# DATA MANAGEMENT PROBLEMS FOR SUPPORTING MULTI-SYNCHRONOUS GROUPWARE AND A SOLUTION

NUNO PREGUIÇA – CORRESPONDING AUTHOR
*CITI/DI, FCT, Universidade Nova de Lisboa*
*Quinta da Torre, 2845 Monte da Caparica, Portugal*

J. LEGATHEAUX MARTINS
*CITI/DI, FCT, Universidade Nova de Lisboa*
*Quinta da Torre, 2845 Monte da Caparica, Portugal*

HENRIQUE DOMINGOS
*CITI/DI, FCT, Universidade Nova de Lisboa*
*Quinta da Torre, 2845 Monte da Caparica, Portugal*

SÉRGIO DUARTE
*CITI/DI, FCT, Universidade Nova de Lisboa*
*Quinta da Torre, 2845 Monte da Caparica, Portugal*

It is common that, in a long-term asynchronous collaborative activity, groups of users engage in occasional synchronous sessions. In this paper, we analyze the data management requirements for supporting this common work practice in typical collaborative activities and applications. This analysis shows that, as users interact in different ways in each setting, some applications have different requirements and need to rely on different data sharing techniques in synchronous and asynchronous settings. We present a data management system that allows to integrate a synchronous session in the context of a long-term asynchronous interaction, using the suitable data sharing techniques in each setting and an automatic mechanism to convert the long sequence of small updates produced in a synchronous session into a large asynchronous contribution. We exemplify the use of our approach with two multi-synchronous applications.

## 1. Introduction

Groupware applications are commonly classified as synchronous or asynchronous depending on the type of interaction they support. Synchronous applications support closely-coupled interactions where multiple users synchronously manipulate the shared data. In synchronous sessions, all users are *immediately* notified about the updates produced by other users. At the data management level, it is usually necessary to maintain multiple copies of the data synchronized in realtime, merging all concurrent updates produced by the users. Several general-purpose systems have been implemented[1,2,3].

Asynchronous applications support loosely-coupled interactions where users modify the shared data without having *immediate* knowledge of the updates produced by other users. At the data management level, it is common to support a model of temporary divergence among multiple, simultaneous streams of activity[4] and to provide some mechanism to automatically merge these streams of activity. Some general-purpose (e.g. Refs. [5],[6]) and application-specific (e.g. Ref. [7] for document editors) systems have been implemented.

A common work practice among groups of individuals seeking a common goal is to alternate periods of closely-coupled interaction with periods of loosely-coupled work. During the periods of closely-coupled interaction, group elements can coordinate and create joint contributions. Between two periods of close interaction, individuals tend to produce their individual contributions in isolation.

In this paper, we address the data management problems of supporting this type of work practice in groupware applications, dubbed as multi-synchronous applications. We describe the three main mechanisms we have used to add support for synchronous sessions in the DOORS system[8], a replicated storage system designed to support asynchronous groupware.

First, a mechanism to allow applications to synchronously manipulate the data stored in the data management system. Second, a mechanism that allows to use different reconciliation and awareness techniques in each setting, as needed by some applications (e.g.: text editing systems tend to use operational transformation[9] in synchronous settings, and versioning[10,11] in asynchronous settings). Finally, a mechanism to automatically convert long sequences of synchronous operations into a small sequence of asynchronous operations. This mechanism is needed to accommodate the difference of granularity in the operations used in each setting (e.g. in text editing systems, *insert/remove character* operations are used in synchronous settings, and *update text line/paragraph/section* operations are usually used in asynchronous settings).

The remainder of this paper is organized as follows. Section 2 analyzes the requirements for supporting applications in synchronous and asynchronous settings. Section 3 discusses our design options. Section 4 present the DOORS system, detailing the integration of synchronous and asynchronous interactions. Section 5 presents multi-synchronous applications implemented in our system. Section 6 discusses related work and Section 7 concludes the paper with some final remarks.

## 2. Requirements of synchronous and asynchronous interactions

In this section we analyze the data managements requirements of synchronous and asynchronous interactions in a set of typical groupware applications. For each application, we analyze how users use the application and what data management techniques must be used.

While synchronous interactions usually last a short period of time, asynchronous interactions tend to span for very long periods. Thus, we also analyze how

to integrate the results of a synchronous interaction in a long-term asynchronous collaborative activity.

### 2.1. *Multi-user message/conferencing systems*

A conferencing system allows multiple users to communicate with each other by exchanging messages. In particular, we are interested in systems that do not restrict communication to two users.

In synchronous settings, the paradigmatic conferencing application is the chat system. This type of application has evolved from very simple text-based applications, such as the chat tools available in old UNIX systems, to recent applications (e.g. ICQ, Micosoft Messenger and Yahoo Messenger) with sophisticated interfaces, advanced management tools, and integration of new features (e.g. voice-based chats and integration with messaging systems from wireless phone networks). However, the basic functionality of chat systems have remained the same: to allow multiple users to send messages to a shared space that is visible to all other users[a]. The only operation that a user can execute is to add a message to the shared space.

The only data management requirement is to maintain, in realtime, a shared space composed by a sequence of messages. Usually, each participant maintains its own private replica of the shared space. Each new message is propagated to all participating sites using some sort of reliable group communication (either based on a centralized or on a peer-to-peer architecture). When a new message is received, it is added to the local replica — usually, it is not required that all messages are added in all replicas by the same order (causal order is usually considered sufficient, guaranteeing that a reply is always posted after the original message).

In asynchronous settings, newsgroups and message boards are the paradigmatic conferencing applications. The basic functionality of this type of application is the same of the chat systems: to allow multiple users to send messages to a shared space that is visible to all other users. Regarding the data management requirements, one major difference exists: the shared space must be stored reliably for an extended period of time even when no user is accessing the data. To this end, unlike chat systems, the data of newsgroups and message boards is usually stored in a server or group of servers that provide high data availability. Clients access these servers to read and post messages in the shared space. When the data is replicated in a group of servers, updates are usually propagated using lazy-propagation techniques[12,13] that guarantee that all replicas receive all messages.

Regarding awareness support, in synchronous tools, besides being able to immediately observe new messages posted by other users, applications often include some active mechanism (e.g. a sound or a pop-up window announces a new message) to catch the users' attention. In asynchronous tools several approaches have been used.

[a]Some chat system include other types of interactions, such as allowing an user to send a private message to another user.

Some applications simply do not have any awareness support. Other applications use passive techniques, such as highlighting unseen messages when accessing the message board. Finally, some applications use active techniques, either using email notifications or simple applications that poll for changes to the shared data.

Although synchronous and asynchronous conferencing tools have the same functionality, users tend to use them in different ways. While messages written in a synchronous tool tend to be small, each one with a small amount of information that is hard to understand outside of the context of a specific conversation, messages written in an asynchronous tool tend to be long and self-contained, often including transcripts of previous messages.

This difference complicates the integration of a synchronous and an asynchronous conferencing tool. However, we can easily imagine scenarios where this integration could be useful: for example, a chat tool could be used to discuss some post in a message board, and the transcripts of the synchronous discussion (or a summary of the discussion) could be taken as the reply to the original post. In this case, the sequence of messages posted in the synchronous interaction should be collapsed into a single message in the asynchronous interaction.

### 2.2. *Collaborative editing systems*

Collaborative editing systems allow multiple users to jointly compose and edit a shared document. In this section, we only consider structured documents composed by text: for example, a LaTeX document, an XML document or a Java source file.

Many realtime collaborative editors have been implemented in the past. In older editors (e.g. DistEdit[14]), users usually took turns at making changes (all other users could only observe the changes in realtime). This approach avoids conflicts, thus greatly simplifying concurrency control. In recent editors (e.g. Grove[9], REDUCE[15]), it is common to allow multiple users to modify the shared document concurrently.

In both cases, each participant usually maintains a copy of the shared data and all updates are propagated to all participants. In the last case, applications must also handle possible conflicts in concurrent updates. Operational transformation[9,16,17,18] has become the technique of choice in realtime editors because it ensures convergence while preserving causality and users' intentions. This technique transforms operations to guarantee that: (1) all replicas converge to the same state despite the different execution order; and (2) the users' *syntactic* intentions are preserved despite the fact that an operation may be executed in a state that is different from the state observed by the user that has executed the operation.

For supporting collaborative edition in asynchronous settings, many systems have been implemented[19,20,7,11,10]. A common model for data access is the copy-modify-merge paradigm, in which a user gets its own private copy of the document, modifies it in isolation and later uploads his changes to be merged with the modifications concurrently produced by other users. This approach has been implemented using either a centralized (e.g. CVS[11]) or a peer-to-peer architecture (e.g. Iris[7]).

Asynchronous editing systems usually merge updates produced in different regions of the document and create multiple versions for updates that modify the same region[b]. In systems that maintain the structure of the documents, the structure offers an obvious definition of a region (e.g. the leaves in documents structured as trees). In system that do not maintain the structure of the document, it is common to implicitly define a region — e.g. the popular RCS algorithm[21] defines each line as a region. Even when multiple versions are created and maintained by the underlying storage system, it is usual that the document remains syntactically consistent[6], allowing users to continue accessing the document without the need to merge the multiple versions immediately (unlike the usual approach in distributed file systems[22] that prevents any normal access before solving conflicts).

Although reconciliation in synchronous and asynchronous collaborative editing systems has the same goal (to automatically merge modifications produced concurrently), different techniques are used. To understand the reason for this difference, it is important to understand the limitations of each technique and how users interact to overcome such limitations in both settings.

It is known that operational transformation can lead to semantic inconsistencies[23,24]. The following example (from Ref. [23]) illustrates the problem. Suppose that a shared document contains the following text:

> There will be student here.

In this text there is a grammatical error that can be corrected by replacing "student" by "a student" or "students". If two users concurrently correct the error by executing different modifications (user 1 inserts the word "a" before the word "student" and user 2 inserts an "s" in the end of the word "student"), operational transformation guarantees that the syntactic intentions of each user are preserved, leading to the following text:

> There will be a students here.

However, the resulting text is semantically incorrect, as it contains a new grammatical error. Moreover, the merged version does not represent any of the users' solution and it is likely that it does not satisfy any of the users.

In synchronous settings, this problem can be easily solved as users immediately observe the modifications produced by other users. Thus, users can coordinate themselves and immediately agree on the preferred change. This is only possible because users have strong and fine-grain awareness information about the changes produced by other users. In this case, the automatic creation of multiple versions to solve conflicts would involve unnecessary complexity. Moreover, it is not clear what user interface widgets would be suitable for presenting these multiple versions.

[b]Older systems (e.g. the original Lotus Notes[13]) used to retain only the most recently produced version, but this approach was considered inappropriate for asynchronous settings where large modifications are usually produced.

In asynchronous settings, updates are not immediately merged and each asynchronous contribution tends to be large. Thus, as users have no (strong) awareness information about the modifications produced by other users, it is likely that using operational transformation to merge updates produced by different users to the same semantic unit would lead to many semantic inconsistencies. This is the main reason for not using this technique in asynchronous editing systems: it seems preferable to maintain multiple versions that are semantically correct and let users merge them later (with the possible help of merging tools), instead of maintaining a single semantically incorrect version that does not satisfy anyone. There are also some technical difficulties related with the management and execution of this technique with a very large number of operations that hamper its use in asynchronous settings — this issues has been partially addressed in Ref. [25]. These problems suggest that the granularity of operations used in asynchronous settings should be large — for example, updating the value of some part in a structured document (e.g. a section in a paper).

A system that supports synchronous and asynchronous interactions should accommodate different reconciliation techniques for synchronous and asynchronous settings. Moreover, it should handle operations with a different granularity: small, character-based, for synchronous interactions and large, region-based, for asynchronous settings. All updates produced during a synchronous interaction can be integrated in the overall asynchronous activity as one (or a small sequence of) large-grain operation.

Additionally, the system should allow different awareness techniques to be used. As it has been said, in synchronous setting users tend to require strong awareness that allows them to immediately observe the changes being produced by other users. This level of awareness support requires sophisticated user interfaces, but it can be usually implemented using only the information about the updates propagated to maintain the data synchronized. In asynchronous settings, it is often sufficient to maintain with each document a log that describes the changes produced by each user in each isolated working-session (e.g. CVS[11])[c]. Additionally, some systems (e.g. BCSCW[10]) allow users to request active notification (e.g. by email) when documents they are interested on are modified.

#### 2.2.1. *Tools for editing graphics*

Several applications for collaborative synchronous creation of graphics have been implemented[26,27,28,29,30]. Some applications use lock-based concurrency control strategies that prevent conflicts. Some more recent solutions[29] propose reconciliation techniques that automatically merge updates that do not interfere with each other and create multiple versions for updates that do interfere — for example, if

[c]This information can also be used by a sophisticated user interface to allows users to graphically identify changed areas, as in synchronous settings. However, the existence of additional information explaining the rationale of the updates can be very useful.

some object (line, square, etc.) is concurrently moved to two different locations, two objects are created. In the user interface, object versions created due to conflict are specially highlighted to allow users to differentiate these objects and solve the conflict.

There are also applications that allow users to collaboratively edit graphics in asynchronous settings[31,32]. In some of these applications[31], asynchronous interaction is limited to edit the same graphics at different times. In this case, a single stream of activity exists.

In other applications[32], several streams of activity may exist leading to divergent versions of the same document. A common approach to merge the divergent streams of activity is to define one stream of activity as the master copy and replay the updates produced in all other streams in the master copy. The simplest approach is to replay updates without trying to find out whether each update conflicts with other concurrent updates or not — in case of conflicts, this approach tends to be similar to a *last-writer wins* strategy. However, as discussed in the context of collaborative edition of text documents, this approach may be inappropriate because the overwritten work may be large and important. In this case, it is not acceptable to arbitrarily discard (or overwrite) the contribution produced by some user, and the creation of multiple versions seems preferable[33].

From this discussion, it seems that creating multiple versions in face of conflicts can be used in both synchronous and asynchronous settings. However, there are some subtle but important differences. In synchronous settings, the multiple versions are created immediately after the concurrent execution of the conflicting operations and users can observe them immediately and act accordingly – for example, by solving the conflict immediately. Moreover, the number and extent of conflicts is expected to be small as the time to propagate updates is very small (and the strong awareness information available allows users to coordinate among themselves).

In asynchronous settings, as an user may produce a long sequence of updates, it is possible that a subset of these updates conflict with updates produced concurrently by other users. For example, in a diagram composed by two green squares, an user may decide to change the color of both squares to blue and another user may decide to change their color to red. In this case, although two versions of each square should be created, only two combinations of these version seem relevant: the first including the two blue squares and the second with the two red squares. Therefore, in asynchronous settings, it seems important to provide a mechanism to manage configurations composed by versions of multiple objects[33]. This approach seems unnecessarily complex for synchronous settings.

Regarding awareness, the requirements seem similar to those of text editing tools. For synchronous setting, users must be able to immediately observe changes being produced. In asynchronous settings, users should be able to observe changes produced since they have last accessed the (graphical) document. In this case, a small summary describing the rationale of the changes can be very useful. Addi-

tionally, support for active notification of modifications also seems interesting.

### 2.3. *Group calendars*

Group calendars manage schedules for groups of individuals and resources. A large number of group calendars have been implemented in research projects[34,35] and in commercial products[36,5,32].

The typical operations include adding a new private appointment and scheduling a group meeting or reserving a resource. For scheduling a private appointment (or reserving a resource), it is only necessary to verify that the user (resource) is free for the complete period of time. For scheduling a group meeting, it is necessary that all users can attend the meeting. To guarantee the participation of all elements, it is possible to simply verify that all users are available or to require an explicit confirmation from each user. Some group calendar applications allow to specify a list of alternative time periods to increase the chance of finding a compatible time period.

A group calendar is a typical asynchronous groupware application, where each user can submit his operations without synchronous interaction with other users. Depending on the underlying system architecture, it may be even possible to submit operations during disconnected operation. When multiple replicas of the calendar exist, the system guarantees that all replicas converge to the same state.

When it is necessary to schedule a group meeting, it may be interesting to have a synchronous session with other participants to decide the best time – for example, the RTCAL application[20] provides such functionality. In the underlying group calendar, the result of a synchronous session is the scheduling of a new group meeting — if appropriate, the summary of the synchronous interaction can be stored as additional meeting information.

### 2.4. *Summary*

Table 1 and table 2 summarize the previous analysis focusing on two important characteristics: the granularity of update notification and the reconciliation techniques used. It also presents a possible strategy to integrate synchronous and asynchronous interactions. In the previous subsections we have presented the rationale for using techniques, although it might be possible to use different approaches with success.

This analysis allows us to identify some important characteristics that must be taken into account when designing a system that supports synchronous and asynchronous interactions.

First, for some applications, updates are propagated among participants using operations with a different granularity in synchronous and asynchronous modes. In synchronous settings, updates tend to be small and to be propagated as soon as a user executes some change to the shared data, thus allowing a tightly-coupled interaction with strong awareness of other users' actions. In asynchronous settings, updates tend to be large, each one including a self-contained contribution. For

| | | Conferencing system | Group calendars |
|---|---|---|---|
| synchronous | updates | technical: messages<br>social: small size | decision-making tools for time agreement<br>add/remove appointment |
| | reconciliation | causal order | merge updates using total order – alternatives for conflict resolution |
| asynchronous | updates | technical: messages<br>social: large size | add/remove appointment |
| | reconciliation | causal order | merge updates using total order – alternatives for conflict resolution |
| integrating synchronous and asynchronous | updates | compress sequence of small messages into a single long message | use decision-making log as appointment information |
| | reconciliation | use different techniques | same technique |

Table 1. Analysis of groupware applications – conferencing systems and group calendars.

| | | Editing tool for structured text document | Editing tool for object-based graphics |
|---|---|---|---|
| synchronous | updates | insert/remove character<br>add/remove element to the structure | insert/modify/remove element |
| | reconciliation | merge updates using operational transformation | merge updates using total order, create versions for solving single op. conflicts |
| asynchronous | updates | update region (e.g. section, paragraph)<br>add/remove element for document structure | insert/modify/remove element |
| | reconciliation | versioning for elements<br>merge structure ops. using total order | merge updates using total order, create versions for groups of ops. |
| integrating synchronous and asynchronous | updates | compress character ops. into a single update element op. | group sync. operation into aggregates |
| | reconciliation | use different techniques | variant of the same technique |

Table 2. Analysis of groupware applications – structured text and graphics editing tools.

supporting both types of interaction, it seems necessary to convert sequences of small updates executed in synchronous interactions into one (or a few number of) large update for use in the long-term asynchronous interactions. This requirement is best exemplified in the context of conferencing and text editing tools.

Second, for some applications, different reconciliation techniques are preferred in different modes. In synchronous settings, reconciliation can be very aggressive and merge all updates in the same data version because users have strong awareness of other users' activities and can immediately solve any problem that occurs. In contrast, in asynchronous settings, it is usually preferable to preserve all contributions from users, even if it is necessary to create multiple data versions, as these

contributions can be long.

Finally, regarding awareness, some applications also use different techniques. The difference is usually an immediate consequence of the different coupling degree. In synchronous settings, users must have immediate feedback about other users' actions. Thus, very accurate and detailed information must be constantly disseminated and presented to users. However, as users tend to be tightly coordinated, no additional information is usually needed. In asynchronous settings, users must also be able to observe the changes produced by other users since they have last accessed the shared data. However, as the coupling degree is smaller, it is often interesting to have additional information explaining the rationale of the changes (e.g. the summary associated with commit in CVS[11]). Additionally, approaches to provide active notification tend to be different: in synchronous settings, user interface widgets tend to capture the immediate attention of users; in asynchronous settings, it is common to rely on email message (or even SMS/pager messages) to provide feedback.

## 3. Design choices

In this section we present our approach to integrate synchronous interactions in an object-based system designed to support the development of asynchronous groupware applications. In this paper we only consider issues related with data management, including awareness support.

### 3.1. *Basic requirements and design choices*

We start our discussion by reviewing the basic requirements that must be addressed to support synchronous or asynchronous interactions independently.

#### 3.1.1. *Synchronous interaction*

In synchronous applications, users access and modify the shared data in realtime. To this end, a common approach is to allow several applications running on different machines to maintain replicas of the shared data. When an update is executed in any replica, it must be immediately propagated to all other replicas. To achieve this requirement, our support for synchronous replication lies on top of a group-communication infrastructure, as it is usual in synchronous groupware.

In this kind of support, it is important to allow latecomers to join an on-going synchronous session. We support this feature using a state-transfer mechanism integrated with the group-communication infrastructure.

The user interface of the synchronous application must be updated not only when the local user updates the shared data, but also whenever any remote user executes an update. To this end, our system allows applications to register callbacks for being notified of changes in the shared data. These callbacks are used to update the GUI of the application. This approach allows a synchronous application

to be implemented using the popular model-control-view pattern, with the model replicated in all participants of the synchronous session.

#### 3.1.2. *Asynchronous interaction*

In asynchronous interactions, users collaborate through the access and modification of shared data. Therefore, to maximize the chance for collaboration, it is usually important to allow users to access and modify the shared data without restrictions (except from access control restrictions). To provide high data availability, our system combines two main techniques. First, it replicates data in a set of servers to mask networks failures/partitions and server failures. Second, it partially caches data in mobile clients to mask disconnections. High read and write availability is achieved using a "read any/write any" model of data access that allows any clients to modify the data independently.

This optimistic approach leads to the need of handling divergent streams of activity (caused by independent concurrent updates executed by different users). Many different reconciliation techniques have been proposed for use in different settings and applications (e.g. the use of undo-redo[37], versioning[11], searching the best solution relying on semantic information[38]) but no single technique seems appropriate for all problems. Instead, different groups of applications call for different strategies. Thus, unlike most systems[35,11,5] that implement a single customizable strategy for reconciliation, our system allows the use of different techniques in different applications.

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[39,40]. Our system includes an integrated mechanism for handling awareness information relative to the evolution of the shared data. Different strategies can be used in different applications, either relying on explicit notification, using a shared feedback approach[39], or combining both styles. Further details on the requirements and design choices for asynchronous groupware only in mobile computing environments are presented elsewhere[8].

### 3.2. *Integrating synchronous and asynchronous interactions*

An asynchronous groupware activity tends to span over a long period of time. During this period, each participant can produce his contributions independently. Groups of participants can engage in synchronous interactions to produce a joint contribution. Thus, it seems natural to consider the result of a synchronous interaction as a contribution in the context of the long-term collaborative process. We have used this approach in our object-based system.

In the following subsections, we address the specific requirements for implementing this strategy.

3.2.1. *Updates with different granularities*

As discussed in section 2, some applications, such as conferencing systems and text editing tools, use operations with different granularities in synchronous and asynchronous settings. To address this approach, our system includes a mechanism to convert and compress the log of operations executed by users.

During a synchronous interaction, the *small* operations executed by users are incrementally converted and compressed into a small sequence of *large* operations. This sequence of *large* operations is the result of the synchronous session and it is integrated in the asynchronous collaborative process as any contribution produced by a single user. The same mechanism is used to compress the updates produced by a single user.

3.2.2. *Different reconciliation and awareness techniques*

As discussed in section 2, some applications use different reconciliation techniques in synchronous and asynchronous settings. To address this requirement, we structure data objects used in collaborative applications according to an object framework that includes independent components to handle most aspects related with data sharing, including reconciliation and awareness management. Thus, when a programmer creates a new data-type to be used in a collaborative application, he can specify different reconciliation techniques (components) to be used in synchronous and asynchronous settings.

The same approach is used for handling awareness information in different ways during synchronous and asynchronous interactions. In our system, when an operation is executed it is possible to generate specific awareness information that is processed by a component of the data object. For example, in a shared document, it may be interesting to maintain a log of modification produced over time. This log can be updated by the awareness component used in asynchronous settings. In synchronous settings, the needed awareness information is usually provided by the applications as the result of updates to the shared data. Therefore, this additional awareness information can be discarded.

An interesting aspect is the dependence between the support for using different reconciliation techniques and the support for using operations with different granularities in each setting. The reason for this lies in the fact that the reconciliation techniques used in each setting tend to expect operations with the granularity usually used in that setting. The same applies for awareness support, as the granularity of awareness information needed in each setting is closely related with the granularity of operations.

## 4. DOORS

In this section, we start by briefly presenting the DOORS system architecture and the DOORS object framework. A more detailed description, discussing support for

asynchronous groupware, can be found elsewhere[8]. Then, we detail the integration of synchronous sessions in the overall asynchronous activity.

### 4.1. *Architecture*

DOORS is a distributed object store based on an "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, such as a structured document or a shared calendar. A coobject is designed as a cluster of sub-objects, each one representing part of the complete 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 the structure). 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 will later describe each component and how they work together).

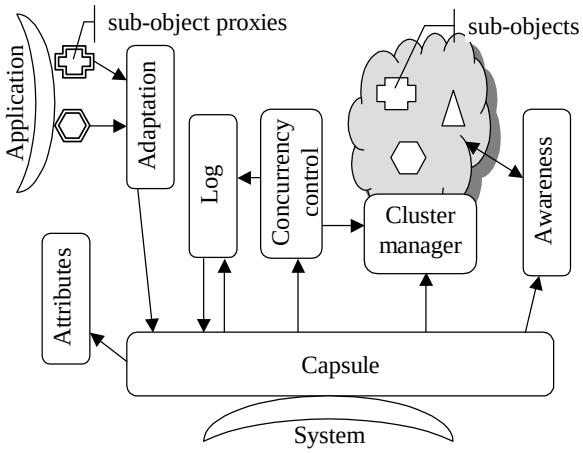


Fig. 1. DOORS object framework.

Figure 2 depicts the DOORS architecture, composed by servers and clients. Servers replicate workspaces composed by sets of related 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 while disconnected. A partial copy of a coobject includes only a subset of the sub-objects (and the operational components needed to instantiate the coobject). Clients can obtain partial replicas directly from a server or from other clients. They can also update their local copies directly from other

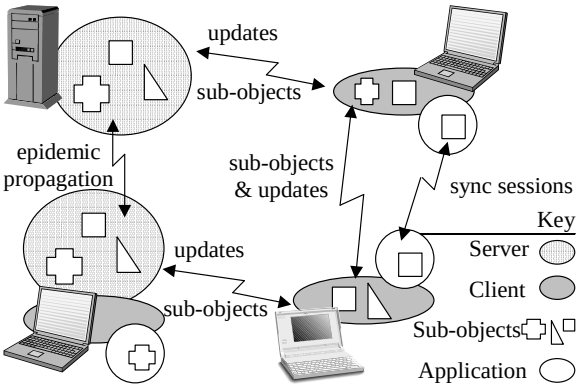


Fig. 2. DOORS architecture with four computers with different configurations. Coobjects are replicated by servers, partially cached by clients and manipulated by applications.

clients, thus exposing to users the recent contributions executed by other users.

Applications run on client machines and access data using a "get/modify locally/put changes" model. First, the application obtains a private copy of the coobject (from the DOORS client). Second, it invokes sub-objects' methods to query and modify its state – update operations are transparently logged 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.

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. Servers synchronize coobject replicas by exchanging unknown operations during pairwise epidemic synchronization sessions.

### 4.2. *DOORS Object Framework*

As outlined above, the DOORS system core executes minimal services and it delegates to the coobjects most of the aspects related with data sharing, including reconciliation and the handling of awareness information. This approach allows the implementation of flexible type-specific solutions but requires coobjects to handle several aspects that are usually managed by the system. To help programmers to create new applications reusing *good* solutions, we have defined an object framework that decomposes a coobject in several components that handle different operational aspects (see figure 1). We now outline this object framework, introducing each component in the context of the local execution of an operation.

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 (e.g. using garbage-collection).

Applications always manipulate a coobject using sub-objects' proxies. When an application invokes a method on a *sub-object proxy*, the proxy encodes the method invocation (into an object that includes all needed information) and hands it over to the adaptation component. The *adaptation component* is responsible for interactions with remote replicas. The most common adaptation component only executes operations locally.

The *capsule component* controls local execution of operations. Queries are immediately executed in the respective sub-object and the result is returned to the application. Updates are logged in the *log component*. When an operation is logged, the capsule calls the concurrency control component to execute it.

The *concurrency control/reconciliation* component is responsible to execute the operations stored in the log. In the client, operations are usually executed immediately. The result of this execution is tentative[35]. An update only affects the *official* state of a coobject when it is finally executed in the servers. In Ref. [8], we have discussed extensively how to use different reconciliation strategies (components) in the context of asynchronous groupware applications.

The execution of an operation may produce some awareness information. The *awareness component* immediately processes this information (e.g. by storing it to be later presented in applications and/or propagating it to the users).

Besides controlling operation execution, the capsule defines the coobject's composition. The composition described in this subsection represents a common coobject, but different compositions can be defined. The capsule implements the interface used by the system to access the coobject. 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 (methods) to query and to change that state. From the sub-objects' code, a pre-processor generates the code of sub-object proxies and factories used to create new sub-objects, handling the tedious details automatically. Second, he must define the coobject composition, selecting the suitable pre-defined components (or defining new ones if necessary). Different components can be specified for use in the server and in the client during private and shared (synchronous) access. Different data-sharing semantics are obtained using different components.

### 4.3. *Integration of Synchronous Sessions*

In this subsection we detail the integration of synchronous sessions in the overall asynchronous activity.

4.3.1. *Maintaining coobjects' replicas in synchronous sessions*

Each site that participates in a synchronous session usually maintains its own copy of the shared data. To this end, we need to maintain several copies of a coobject synchronously synchronized.

To achieve this goal, we use the synchronous adaptation component that propagates updates executed in any replica to all replicas. This component relies on a group communication sub-system (GCSS) – JGroups[41] in the current implementation – for managing communications among session participants.

An application (user) may *start a synchronous session* in a client when it loads a coobject from the data storage. In this case, the coobject is instantiated with the components specified for shared access in the client[d]. In particular, a version of the synchronous adaptation component must be used. This component creates a new group (in the GCSS) for the synchronous session.

When a new user wants to *join a synchronous session*, the user's application has to join the group for the synchronous session (using the name of the session and the name of one computer that participates in the session). During this process, the application receives the current state of the coobject (relying on the state transfer mechanism of the GCSS) and creates a private copy of the coobject. Any user is allowed to *leave the synchronous session* at any moment.

In each group there is a designated primary (that can change during the group lifetime). Besides being responsible to save the result of the synchronous session, the primary plays an important role in the instantiation of sub-objects. When the cluster manager of any replica needs to instantiate a new sub-object, it asks the primary to send the initial state of the sub-object (as obtained from the DOORS client) to all replicas. This approach guarantees that all replicas instantiate all sub-objects in a coherent way.

*Applications manipulate coobjects* by executing operations in sub-objects' proxies, as usual. The proxy encodes the operation and delivers it to the adaptation component for processing. Query operations are processed locally as usual. For an update operation, the adaptation component propagates the operation to all elements of the synchronous session using the GCSS (step 2 of figure 3).

The GCSS may deliver operations in the same total order or in FIFO order to all replicas. When the operation is received in (the adaptation component of) a replica, including the replica where it has been initially executed, its execution proceeds as usual (by handing the operation to the capsule for local execution, as explained in section 4.2). When total order is used, replicas are kept consistent by

[d]It is also possible to start a synchronous session using a private copy of a coobject that is being modified. In this case, the system replaces the components used for private access by the components used for shared access (when they are different). The new components are initialized with the state of the old components. To this end, we have defined an interface to export and import the relevant state of components in a generic way. If some used component does not implement this interface, it is only possible to start a synchronous session with a freshly loaded cooobject.

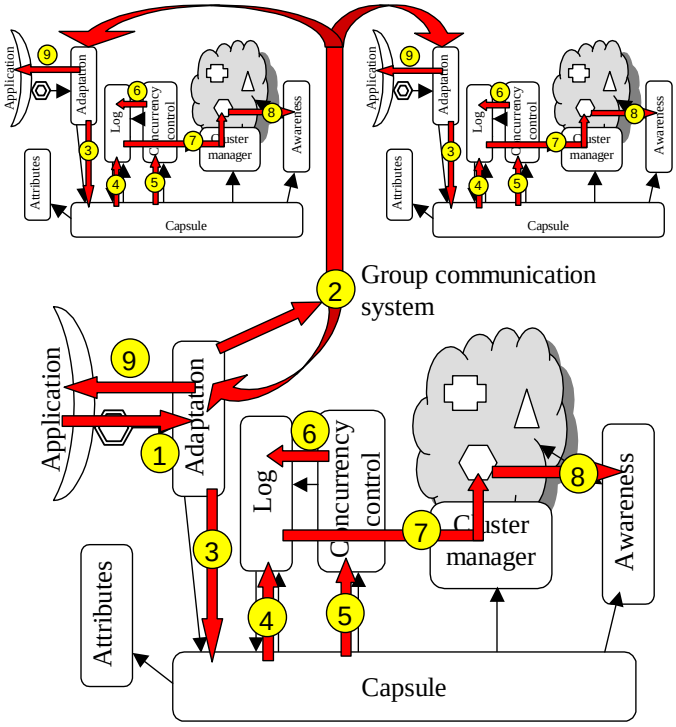


Fig. 3. Synchronous processing of an update operation in three replicas of a coobject.

simply executing all operations by the order they are received. When FIFO order is used, no delay is imposed on local operations, but replicas receive operations in different orders. Thus, it is usually necessary to use an operational transformation reconciliation component to guarantee replica convergence.

To *update the application GUI*, an application may register callbacks in the adaptation component to be notified when sub-objects are modified due to operations executed by remote users (or local users). These callbacks are called by the adaptation component when the execution of an operation ends (step 9) – the application receives information about the executed operation, including its target sub-object.

The DOORS approach to manage synchronous interactions, described in this subsection, does not imply any contact with the servers. An application running on a DOORS client can participate in a synchronous session if it can communicate with other participants using the underlying GCSS. Thus, a group of mobile clients, disconnected from all servers, may engage in a synchronous interaction even when they are connected using an ad hoc wireless network.

4.3.2. *Saving the result of a synchronous session*

As discussed in section 3.2, some applications need to convert the *small* operations used in synchronous mode into the *large* operations used in asynchronous mode.

In the DOORS system, this is achieved by the log compression mechanism implemented by the log component. As described in section 4.2, all update operations executed in a synchronous session are stored in the log before being executed. Besides the full sequence of operations, the log component also maintains a compressed version of this sequence. An operation is added to the compressed sequence after being stably executed (and after the reconciliation component executes the last undo or transformation to the operation) using the algorithm presented in figure 4. This process is executed in background to have minimal impact on the performance of the synchronous session.

```
CompressLog (seqOps: list, newOp: operation) : list =
    FOR i:= seqOps.size - 1 TO 0 DO
        IF Compress( seqOps, i, newOp) THEN RETURN seqOps
        ELSE IF NOT Commute( seqOps, i, newOp) THEN BREAK
    END FOR
    seqOps.add( ConvertToLarge( newOp))
    RETURN seqOps
```

Fig. 4. Algorithm used for log-compression.

The basic idea of the algorithm is to find out an operation already in the log that can compress the new operation (e.g. an insert/remove operation in a text element can be integrated into an operation that sets a new value to the text element by changing the value of the text). If no such operation exists, the new operation is converted into an asynchronous operation and logged (e.g. an insert/remove operation can be converted into an operation that sets a new value to the text element – the value of the text after being modified).

To use this approach, the coobject must define the following methods of the compression algorithm: *Compress*, for merging two operations; *Commute*, for testing if it is possible to execute some operation in a different log position with the same result; *ConvertToLarge*, for converting a small *synchronous* operation into a large *asynchronous* operation The examples presented in the next section show that these methods are usually simple to write.

The result of the synchronous session is the compressed sequence of operations. Only the designated primary can save the result of the session. In respect to the overall evolution of the coobject, the sequence of operations is handled in the same way as the updates executed asynchronously by a single user. Thus, the sequence of operations is propagated to the servers, where it is integrated according to the reconciliation policy that the coobject uses in the server.

4.3.3. *Using different reconciliation and awareness strategies*

As discussed in section 3.1, some applications need to use different reconciliation and awareness techniques during synchronous and asynchronous interactions. In our system, different techniques can be used by specifying that a coobject is composed by different components in the server and during shared access in the client.

The reconciliation and awareness components, defined for use during shared access, control data evolution and awareness in the synchronous session. The reconciliation and awareness components, defined for use in the servers, control behavior during asynchronous interactions, i.e., how stable replicas stored in the servers evolve and what awareness information is maintained.

## 5. Applications

In this section, we present two applications that exemplify our approach to integrate synchronous and asynchronous interactions. These applications and the DOORS prototype have been implemented in Java 2 SE.

### 5.1. *Multi-synchronous Document Editor*

The multi-synchronous document editor allows users to produce structured documents collaboratively — these documents are represented as coobjects. For example, users may use a synchronous session to discuss and create the outline of the document and to edit controversial parts. Each user may, after that, asynchronously produce his contributions editing the sections he is responsible for.

A document is a hierarchical composition of containers and leaves. Containers are sequences of other containers and leaves. A single sub-object stores the complete structure of a document, including all containers. Leaves represent atomic units of data that may have multiple versions and different data types. A sub-object that extends the multi-version sub-object stores each leaf.

For example, a LaTeX document has a root container with text leaves and scope containers. A scope container may also contain text leaves and scope containers. Scope containers can encapsulate the document structure but they have no direct association with LaTeX commands. For example, a paper can be represented as a sequence of scope elements, one for each section (see figure 5). The file to be processed by LaTeX is generated by serializing the document structure.

5.1.1. *Asynchronous edition*

During asynchronous edition, users can modify the same elements independently. The coobject maintains syntactic consistency automatically, as follows. Concurrent updates to the same text leaf are merged using the pre-defined strategy defined in its super-class: two versions are created if the same version is concurrently modified; a remove version is ignored if that version has been concurrently modified; otherwise,

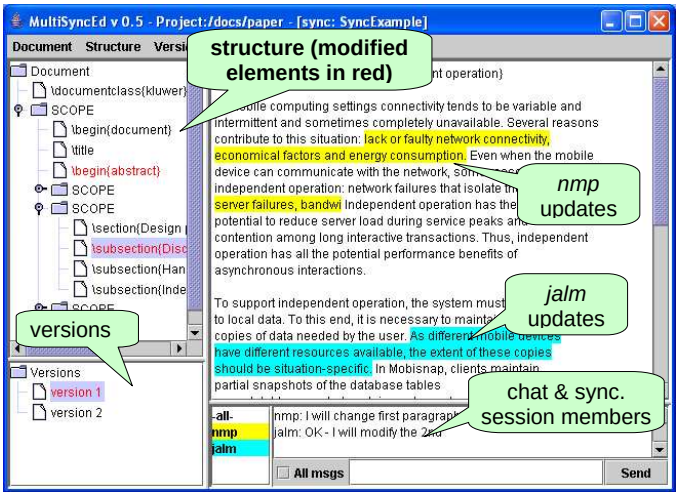


Fig. 5. Multi-synchronous document editor with a LaTeX document, while synchronously editing one section.

both updates are considered. Users should merge multiple versions later. Concurrent changes to the same container are merged by executing all updates in a consistent way in all replicas (using an optimistic total order reconciliation component in the server).

### 5.1.2. *Synchronous edition*

The multi-synchronous editor allows multiple users to synchronously edit a document. To this end, a document coobject is maintained synchronously synchronized using the synchronous adaptation component that immediately executes operations locally. Thus, users observe their operations without any delay. For handling reconciliation during a synchronous session, a reconciliation component that implements the GOTO operational transformation algorithm[23] is used.

For supporting synchronous edition, a text element also implements operations to insert/remove a string in a given version. These operations are submitted when the user writes something in the keyboard or executes a cut or paste operation. Remote changes are reflected in the editor's interface using the callback mechanism provided by the adaptation component. For example, figure 5 shows a synchronous session with two users. The selected text version presents updates from each user with a different color. In the structure and versions windows, elements that have been modified in the current session are presented in red.

5.1.3. *Saving the result of synchronous sessions*

The result of a synchronous editing session is stored as a small set of *large* operations. For converting *synchronous* operations into *asynchronous* operations, it is necessary to define the *Commute*, *Compress* and *ConvertToLarge* methods used in the compression algorithm presented in section 4.3.2.

As mentioned before, *small* operations are only defined in text leaf elements. Therefore, it is only necessary to compress these operations. In figure 6, we present the low-level operations defined in text leaf elements, grouped in *small* and *large* operations. The version identifiers used in these operations are hidden from application programmers by the objects that represent the text leaf and its versions. For example, the version object defines the operation *UpdateVersion( newText)* that internally executes the low-level operation *UpdateVersion( oldId, newId, newText)* with *oldId* the internal identifier of the version and *newId* a newly created unique identifier. The same approach is used for all other operations.

**Leaf *large* operations:**

| | |
|---|---|
| *CreateVersion( id, text)* | creates a new version with the given identifier and text |
| *DeleteVersion( id)* | deletes the version with the given identifier |
| *UpdateVersion( oldId, newId, newText)* | replaces the version *oldId* by the version *newId* with the given new text |

**Leaf *small* operations:**

| | |
|---|---|
| *InsertString( oldId, oldVersionRef, newId, pos, string)* | insert the given string in position *pos* of version *oldId* that becomes *newId* |
| *DeleteString( oldId, oldVersionRef, newId, pos, len)* | deletes *len* characters in position *pos* of version *oldId* that becomes *newId* |

Fig. 6. Operation defined in a leaf object – *small* operations are only used during synchronous sessions.

As operations on the same text elements are not necessarily executed in sequence, and may be mixed with operations on other containers and elements, the *Commute* method must be defined for all operations, allowing the compression algorithm to consider all operations on one element together. Thus, the *Commute* operation returns true for: every pair of operations that act upon different elements; and for every pair of operations that act upon the same leaf element, if they act upon different versions. Otherwise, the two operations do not commute.

The *convertToLarge* method is responsible to convert the first *small* operation into a *large* operation. In this case, *convertToLarge* only converts *InsertString* and *DeleteString* operations in the correspondent *UpdateVersion( oldId, newId, newText)* operation, with *newText* being computed by applying the defined operation to the value of the old data version.

The *Compress* method compresses *InsertString* and *DeleteString* operations executed in the resulting version of an *UpdateVersion* or *CreateVersion* operation, by updating the parameters of these operations: the *newText* is updated with the

correspondent operation and the resulting version identifier is set to the new identifier of the *small* operation. The pseudo-code of the defined methods is presented in figure 7 (for simplicity, the *Compress* method only includes compression with the *UpdateVersion* operation)[e]

```
Commute (op1: operation, op2: operation) : boolean =
  IF( op1.targetSubObj() != op2.targetSubObj())
    RETURN TRUE;
  ELSE IF( op1.targetSubObj() instanceof TextLeaf)
    RETURN NOT op1.targetVersions().overlap( op2.targetVersions());
  ELSE
    RETURN FALSE;

ConvertToLarge (op: operation) : operation =
  IF( op == "InsertString( oldId, oldVersionRef, newId, pos, string)")
    newText = oldVersionRef.getValue().insert( pos, string);
    RETURN "UpdateVersion( oldId, newId, newText)";
  ELSE IF( op == "DeleteString( oldId, oldVersionRef, newId, pos, len))"
    newText = oldVersionRef.getValue().delete( pos, len);
    RETURN "UpdateVersion( oldId, newId, newText)";
  ELSE
    RETURN op;

Compress (largeOp: operation, smallOp:operation) : boolean =
  IF(smallOp == "InsertString(oldId2,oldVRef,newId2,pos,string)" AND
      largeOp == "UpdateVersion(oldId,newId,newText)" AND oldId2 == newId)
    newText2 = newText.insert( pos, string);
    largeOp = "UpdateVersion( oldId, newId2, newText2)";
    RETURN TRUE;
  ELSE IF(smallOp=="DeleteString( oldId2, oldVRef, newId2, pos, len)" AND
      largeOp=="UpdateVersion( oldId, newId, newText)" AND oldId2 == newId)
    newText2 = newText.delete( pos, len);
    largeOp = "UpdateVersion( oldId, newId2, newText2)";
    RETURN TRUE;
  ELSE
    RETURN FALSE;
```

Fig. 7. Methods defined for log compression in the multi-synchronous text editor.

[e]Note that the parameters of the *Commute* and *Compress* methods defined in figure 7 include the two operations involved. The methods defined in the log compression algorithm presented in figure 4 receive the new operation and the log of already executed operations, being the second operation identified by its position in the log. This approach allows to easily identify the second operation, but also allow to transform the new operation to improve the change of being able to commute and compress it, if appropriate (as in operational transformation).

### 5.2. *Multi-synchronous Conferencing Tool*

The multi-synchronous conferencing tool allows to maintain an integrated repository of synchronous and asynchronous messaging interactions produced in the context of some workgroup. To this end, the application manipulates a newsgroup-like shared data space that users can use to discuss some topic. Besides allowing users to post new messages in the shared data space, the application allows to integrate conversations produced in a chat tool as posts of some discussion.

A shared space is used to discuss some topic and it may include multiple threads of discussion (see figure 8). A shared space is represented as a coobject and each thread is stored in a single sub-object (called *thread sub-object*). In each shared space, there is an additional sub-object that indexes all threads of discussion (called *index sub-object*).

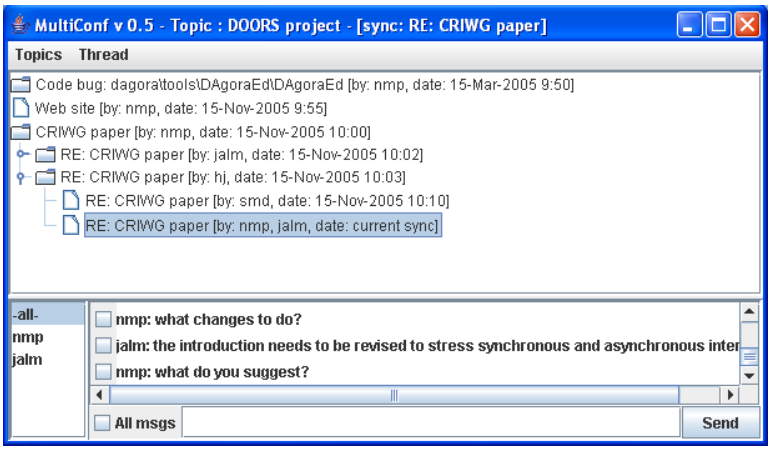


Fig. 8. Multi-synchronous conferencing tool, while a post is being synchronous edited.

#### 5.2.1. *Asynchronous mode*

In asynchronous mode, users can update the shared data space by creating new threads of discussion or posting replies to existent messages. To this end, the following operations are defined. The *thread sub-object* includes an operation to create a reply to an existent message. The *index sub-object* includes an operation to create a new thread of discussion with an initial message. The execution of this operation creates a new *thread sub-object*.

When users update the shared data space, the application immediately saves its state sending the executed operations to a server. As it is usual in DOORS, the operations are disseminated to all server replicas using a lazy epidemic propagation strategy. The following reconciliation strategy is used in the servers to handle con-

current updates: all updates are executed in all replicas using a causal order (the causal order component is used). This approach guarantees that all *reply* messages are stored in all replicas after the original message. However, it does not guarantee that all messages are stored in the same order in all replicas.

Regarding awareness, the current version of the application is very limited: it only displays the new messages with a different color. The inclusion of active notification techniques would be interesting, for example, for allowing users to be actively notified when a new message is posted to a thread the user is interested on.

#### 5.2.2. *Synchronous mode*

Our tool also allows users to maintain several replicas of a shared space synchronously synchronized. This is achieved using the synchronous adaptation component, as before. The reconciliation component executes all operations immediately in a causal order (as in the servers). During synchronous interaction, users can engage in synchronous discussions that are added to the shared space as a single reply to the original post — replies are created using a chat tool.

For supporting these synchronous interactions, the *thread sub-object* defines one additional operation: add a message to a post (that is being synchronously edited). Each message of the synchronous conversation is propagated using this operation.

Our application includes an additional operation in the *thread sub-object*: delete a message in a post. This operation was added to allow users to keep only a partial transcript of the synchronous conversation in the asynchronous discussion thread. To this end, in the end of the synchronous conversation the users are allowed to select messages to be removed from the final text (by selecting the checkboxes displayed with each message – see figure 8).

#### 5.2.3. *Saving the result of synchronous sessions*

The result of a synchronous conversation is stored as a common post in the appropriate *thread sub-object*. To this end, when an user decides to start a new synchronous discussion, it issues a *post message*. The following *add and delete message* operations are compressed into this *post message* operation as explained in the remaining of this section.

**Thread sub-object *large* operation:**

| | |
|---|---|
| *PostMessage( id, replyToId, subject, message)* | posts a reply to message *replyToId* with the given subject and message |

**Thread sub-object *small* operations:**

| | |
|---|---|
| *AddMessage( postId, msgId, text)* | adds the given message (text) to post *postId* |
| *DelMessage( postId, msgId, len)* | delete the message *msgId* from post *postId* |

Fig. 9. Operation defined in a thread sub-object – *small* operations are only used during synchronous conversations.

```
Commute (op1: operation, op2: operation) : boolean =
  IF( op1.targetSubObj() != op2.targetSubObj())
    RETURN TRUE;
  ELSE IF( op1.targetSubObj() instanceof ThreadSubObject)
    RETURN op1.targetThread() != op2.targetThread() OR
            NOT op1.targetMsgs().overlap( op2.targetMsgs());
  ELSE
    RETURN FALSE;

ConvertToLarge (op: operation) : operation =
    RETURN op;

Compress (largeOp: operation, smallOp:operation) : boolean =
  IF(largeOp == "PostMessage( id, replyToId, subject, message)" AND
      smallOp == "AddMessage( postId, msgId, text)" AND postId == id)
    newMessage = message.appendLine("["+msgId+"]: " + text);
    largeOp = "PostMessage( id, replyToId, subject, newMessage)";
    RETURN TRUE;
  ELSE IF(largeOp == "PostMessage( id, replyToId, subject, message)" AND
      smallOp == "DelMessage( postId, msgId, len)" AND postId == id)
    newMessage = message.removeStartingWith("["+msgId+"]:",len);
    largeOp = "PostMessage( id, replyToId, subject, newMessage)";
    RETURN TRUE;
  ELSE
    RETURN FALSE;
```

Fig. 10. Methods defined for log compression in the multi-synchronous conferencing tool.

Figure 9 presents the operations defined in the *thread sub-object* and figure 10 presents the methods used in the log compression algorithm that converts *small* operations into *large* operations. As in the previous example, to allow compression, it is necessary to consider operations on the same post together. To this end, the *Commute* method returns true for: any pair of operations on different threads; any pair of operations on the same thread of discussion, if they act upon different messages. Otherwise, messages do not commute.

No rule is need for converting *small* operations to *large* operations, as in the beginning of the synchronous conversation a *post message* is executed. This operation is used to compress all the following *small* messages.

The compression rules are very simple. The *add message* operation is compressed in a *post message* operation by appending the *add message* text to the end of the *post message* text. A convention is used to allow the deletion of messages – a message starts with "[msgId]", where *msgId* is a unique identifier of the message. The application interface skips this prefix when displaying messages.

The *delete message* operation is compressed in a *post message* operation by deleting the message text from the *post message* text. The start of the message to delete is identified using the message identifier.

## 6. Related Work

Several systems have been designed or used to support the development of asynchronous groupware applications in large-scale distributed settings (e.g. Lotus Notes[5], Bayou[35], BSCW[10], Prospero[6], Sync[42], Groove[32]). Our basic system shares goals and approaches with some of these systems but 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, unlike our system and BSCW, all other systems handle the reconciliation problem but do not address awareness support. From these systems, three can provide some integration between synchronous and asynchronous interactions.

In Prospero[6], it is possible to use the concept of streams (that log executed operations) to implement multi-synchronous applications (by varying the frequency of stream synchronization). This approach cannot support application that need to use different operations or different reconciliation strategies.

In Bayou, a replicated database system, the authors claim that it is "possible to support a fluid transition between synchronous and asynchronous mode of operation"[35] by connecting to the same server. However, without a notification mechanism that allows applications to easily update their interface and relying on a single replica, it is difficult to support synchronous interactions efficiently.

In Groove[32], some applications can be used in synchronous and asynchronous (off-line) modes. In Sketchpad, the same reconciliation strategy seems to be used (execute all updates by some coherent order, using a *last-writer wins* strategy). This may lead to undesired results in asynchronous interactions as the overwritten work may be large and important. In this case, it is not acceptable to arbitrarily discard (or overwrite) the contribution produced by some user, and the creation of multiple versions seems preferable[29,33].

Other groupware systems support multi-synchronous interactions. In Ref. [31], the authors define the notion of a room, where users can store objects persistently and run applications. Users work in synchronous mode if they are inside the room at the same time. Otherwise, they work asynchronously. In SEPIA[43], the authors present a multi-synchronous hypertext authoring system. A tightly coupled synchronous session, with shared views, can be established to allow multiple users to modify the same node or link simultaneously. In Ref. [44], 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.

Unlike DOORS, these systems lack support for asynchronous groupware in mobile computing environments, as they do not support disconnected operation (they all require access to a central server). Furthermore, either they do not support divergent streams of activity to occur during asynchronous edition or they use a single

reconciliation solution (versioning). Our solution is more general, allowing to use the appropriate reconciliation solutions for each setting.

In Ref. [25], the authors propose a general notification system that supports multi-synchronous interactions by using different strategies to propagate updates. They also present a specific solution for text editors that implements an operational transformation (OT) algorithm that solves some technical problems for using OT in asynchronous settings. However, as discussed in section 3.2, in asynchronous settings, OT may lead to unexpected results that do not satisfy any user – creating multiple version seems preferable. Our approach, allowing the use of a different reconciliation technique in each setting, can address this problem.

SAMS[24] is an environment that supports multi-synchronous interactions using an OT algorithm extended with a constraint-based mechanism to guarantee semantic consistency. The proposed approach seems difficult to use and, as the previous one, it does not allow to use different operations or reconciliation techniques in each setting (as it is important for supporting some applications).

In Ref. [45], the authors present a system that supports both synchronous and asynchronous collaboration using a peer-to-peer architecture to replicate shared objects. In this system, replica consistency is achieved in both settings by executing all operations in the same order – an optimistic algorithm using roll back/roll forward is used. Again, this approach does not address the need of using different operations and different reconciliation strategies in each setting.

## 7. Final Remarks

In this paper, we have proposed a model to integrate synchronous and asynchronous interactions in mobile computing environments, detailing the work presented in [46]. We have implemented the proposed approach on top of the DOORS replicated object store, that supports asynchronous groupware relying on optimistic server replication and client caching.

To integrate synchronous sessions in the overall asynchronous activity we address the three main problems identified as important in section 3. First, our system maintains multiple replicas of the data objects stored in the DOORS repository synchronized in realtime. To this end, we rely on a group communication infrastructure to propagate all operations to all replicas.

Second, our system addresses the problem of using different reconciliation and awareness strategies in different settings. To this end, the programmer may use an extension to the DOORS object framework that allows to use different reconciliation and awareness components in each setting.

Finally, it addresses the problem of using operations with different granularities for propagating updates in synchronous and asynchronous settings. To this end, it integrates a compression algorithm that converts a long sequence of *small* operations used in synchronous settings into a small sequence of *large* operations.

The combination of these mechanisms allows our system to provide support for

multi-synchronous applications – the applications presented in section 5 exemplify the use of the proposed approach. To our knowledge, our system is the only one to provide an integrated solution for all those problems in a replicated architecture that supports disconnected operation. More information about the DOORS system is available from `http://asc.di.fct.unl.pt/doors/`.

## 8. Acknowledgments

This work was partially supported by FCT/MCTES through POS_Conhecimento / FEDER.

## References

1. Mark Roseman and Saul Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(1):66–106, 1996.
2. Hyong Sop Shim, Robert W. Hall, Atul Prakash, and Farnam Jahanian. Providing flexible services for managing shared state in collaborative systems. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97)*, pages 237–252. Kluwer Academic Publishers, September 1997.
3. Christian Schuckmann, Lutz Kirchner, Jan Schümmer, and Jörg M. Haake. Designing object-oriented synchronous groupware with COAST. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, pages 30–38. ACM Press, 1996.
4. Paul Dourish. The parting of the ways: Divergence, data management and collaborative work. In *Proceedings of the European Conference on Computer-Supported Cooperative Work ECSCW'95*, pages 213–222. ACM Press, September 1995.
5. Lotus. IBM Lotus Notes. `http://www.lotus.com/notes`.
6. Paul Dourish. Using metalevel techniques in a flexible toolkit for cscw applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(2):109–155, 1998.
7. M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work*, 3(3-4):359–378, 1995.
8. Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, and Sérgio Duarte. Data management support for asynchronous groupware. In *Proc. of the 2000 ACM Conference on Computer supported cooperative work*, pages 69–78. ACM Press, 2000.
9. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of data*, pages 399–407. ACM Press, 1989.
10. R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel. Basic Support for Cooperative Work on the World Wide Web. *International Journal of Human Computer Studies: Special issue on Novel Applications of the WWW*, 46(6):827–856, 1997.
11. Per Cederqvist, Roland Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual`.
12. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth Annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.
13. L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozme, and I.Greif. Replicated document management in a group communication system. In D. Marca and G. Bock, editors, *Groupware: Software for Computer-Supported Cooperative Work*, pages 226–235. IEEE Computer Society Press, Los Alamitos, CA, 1992.

14. Michael J. Knister and Atul Prakash. DistEdit: a distributed toolkit for supporting multiple group editors. In *Proceedings of the 1990 ACM Conference on Computer-supported cooperative work*, pages 343–355. ACM Press, 1990.
15. Yun Yang, Chengzheng Sun, Yanchun Zhang, and Xiaohua Jia. Real-time cooperative editing on the internet. *IEEE Internet Computing*, 4(3):18–25, 2000.
16. Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer supported cooperative work*, pages 59–68. ACM Press, 1998.
17. Gérald Oster Abdessamad Imine, Pascal Molli and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work (ECSCW'03)*, September 2003.
18. Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM Conference on Computer supported cooperative work*, pages 171–180. ACM Press, 2000.
19. François Pacull, Alain Sandoz, and André Schiper. Duplex: a distributed collaborative editing environment in large scale. In *Proceedings of the 1994 ACM Conference on Computer supported cooperative work*, pages 165–173. ACM Press, 1994.
20. Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Information Systems (TOIS)*, 5(2):187–211, 1987.
21. Walter F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.
22. James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.
23. Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.
24. Pascal Molli, Hala Skaf-Molli, Gérald Oster, and Sébastien Jourdain. SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments. In *Proceedings of the 2002 ACM Conference on Computer supported cooperative work in design*, 2002.
25. Haifeng Shen and Chengzheng Sun. Flexible notification for collaborative systems. In *Proceedings of the 2002 ACM Conference on Computer supported cooperative work*, pages 77–86. ACM Press, 2002.
26. Dave Webster Saul Greenberg, Mark Roseman and Ralph Bohnet. Issues and experiences designing and implementing two group drawing tools. In *Proceedings of 25th Annual Hawaii International Conference on System Sciences*, pages Vol. 4: 139–150, 1992.
27. R. E. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit locking in the ensemble concurrent object-oriented graphics editor. In *Proceedings of the 1992 ACM Conference on Computer-supported cooperative work*, pages 265–272. ACM Press, 1992.
28. Dongqiu Qian and M. D. Gross. Collaborative design with NetDraw. In *Proceedings of Computer Aided Architectural Design (CAAD) Futures '99*, 1999.
29. Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(1):1–41, 2002.
30. Jeffrey D. Campbell. Usability and interference for collaborative diagram development. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work: Third Annual collaborative editing workshop*. ACM Press,

2001.
31. Saul Greenberg and Mark Roseman. Using a room metaphor to ease transitions in groupware. Technical Report 98/611/02, Department of Computer Science, University of Calgary, Alberta, Canada, January 1998.
32. Groove. Groove workspace v. 2.5. `http://www.groove.net`.
33. Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.
34. David Beard, Murugappan Palaniappan, Alan Humm, David Banks, Anil Nair, and Yen-Ping Shan. A visual calendar for scheduling group meetings. In *Proceedings of the 1990 ACM Conference on Computer-supported cooperative work*, pages 279–290. ACM Press, 1990.
35. W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th Annual ACM Symposium on User interface software and technology*, pages 119–128. ACM Press, 1997.
36. Microsoft. Microsoft outlook. `http://www.microsoft.com/outlook`.
37. Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 195–202. IEEE Computer Society Press, May 1993.
38. Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth Annual ACM Symposium on Principles of distributed computing*, pages 210–218. ACM Press, 2001.
39. Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM Conference on Computer-supported cooperative work*, pages 107–114. ACM Press, 1992.
40. Carl Gutwin and Saul Greenberg. Effects of awareness support on groupware usability. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 511–518. ACM Press/Addison-Wesley Publishing Co., 1998.
41. JGroups. `http://www.jgroups.org`.
42. Jonathan P. Munson and Prasun Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.
43. Jörg M. Haake and Brian Wilson. Supporting collaborative writing of hyperdocuments in SEPIA. In *Proceedings of the 1992 ACM Conference on Computer-supported cooperative work*, pages 138–146. ACM Press, 1992.
44. Changtao Qu and Wolfgang Nejdl. Constructing a web-based asynchronous and synchronous collaboration environment using webdav and lotus sametime. In *Proceedings of the 29th Annual ACM SIGUCCS Conference on User services*, pages 142–149. ACM Press, 2001.
45. Werner Geyer, Jörgen Vogel, Li-Te Cheng, and Michael Muller. Supporting activity-centric collaboration through peer-to-peer shared objects. In *GROUP '03: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting group work*, pages 115–124, New York, NY, USA, 2003. ACM Press.
46. Nuno M. Preguiça, José Legatheaux Martins, Henrique João L. Domingos, and Sérgio Duarte. Integrating synchronous and asynchronous interactions in groupware applications. In Hugo Fuks, Stephan Lukosch, and Ana Carolina Salgado, editors, *CRIWG*, volume 3706 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 2005.