheet. Unlike our previous version, a coobject is designed as a cluster of sub-objects, each one representing part of the whole *data type* (e.g. a structured document can be composed by one sub-object that maintains the structure of the document and one sub-object for each element of this structure). It is important to notice that each sub-object may still represent a complex data structure and it may be implemented as an arbitrary composition of common objects. Besides the cluster of sub-objects, a coobject contains several components that manage the operational aspects of data sharing – figure 1 depicts the approach (we detail each component and how they work together later). Sets of related coobjects are grouped in volumes that represent collaborative workspaces and store the data associated with some work-group and/or collaborative project.

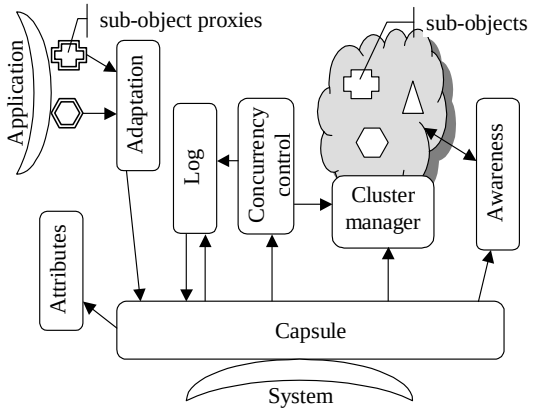

Figure 1 – DOORS object framework.

The DOORS architecture is composed by servers and clients, as depicted in figure 2. Servers replicate volumes of coobjects to mask network failures/partitions and server failures. Server replicas are synchronized during pair-wise epidemic synchronization sessions. Clients partially cache key coobjects to allow users to continue their work, even while disconnected. A partial copy of a coobject includes all the components necessary to instantiate a coobject (the components that manage the operational aspects of data sharing) and a

subset of the sub-objects included in the coobject. Clients can obtain these partial replicas directly from a server or from other clients. They can also update their local copies directly from other clients, thus exposing to users the recent contributions executed by other users.

Applications run on client machines and usually use a "get/modify locally/put changes" model of data access. First, the application obtains a private copy of the coobject (from the DOORS client). Second, the application invokes sub-objects' methods to query and modify its state (as it would do with common objects). The update operations are transparently logged (and compressed) in the coobject. Sub-objects are only loaded (instantiated) when they are accessed – this process is transparent for the applications. Finally, if the user chooses to save her changes, the logged sequence of operations is (asynchronously) propagated to a server.

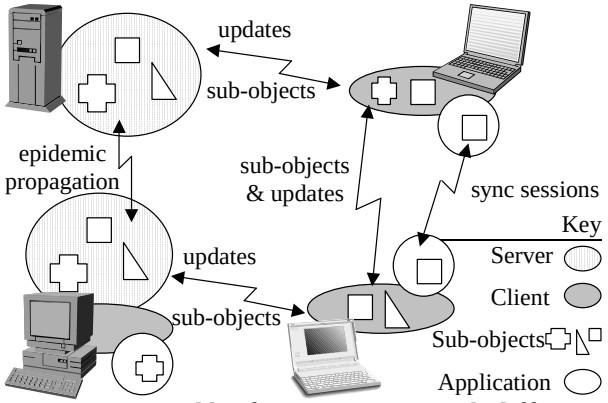

Figure 2 – DOORS architecture composed by four computers with different configurations. Coobjects are replicated by servers, partially cached by clients and manipulated by users' applications.

When a server receives operations from a client, it delivers the operations to the local replica of the coobject. It is up to the coobject replica to store and process these operations. During the epidemic synchronization sessions, servers propagate sets of operations between coobjects' replicas.

DOORS is fully built around the notion of operation-based update propagation, as the previous description shows. The system core is almost restricted to propagate the sequences of updates and to maintain the client cache, i.e., it only executes the minimal services that represent the common aspects of data management. DOORS delegates on the coobjects most of the aspects related with the management of data sharing, such as concurrency control and the handling of awareness information. The rationale behind this design is to allow flexible type-specific solutions.

Object framework

The outlined design imposes a heavy burden on the implementation of coobjects, which must handle several aspects that are usually managed by the system. To alleviate programmers from much of this burden and to allow the reuse of "good" solutions in multiple data types, we have defined an object framework that decomposes a coobject in several components that handle different operational aspects (see figure 1). In this section we outline the complete object framework, focusing on the elements (adaptation component and sub-objects, sub-object proxies and the cluster manager) introduced to accommodate partial caching, *blind* invocation and the integration of synchronous sessions. A deeper discussion about the other components, including the implemented pre-defined solutions, can be found in [2].

Each coobject is composed by a set of sub-objects that may reference each other using sub-object proxies. These sub-objects store the internal state and define the operations of the implemented data-type. The cluster manager is responsible to manage the sub-objects that belong to the coobject, including: the instantiation of sub-objects (when needed); and the control of sub-objects' persistency. Sub-objects' persistency can be

achieved through garbage-collection or explicit creation/deletion instructions (leading to two different component implementations).

Applications always manipulate coobjects' data through sub-objects' proxies. When an application invokes a method on a sub-object proxy, the proxy encodes the method invocation (into a simple object) and hands it over to the adaptation component[1]. The adaptation component is responsible to decide the interactions with the exterior. The most common adaptation component executes operations locally. We have also defined a remote-execution adaptation component that immediately propagates all operations to be executed in a server (using a service provided by the system core). It is also possible to define adaptation components that adapt to local or remote execution depending on the connectivity. In section 5 we will show how we have used this component to enable coobjects to be manipulated during synchronous sessions.

Local execution is controlled by the capsule component. Query operations are executed immediately in the respective sub-object and the result is returned to the application. Update operations are logged in the log component, which adds to these operations the necessary information to order them and to trace their dependencies.

The concurrency control/reconciliation component is responsible to execute the operations stored in the log. In the client, an immediate tentative execution is usually done so that users can see the expected results of their updates. However, an update only affects the "official" state of a sub-object when it is finally executed in the servers. To guarantee that the multiple (server) replicas of the coobject evolve in a consistent way and that users intentions are respected when the updates are executed in the servers, different concurrency control/reconciliation components implementing different strategies may be used in the server (we have discussed this problem extensively in [2]). During the execution of the operations some awareness information may also be produced. This information is handed over to the awareness component that immediately processes it (storing it to be later present in applications and/or propagating it to the users using the services of the system core).

Besides controlling the local execution of operations, the capsule component defines the coobject composition and aggregates its components. Although the presented composition represents a common coobject, it is possible to define different compositions– for example, it is possible to maintain a tentative and a committed version of the sub-objects relying on two different concurrency control components to execute the updates stored in the log using an optimistic and a pessimistic total order strategy respectively. The capsule component also defines the interface with the system (to exposing the logged operations and processing the operations received). Finally, the attributes component stores the system and type-specific properties of the coobject.

To create a new data-type (coobject) the programmer must do the following. First, he must define the sub-objects that will store the data state and define the operations (object methods) to be used to query and to change its state. From the sub-objects code, a pre-processor generates the code of sub-object proxies and factories to be used to create new sub-objects, handling the tedious details automatically.

Second, he must define the coobject composition used in the client and in the server selecting the adequate pre-defined components (or defining new ones if necessary). It is important to notice that coobjects encode most of the data-sharing semantics through these components, thus allowing the definition of different semantics using different pre-defined components. Another important aspect is that the operational components that are part of a coobject may be different in the server and in the client – for example, the concurrency control/reconciliation component uses different algorithms in the server and in the client.

---

[1] When an operation is invoked in a sub-object proxy as the result of the execution of other operation (note that sub-objects reference each other using proxies), it is simply executed against the respective sub-object.

MECHANISMS FOR MOBILE COMPUTING

In this section we will detail the partial caching and the *blind* invocation mechanisms. These mechanisms are very important to allow mobile users in disconnected mobile computers to participate in collaborative activities. To illustrate the use of these mechanisms we use some applications that we have implemented – however, due to space limitations, we can only describe more closely the multi-synchronous document editor (in [2] we present descriptions of the older versions of other applications).

Example: Multi-synchronous document editor

The multi-synchronous document editor allows users to produce structured documents collaboratively – these documents are represented as coobjects. A document is a hierarchical composition of two types of elements: containers and leaves. Containers are sequences of other containers and/or leaves. Leaves represent atomic units of data that may have multiple versions and that may be of different types. For example, a LaTeX document has a root container that may contain a sequence of text leaves and/or scope containers. A scope container may also contain a sequence of text leaves and/or scope containers. There is no direct association between these elements and LaTeX commands/elements. Users are expected to use scope elements to encapsulate the document structure. For example, a paper can be represented as a sequence of scope elements, each one containing a different section (see figure 3). The file to be processed by LaTeX is generated serializing the document structure – all text is contained in text leaves.

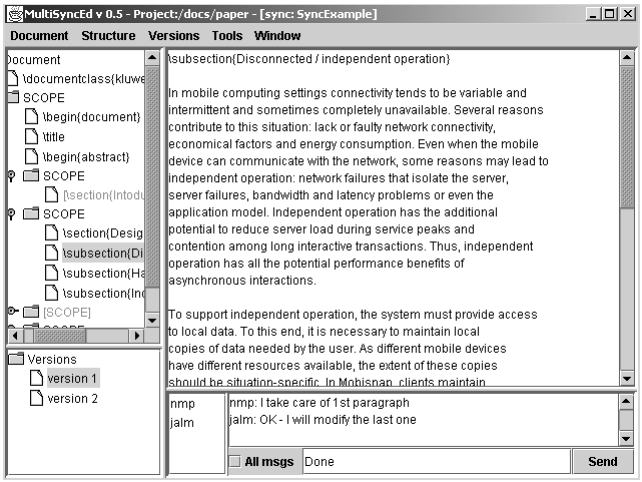

Figure 3 – Multi-synchronous document editor with a LaTeX document, while synchronously editing one section.

When editing the document asynchronously, users are allowed to change the same elements independently. The coobject manages concurrent modifications automatically, maintaining syntactic consistency, as follows. Concurrent modifications to the same text leaf are merged using the pre-defined strategy implemented in the multi-version sub-object (text leaves are defined as a sub-type of this sub-object) – two versions are created if the same version is concurrently modified; a remove version is ignored if that version has been concurrently modified; otherwise, both updates are considered. Users should merge multiple versions into a single version later. Concurrent modifications to the same container are merged executing all updates in a consistent way in all replicas (to this end, an adequate total order reconciliation component is used in the servers).

Awareness information is maintained with the document to show data evolution using a *shared-feedback* [6] style. Unlike the previous version, this information is maintained separately for each sub-object (to support partial caching).

During synchronous edition, users can observe immediately the modification executed by other users. Additionally, they can synchronously edit the same text leaves.

Partial caching

As it has already been outlined, the partial caching solution implemented in out system is based on the division of the coobject's data in smaller elements: the sub-objects. Each sub-object is composed internally by a composition of small objects and it can only reference other sub-objects through sub-object proxies. Applications modify the cobjects' state executing operations on these sub-objects. When a client caches some coobject it may cache only a subset of all sub-objects contained in the coobject.

The rationale for this approach is the following: the programmer that designs the coobject is the person that knows better which small objects should be cached together (based on the internal organization and operations defined in the coobject). Therefore, if she can makes this information explicit to the system, the system will be able to do a better job on partial caching. Thus, sub-objects are used to group the small objects that should be cached together.

Consider the LaTeX structured document used in our multi-synchronous document editor. Sub-objects are identified immediately in consequence of the internal organization of the coobject: containers and text leaves are sub-objects. Each client may cache only the sub-objects (containers and text leaves) that contain the part of the document the user is interested on. The editor shows that some document elements are not locally available in the interface using a different color to represent those elements – see figure 3.

In other applications, sub-objects can also be identified easily. In a conferencing system, each thread of messages can be a sub-object. In a calendar, the appointments for each day can be grouped together in a single sub-object. These examples seem to confirm the idea that, in many applications, programmers can easily identify the units for partial caching from the data organization.

DOORS also defines an advanced mechanism that allows to indicate, for each sub-object, the set of other sub-objects that should be cached with it: the system uses this information when a sub-object is cached. The cluster manager is responsible to manage this information. Our pre-defined cluster manager components only keep track of relations specified when sub-objects are created, but more complex policies using automatic and dynamic grouping of sub-objects could be implemented.

In the LaTeX document, we have used this mechanism in a simple way. When a text leaf/container is created it specifies the container where it is going to be inserted as a "must also be cached" element. Thus, it is possible to guarantee that the relevant document structure elements will be available when accessing some sub-object[2].

In a calendar, where appointments are to be accessed either by date or by user, this feature can also be useful. A possible approach would be to define each appointment as a single sub-object and the indexing structures also as sub-objects (e.g. a sub-object maintains references for the appointments scheduled for a given day). Each indexing structure should force included appointments to be cached with it – the necessary relations can be set easily when appointments are created/changed. Therefore, a client can cache the appointments for a given day or a given user, depending on the information the user is interested on (our calendar application always presents information by date, therefore we have used the organization mentioned earlier).

Blind invocation

The blind execution mechanism allows users to execute new contributions, while disconnected, even during cache misses. We expect that the coordination and awareness information associated with the collaborative process allow users in collaborative tasks to produce good-quality contributions (exploiting blind invocation).

---

[2] An efficient caching algorithm should be able to provide the same guarantees (e.g.[17]), but this problem is out of the scope of our work. In our current prototype, we have implemented a simple least-recently-used algorithm.

As we have presented, sub-objects reference other sub-objects using sub-object proxies. Due to partial caching, it is possible that some sub-object reference other sub-objects that are not locally available (e.g. in a LaTeX document, a scope container may have references to text leaves that are not available locally). The *blind* invocation allows users to execute operations on these sub-object proxies.

When an operation is executed on a sub-object proxy and the sub-object is not (and can not currently be) cached, the processing of the invocation proceeds as usually until the execution of the operation (e.g., an update operation is marshaled by the proxy, handled by the adaptation component, logged in the log component and scheduled for local execution). In this moment, if the sub-object (or any other sub-object used during operation execution) is not available, an exception is thrown notifying the application that a cache miss has occurred. However, as the update operations have been logged, they will be propagated to a server, where they will be executed as usually (under control of the defined reconciliation component). The only difference is that, in the client, the user cannot immediately observe the expected result of the operation. Some users/applications may find this blind execution model inconvenient. In this case, the application may disable it.

This mechanism is used in a calendar coobject to allow user to schedule appointments for locally unavailable dates. To this end, the coobject has a root sub-object with an interface to schedule appointments for every date. The operations defined in this sub-object will insert/remove the appointments in the appropriate sub-objects. Therefore, as the reference to this sub-object is always available (because it is a root sub-object), users can request appointments for every date, even when no sub-object is locally cached (if the coobject can be instantiated).

For some applications, the fact that users cannot observe the effects of their *blind* invocations may be a problem. For example, in the document editor, if some user writes a new version of some uncached text leaf, he expects to be able to observe this new version (and change it during the current editing session). If the private copy of the coobject manipulated by the application does not maintain this new version, the editor has to implement this functionality using some ah hoc mechanism.

To support these situations, sub-objects can be instantiated in “replacement” mode. When some operation is executed in a sub-object that is not locally available, a “replacement” sub-object is created and the operation is locally executed in the newly created sub-object. It is important to notice that, besides this *tentative* execution, the operation is processed as before, being propagated to the server where it is integrated in the current coobject’s state.

When a coobject is defined, the programmer specifies the sub-objects that can be instantiated in “replacement” mode and the initial state of these copies (proxies are responsible to create these copies). The application may disable this default mechanism if the user does not want to use it.

In the LaTeX document coobject, users can observe the results of operations executed in sub-objects that are not locally available using this mechanism. For example, the new versions of text leaves can be observed and modified in the document editor – the editor exposes this situation to users using a different color to represent the elements.

We have implemented an additional mechanism to allow the execution of operations over coobjects that are not locally cached. The applications may submit small pieces of generic code that are executed in the server. This code is propagated to a server where it is executed in a controlled environment (it is only possible to invoke operations in the sub-objects of the target coobject). This mechanism can be used to insert appointments in a shared calendar coobject that is completely unavailable.

INTEGRATING SYNCHRONOUS COLLABORATION

In this section we describe the support to manipulate coobjects during synchronous sessions. To this end, it is necessary to be able to maintain several copies of coobjects synchronously synchronized. To achieve this property, we use a synchronous adaptation component that relies on a group communication subsystem im-

plemented on top of the DAgora event-dissemination service (Deeds) [4] (due to space limitation, we do not present any implementation details of group communication, but all used features are common in similar systems).

An application/user may start a synchronous session with the private copy of the coobject that it is being currently manipulated. The synchronous session is started replacing the current adaptation component of the coobject by a new instance of the synchronous adaptation component. This component creates a new group (in the group communication subsystem) for the synchronous session. When a new user wants to join this synchronous session, the application has to join the group for the synchronous session (using the name of the session and the name of one computer participating in the session) and obtains the current state of the coobject from a designated primary in the group (including all instantiated sub-objects and an handle to locally instantiate other sub-objects in a coherent way). Any user is allowed to leave the synchronous session at any moment.

Applications manipulate coobjects as usually, i.e., executing operations in sub-objects proxies. When an update operation is invoked, the adaptation component propagates the invocation to all elements of the synchronous session. Currently, we have implemented a simple strategy to guarantee that all replicas are kept synchronized. The adaptation component only forwards operations to be processed locally after being received (and ordered) from the group communication sub-system. Therefore, the concurrency control component (in the client) just has to execute all operations in the order they are received.

However, during synchronous sessions, it is possible to use concurrency control/reconciliation components that use optimistic approaches and are specially designed to solve synchronous conflicts (e.g. based on operational transformations [23,24]) – in this case, the adaptation component immediately forwards the operations for optimistic local execution (besides propagating them to the group).

Applications may register callbacks in the adaptation component to be notified when sub-objects are modified due to operations executed by other users – to reflect the changes of other users in the GUI of the application.

The updates executed during the synchronous session can only be saved by the designated primary. In respect to the overall evolution of the coobject, the updates executed during synchronous sessions are handled in the same way as the updates executed asynchronously by a single user. Thus, the sequence of executed operations is propagated to the servers, where it is integrated according to the reconciliation policy defined for the coobject.

In the multi-synchronous document editor, we have used the described approach to allow documents to be synchronously edited. However, in this case, this approach is not sufficient: the operations defined to manipulate a document during asynchronous editing are not sufficient for synchronous editing. In particular, synchronous editing calls for operations with small granularity (e.g., insert/remove a single character in some piece of text – represented in the LaTeX coobject as a version in a text leaf, or simply, text version), while asynchronous collaboration tends to use operations with higher granularity (e.g., set a new value to a text version). Two approaches can be used to address this problem.

First, it is possible to extend the current interfaces of sub-objects to include these new operations. This solves the problem for synchronous sessions. However, having to consider all these *small* operations during asynchronous reconciliation would be a nuisance (not only for reconciliation, but also for operation management and propagation). Fortunately, the log component implements a log-compression algorithm that compresses operations executed in clients. Therefore, groups of *small* operations used for synchronous editing can be compressed into a single *big* operation (e.g. the insert/remove operations executed in a given text version are compressed into a single operation that sets the correspondent new value to that text version). However, to use this compression mechanism the programmer needs to write the adequate compression functions for the *small* operations.

A second approach to handle operations with small granularities is to do it outside of the coobject's control. In the document editor example, this means that the editor should include a synchronous tool to allow subgroups of users to edit the text version – starting with its current value. In the end of the synchronous edition, the correspondent operation that sets a new value to the text version is executed in the coobject (thus updating all synchronous replicas of the coobject). These synchronous tools can be implemented relying on the group communication sub-system used to manage the coobject's synchronous session. This approach also allows the use of third-party tools to manage these synchronous interactions – the updates executed in these tools are integrated in the state of the coobject using one (or more) operation that summarizes all modifications. In our multi-synchronous document editor we have decided to use this approach to allow multiple users to synchronously edit the same text versions.

Discussion

For some applications, there are some inherent differences between synchronous and asynchronous collaboration that lead to differences in the support required. In synchronous collaboration, the contributions executed at each step tend to be very small (e.g. insert/remove a character). Furthermore, these contributions are propagated to other users quickly, allowing them to influence the contribution executed by the other users – the users in a synchronous session have strong awareness information about the contributions that are being executed. These properties allow the concurrency control mechanism used during synchronous sessions to be very aggressive, trying to merge all contributions that have been executed (e.g. concurrent insert/removes in the same piece of text are usually merged). In the worst case (where concurrency control fails to have the desired result), it is possible for the user to immediately solve the problem because he can immediately observe the result and the contributions involved are small (although this situation should not occur).

On the other hand, in asynchronous collaboration, the contributions executed tend to be large (e.g. the change of a section in a document). Furthermore, the users have no fine-grain awareness information about the contribution of the other users (e.g. one user may know that other user will change some section in the document, but he does not know the exact changes she will produce). Therefore, the concurrency control/reconciliation strategy used has to be much less aggressive when solving concurrent update and work at a higher granularity (e.g. merging the changes executed asynchronously to the same piece of text by two users at the level of character insertion/removal would usually lead to a result that would not satisfy anyone). Moreover, as contributions tend to be large, it is not usually acceptable for some user to loose their contributions (this is the reason to create multiple versions in the document editor).

Although it is possible to draw a continuum between the two extremes of synchronous and asynchronous work, based on how quickly the users are informed about other users' contributions, we believe that there will always be a point beyond which it is necessary to use different techniques for concurrency control. The support for awareness information also has to be adapted – while in synchronous session most awareness information can be obtained from the observation of changes executed synchronously, in asynchronous collaboration, the users usually have to be informed of the changes that have been made in the past.

In DOORS, it is possible to address the integration of synchronous sessions in the overall asynchronous activity using the approach discussed earlier (when different operations must be used in the two settings). The multi-synchronous document editor that we have implemented demonstrates the use of this approach.

When the same operations can be used adequately for synchronous and asynchronous collaboration, the basic approach described (that includes only the use of the synchronous adaptation component to maintain several coobjects synchronously synchronized) solves the problem of integrating coobejcts in synchronous session.

RELATED WORK

Several systems have been designed to manage data in large-scale distributed environments. While some of these systems are typical distributed storage system (e.g. Coda [16], Thor [10]), others have been designed

or used to support groupware applications (e.g. Lotus Notes [18], Bayou [8], BSCW [12], Prospero [7], Sync [19]). Although our system shares goals and approaches with some of these systems, it presents two distinctive characteristics. First, the object framework not only helps programmers in the creation of new applications but it also allows them to use different data-management strategies in different applications (while most of those systems only allow the customization of a single strategy). Second, most of those systems (excluding BSCW) concentrate their attention on the reconciliation problem and do not address awareness support.

Partial caching has been identified as important to support mobile computing in several systems. Oracle Lite [20] (and other database systems) allows users to cache the subset of the database they are interested on (using database queries). Some OODBs (e.g. Thor [10]) also allow users to cache only a subset of objects. Our partial caching mechanism has similar objectives. However, instead of working at the level of basic objects as it is usual in those systems, it works at the level of sets of objects grouped together using semantic information (sub-objects). Therefore, the complexity involved in the selection of objects to cache is reduced (and consequently the likelihood of incomplete caching).

Data management systems usually prevent any data access over data that is not cached. To our knowledge, the *blind* invocation mechanism that includes the instantiation of "replacement" objects is unique to our system. The most similar mechanism we are aware of is the deferred RPC mechanism in Rover [13]. It allows disconnected users to submit operations that will be propagated to the object server when connectivity is available. However, this system only allows the invocation of operations on objects that have been previously imported (cached).

Several groupware systems have presented solutions to integrate synchronous and asynchronous cooperative work. In [9] the authors define the notion of a room, where users can store objects persistently. Applications also run inside the room. A user may connect to the central server that stores the room to observe and modify the room state (using the applications that run inside the room). Users can work in a synchronous mode if they are *inside* the room at the same time. Otherwise, they work asynchronously. In [11] the authors present a hypertext authoring system that allows users to work synchronously and asynchronously. A tightly coupled synchronous session, with shared views, should be established to allow more than one user to modify the same node or link simultaneously (a locking mechanism prevents any other concurrent modification of those elements). In [22], the authors describe a distance-learning environment that combines synchronous and asynchronous work. Data manipulated during synchronous sessions is obtained from the asynchronous repository, using a simple locking or check-in/check-out model.

The previous systems do not support disconnected operation (as they all require access to a central server while using the system). Furthermore, either they do not support divergent streams of activity to occur (besides very short-time divergence during synchronous sessions) or they solve the problem through versioning. Our approach allows to integrate the updates executed during synchronous sessions in the overall asynchronous activity composed by several divergent streams of activity that are (eventually) reconciled using type-specific solutions (instead of simple versioning).

In Prospero [7], it is possible to use the concept of streams (that log the sequence of operations executed) to implement synchronous and asynchronous applications (depending on how often streams are synchronized). This mechanism can be used to implement the integration of synchronous and asynchronous sessions when the same operations can be used in both styles of cooperation. However, the authors do not address the problem of applications that need to use different operations.

FINAL REMARKS

The DOORS replicated object store provides data management support for typically asynchronous groupware in mobile environments. The system core provides high data availability relying on server replication and client caching. The system explores ad hoc networks established among clients to update local caches. The system delegates on coobjects most of the aspects related with data sharing, including concurrency con-

trol, the management of awareness information and adaptation to variable network connectivity (allowing operations to be executed in servers if good connectivity is available). The DOORS object framework allows the definition of new data types with adequate data sharing semantics composing several pre-defined solutions to the problems identified in the framework.

The system also presents other mechanisms to maximize the chance for new contributions to be produced in mobile environments. First, it implements partial caching based on the definition of coobjects as a set of sub-objects. This allows each client to cache only a subset of all sub-objects, thus reducing the storage (and communication) requirements. Second, the *blind* invocation mechanism allows disconnected users to produce contributions in data not locally cached. Replacement sub-objects may be instantiated to allow users to observe the tentative results of these contributions. To our knowledge, this feature is unique to our system and we expect that the awareness information associated with the collaborative process allow users to produce good-quality contributions.

We have also addressed the integration of synchronous sessions in the overall asynchronous activity. To this end, several copies of coobjects can be maintained synchronously synchronized using a synchronous adaptation component. The problem of using operations with different granularities for synchronous and asynchronous collaboration is discussed and our solution is presented.

We have implemented a prototype of the DOORS system in Java 2 (the pre-processor is implemented using JavaCC). Currently, we are addressing some problems that need further research. In particular, we are investigating the use of a component-oriented programming language, ComponentJ, to support the definition of coobjects.

More information about the DAgora project (including DOORS) is available from [1].

REFERENCES

1. http://dagora.di.fct.unl.pt
2. Preguiça, N., Legatheaux Martins, J., Domingos, H., Duarte, S. Data management support for asynchronous groupware. In *Proc.of CSCW'2000*, Dec. 2000.
3. Domingos, H., Preguiça, N., Legatheaux Martins, J. Coordination and Awareness Support for Adaptive CSCW Sessions. In *Proceedings of CRIWG'98*, 1998.
4. Duarte, S., Legatheaux Martins, J., Domingos, H., Preguiça, N. DEEDS - A Distributed and Extensible Event Dissemination Service. In Proc.s 4rd European Research Seminar on Advances in Distributed Systems, 2001.
5. Bluetooth. www.bluetooth.com
6. P. Dourish, V. Bellori. Awareness and Coordination in Shared Workspaces. In *Proc. of CSCW'92*, 1992.
7. Dourish, P. Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Transactions on Computer-Human Interaction*, June 1998.
8. Edwards, W., Mynatt, E., Petersen, K., Spreitzer, M., Terry, D., Theimer, M. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of UIST'97*, Oct. 1997.
9. Greenberg, S., Roseman, M. Using a room metaphor to ease transitions in groupware. Research rep. 98/611/02, Dep. Computer Science, Univ. Calgary, 1998.
10. R. Gruber, F. Kaashoek, B. Liskov, L. Shrira. Disconnected operation in the Thor Object-Oriented Database System. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, Dec.1994.
11. Haake, J., Wilson, B. Supporting Collaborative Writing of Hyperdocuments in SEPIA. In *Proceedings of CSCW'92*, 1992.
12. Horstmann, T., Bentley, R. Distributed Authoring on the Web with the BSCW Shared Workspace System. *ACM Standards View*, Mar. 1997.

13. Joseph, A., Tauber, J., Kaashoek, M., Mobile computing with the Rover toolkit. *IEEE Trans. Computers*, Mar. 1997.
14. Karsenty, A., Beaudouin-Lafon, M. An Algorithm for Distributed Groupware Applications. In *Proceedings of 13th ICDCS*, May 1993.
15. Kermarrec, A., Rowstron, A., Shapiro, M., Druschel, P. The IceCube approach to the reconciliation of diverging replicas. In *Proceedings of 20th PODC*, 2001.
16. Kistler, J., Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, Feb. 1992.
17. G. Kuenning, G. Popek. Automated Hoarding for Mobile Computers. In the Proc. of 16th SOSP, 1997.
18. Lotus Notes. http://www.lotus.com
19. Munson, J., Dewan, P. Sync: A Java Framework for Mobile Collaborative Applications. *IEEE Computer*, June 1997.
20. Oracle Lite. http://www.oracle.com
21. Pankoke-Babatz, U., Syri, A. Collaborative Worspaces for Time Deferred Electronic Cooperation. In *Proceedings of GROUP'97*, 1997.
22. Qu, C., Nejdl, W. Constructing a web-based asynchronous and synchronous collaboration environment using WebDAV and Lotus Sametime. In Proceedings on University and College Computing Services, 2001.
23. Sun, C., Ellis, C. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Prooceedings of CSCW'98*, 1998.
24. Vidot, N., Cart, M, Ferrié, J., Suleiman, M. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of CSCW'2000*, 2000.
25. IEEE 802.11. http://grouper.ieee.org/groups/802/11