Integrating Synchronous and Asynchronous Interactions in Groupware Applications [*]

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

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

Abstract. It is common that, in a long-term asynchronous collaborative activity, groups of users engage in occasional synchronous sessions. In this paper, we discuss the data management requirements for supporting this common work practice. As users interact in different ways in each setting, requirements and solutions often need to be different. 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 [25, 28, 26].

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. [18, 5]) and application-specific (e.g. [17] 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.

[*] This work was partially supported by FCT/MCTES through POSI/FEDER.

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 [21], 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 [8] in synchronous settings, and versioning [2, 3] 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 discusses the requirements and presents the design choices used for supporting applications in synchronous and asynchronous settings. Section 3 present the DOORS system, detailing the integration of synchronous and asynchronous interactions. Section 4 presents multi-synchronous applications implemented in our system. Section 5 discusses related work and Sect. 6 concludes the paper with some final remarks.

2 Design Options

In this section we present the design options used to integrate synchronous interactions in an object-based system designed to support the development of asynchronous groupware applications. We start by reviewing the basic requirements that must be addressed to support each type of interaction independently.

2.1 Basic Requirements and Design Options

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 and includes support for latecomers, as it is usual in synchronous groupware.

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.

Asynchronous Interaction: In asynchronous interactions, users collaborate by accessing and modifying shared data. To maximize the chance for collaboration, it is usually important to allow users to access and modify the shared data without restrictions (besides 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 network 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). Several reconciliation techniques have been proposed in different situations (e.g. the use of undo-redo [15], versioning [3], operational transformation [8, 30, 31], searching the best solution relying on semantic information [23]) but no single technique seems appropriate for all problems. Instead, different groups of applications call for different strategies. Thus, unlike most systems [7, 3, 18] that implement a single customizable strategy, our system allows different applications to use different reconciliation techniques.

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, 12]. 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 [6], or combining both styles. Further details on the requirements and design choices for asynchronous groupware in mobile computing environments are presented elsewhere [21].

2.2 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 address the specific requirements for implementing this strategy in our object-based system in the remaining of this section.

Updates with different granularities: Some applications use operations with different granularities in synchronous and asynchronous settings. For example, consider collaborative editing systems[1]. Synchronous editors (e.g. Grove [8], REDUCE [32]) allow multiple users to modify a shared document by executing operations to insert or remove a single character. These operations are immediately propagated and executed in all users' replicas. In contrast, systems for asynchronous settings (e.g.: CVS [3], Iris [17]) tend to use a copy-modify-merge

[1] Similar situations occur for other applications (e.g. conferencing systems, graphical editors), as discussed in [22].

paradigm, where reconciliation of divergent replicas is achieved by considering updates on large regions (e.g.: lines in CVS and document elements in Iris).

One reason for this situation is the difference in the level of expected awareness. In synchronous settings, users expect to have immediate knowledge about all other users' updates. Thus, all update operations must be propagated. In asynchronous settings, users are expected to work in isolation without having immediate knowledge of the modifications produced by other users. Therefore, coarse-grain updates can be propagated when a user finishes a working session.

Two additional reasons exist. The first is related with the reconciliation techniques used in each setting and it will be discussed later. The second reason is related with the technical difficulty of managing a very large number of small operations. For each operation, an excessive amount of data is created (including the type and parameters of the operation and information to order and trace dependencies among operations – the problem of reducing the information needed to trace dependencies is only partially addressed in [27]). This poses problems in terms of storage, network bandwidth and complexity of the reconciliation process.

The above reasons suggest that the granularity of operations used in each setting should be different: small for synchronous settings and large for asynchronous settings. To this end, our system includes a mechanism to compress the log of *small* operations executed by users. During a synchronous interaction, the *small* operations are incrementally converted and compressed in a small sequence of *large* operations in background. 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.

Different reconciliation and awareness techniques: In some applications, different reconciliation and awareness techniques are used in synchronous and asynchronous settings. For example, in collaborative text editors, operational transformation [8, 30, 14] has become the reconciliation technique of choice in synchronous mode while versioning [3, 18, 2] is used in asynchronous mode. 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.

It is known that operational transformation can lead to semantic inconsistencies [30, 19] when concurrent updates are executed. The following example illustrates the problem. Suppose that a document contains: *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 execute these different changes, operational transformation leads to: *There will be a students here.* The result is semantically incorrect, as it contains a new error. Moreover, the merged version does not represent any of the users' solutions 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 all concurrent modifications. 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 which user interface widgets to use for presenting these multiple versions.

In asynchronous settings, updates are not immediately merged and each contribution tends to be large. Thus, as users have no (strong) awareness information about the updates produced by other users, it is likely that using operational transformation to merge concurrent updates 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 semantically correct versions and let users merge them later, instead of a single semantically incorrect version that does not satisfy anyone.

Regarding awareness, the difference in the used techniques is an immediate consequence of the 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. In asynchronous settings, it is common that users only need to know what changes have been produced recently (and what users may be editing the document). Thus, 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 [3]).

A system that supports synchronous and asynchronous interactions should accommodate different reconciliation and awareness techniques for each settings. To this end, 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, for each data type, the programmer may specify a different technique (component) to be used in each setting.

As discussed earlier, our system allows to use operations with different granularities in each setting by automatically converting the operations. This approach is important for reconciliation as the techniques used in each setting expect operations with different granularities. It is also important for awareness support, as the granularity of awareness information needed in each setting is closely related with the granularity of operations. In our system, the awareness component handles the awareness information produced when an operation is executed.

3 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 in [21]. Then, we detail the integration of synchronous sessions in the overall asynchronous activity.

3.1 Architecture

DOORS is a distributed object store based on an "extended client/replicated server" architecture. It manages coobjects: objects structured according to the

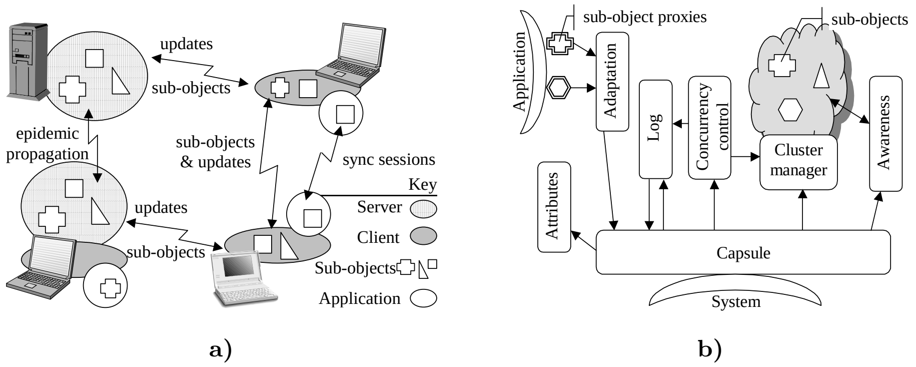


Fig. 1. DOORS architecture (a) with four computers with different configurations. Coobjects are replicated by servers, partially cached by clients and manipulated by applications. Coobjects are structured according to the DOORS object framework (b).

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 — Fig. 1.b depicts the approach (we describe each component and how they work together later).

Figure. 1.a 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.

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. 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.

3.2 DOORS Object Framework

As outlined above, the DOORS system core executes minimal services and it delegates on the coobjects most of the aspects related with data sharing, including reconciliation and the handling of awareness information. 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 Fig. 1.b). 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.

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 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 [7]. An update only affects the *official* state of a coobject when it is finally executed in the servers. In [21], 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.

3.3 Integration of Synchronous Sessions

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

Manipulate coobjects 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 [1] 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. 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 Fig. 2).

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 Sect. 3.2). When total order is used, replicas are kept consistent by 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 operation in different order. 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 op-

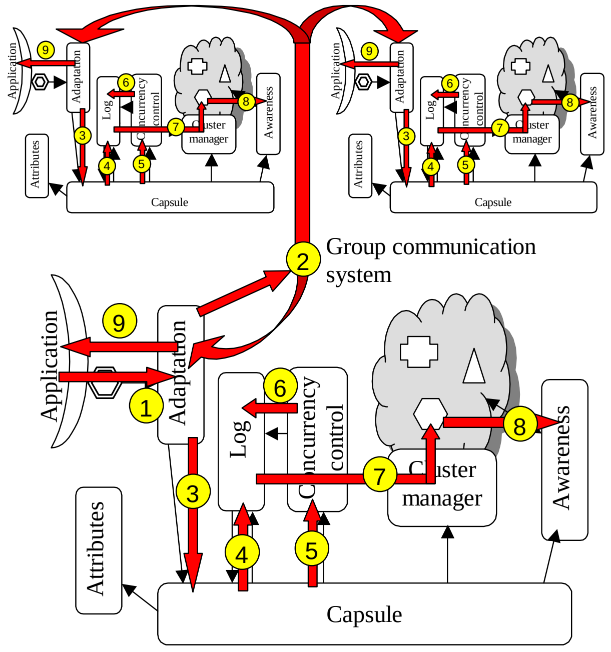


Fig. 2. Synchronous processing of an update operation in three replicas of a coobject.

erations 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 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.

Saving the result of a synchronous interaction as an asynchronous contribution: As discussed in Sect. 2.2, 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 Sect. 3.2, 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 Fig. 3. This process is executed in background to have minimal impact on the performance of the synchronous session.

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

Fig. 3. 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 the result of executing two operations does not depend on the execution order; *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.

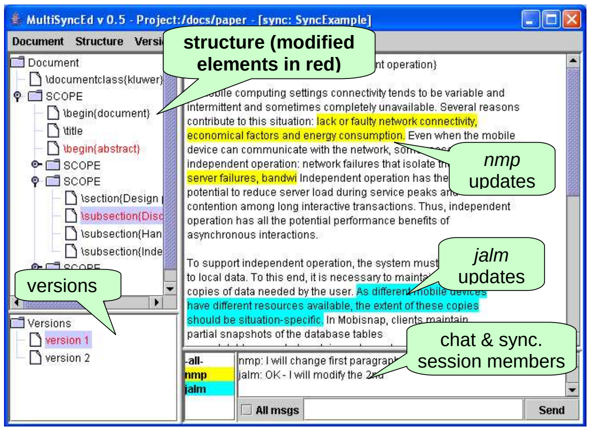
The result of the synchronous session is the compressed sequence of operations. Only the designated primary can save the result of the session. In respect to the overall evolution of the coobject, the sequence of operations is handled in the same way as the updates executed asynchronously by a single 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.

Using different reconciliation and awareness strategies: As discussed in Sect. 2.1, 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.

4 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.



nt operation}

bile computing settings connectivity tends to be variable and intermittent and sometimes completely unavailable. Several reasons contribute to this situation: lack or faulty network connectivity, economical factors and energy consumption. Even when the mobile device can communicate with the network, som independent operation: network failures that isolate th server failures, bandwi Independent operation has the potential to reduce server load during service peaks an contention among long interactive transactions. Thus, independent operation has all the potential performance benefits of asynchronous interactions.

To support independent operation, the system must to local data. To this end, it is necessary to mainta copies of data needed by the user. As different mobile devices have different resources available, the extent of these copies should be situation-specific. In Mobisnap, clients maintain partial snapshots of the database tables

nmp: I will change first paragrap
jalm: OK - I will modify the 2nd

Fig. 4. Multi-synchronous document editor with a LaTeX document, while synchronously editing one section.

4.1 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.

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 Fig. 4). The file to be processed by LaTeX is generated by serializing the document structure.

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, 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).

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 [30] 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, Fig. 4 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.

For converting *synchronous* operations into *asynchronous* operations, the following rules are used. Operations *commute* if they act upon different structure elements or different versions. Otherwise, they do not commute. The update version operation *compresses* insert/remove string operations — the new value of the version is updated to reflect the insert/remove operations. No other compression rule is needed for converting a synchronous session into an asynchronous contribution[2]. An insert/remove operation can be *converted to a large* update version operation, where the new value of the version is the result of applying the given operation to the current state of the version.

4.2 Multi-synchronous Conferencing Tool

In this section we describe a *conferencing* tool that allows to integrate discussions produced in a chat tool as posts in a message board, thus allowing to maintain an integrated repository of synchronous and asynchronous messaging interactions produced in the context of some workgroup.

This application maintains a newsgroup-like shared space where users can post messages asynchronously. A shared space is used to discuss some topic and it may include multiple threads of discussion. A shared space is represented as a coobject and each thread is stored in a single sub-object. In each shared space, there is an additional sub-object that indexes all threads of discussion.

Two operations are defined: create a new thread of discussion with an initial message and post a (reply) message to an existing thread. The following reconciliation strategy is used in the servers: all updates are executed in all replicas using a causal order. This approach guarantees that all *reply* messages are stored in all replicas before the original message, but it does not guarantee that all messages are stored in the same order – this is usually considered sufficient in this context.

[2] Additional compression rules are applied as part of the normal log compression mechanism: create/delete version pairs are removed; add/remove element pairs are removed; an update version replaces a previous update version.

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.

The thread sub-object defines an additional operation for synchronous interactions: add a message to a previous message. When the user decides to start a new discussion, it issues a *post message*. This initial *post message* operation compresses all following *add message* operations issued in the synchronous discussion (by including the new messages). In this case, the other rules needed for log compression are very simple: two operations, a and b, commute if they neither modify the same message nor b posts a reply to the message posted by a, or vice-versa; no rule is need for converting operations as all add messages are compressed into the initial post message.

5 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 [18], Bayou [7], BSCW [2], Prospero [5], Sync [20], Groove [11]). 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 [5], 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" [7] 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 [11], 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, 16].

Other groupware systems support multi-synchronous interactions. In [10], 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 [13], 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 [24], 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 [27], 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 Sect. 2.2, 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.

In [19], the authors present a brief overview of SAMS, 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 [9], 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.

6 Final Remarks

In this paper, we have presented a model to integrate synchronous and asynchronous interactions in mobile computing environments. Our approach is built 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 the discussion of

Sect. 2. 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 Sect. 4 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/dagora/`. DOORS code is available on request.

References

1. JGroups. `http://www.jgroups.org`.
2. 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. *Int. Journal of Human Computer Studies*, 46(6):827–856, 1997.
3. P. Cederqvist, R. Pesch, et al. Version Management with CVS. `http://www.cvshome.org/docs/manual`.
4. P. Dourish. The parting of the ways: Divergence, data management and collaborative work. In *Proc. of the European Conf. on Computer-Supported Cooperative Work (ECSCW'95)*, 1995.
5. P. Dourish. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM Trans. on Computer-Human Interaction (TOCHI)*, 5(2):109–155, 1998.
6. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proc. of the 1992 ACM Conf. on Computer-supported cooperative work*, 1992.
7. W. Edwards, E. Mynatt, K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proc. of the ACM Symp. on User interface software and technology*, 1997.
8. C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of data*, 1989.
9. W. Geyer, J. Vogel, L.-T. Cheng, and M. Muller. Supporting activity-centric collaboration through peer-to-peer shared objects. In *Proc. of the 2003 ACM Conf. on Supporting group work (GROUP '03)*, 2003.
10. S. Greenberg and M. Roseman. Using a room metaphor to ease transitions in groupware. Tech. Report 98/611/02, Dep. Comp. Science, Univ. of Calgary, 1998.
11. Groove. Groove Workspace v. 2.5. `http://www.groove.net`.
12. C. Gutwin and S. Greenberg. Effects of awareness support on groupware usability. In *Proc. of the Conf. on Human factors in computing systems*, 1998.

13. J. Haake and B. Wilson. Supporting collaborative writing of hyperdocuments in SEPIA. In *Proc. of the ACM Conf. on Computer-supported cooperative work*, 1992.
14. A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proc. of the 8th European Conf. on Computer-Supported Cooperative Work (ECSCW'03)*, Sept. 2003.
15. A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. of the* 13th *Int. Conf. on Dist. Computing Systems*, 1993.
16. R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.
17. M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work*, 3(3-4):359–378, 1995.
18. Lotus. Ibm lotus notes. `http://www.lotus.com/notes`.
19. P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain. SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments. In *Proc. of the 2002 ACM Conf. on Computer supported cooperative work in design*, 2002.
20. J. P. Munson and P. Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.
21. N. Preguiça, J. L. Martins, H. Domingos, and S. Duarte. Data management support for asynchronous groupware. In *Proc. of the 2000 ACM Conf. on Computer supported cooperative work*, 2000.
22. N. Preguiça, J. L. Martins, H. Domingos, and S. Duarte. Integrating synchronous ans asynchronous interactions in groupware applications. Technical Report TR-01-2005 DI-FCT, Univ. Nova de Lisboa, 2005.
23. N. Preguiça, M. Shapiro, and C. Matheson. Semantic-based reconciliation for collaboration in mobile environments. In *Proc. of the* 11th *Conf. on Cooperative Information Systems (CoopIS) - LNCS 2888*. Springer, 2003.
24. C. Qu and W. Nejdl. Constructing a web-based asynchronous and synchronous collaboration environment using webdav and lotus sametime. In *Proc. of the* 29th *ACM SIGUCCS Conf. on User services*, 2001.
25. M. Roseman and S. Greenberg. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Trans. on Computer-Human Interaction (TOCHI)*, 3(1):66–106, 1996.
26. C. Schuckmann, L. Kirchner, J. Schümmer, and J. M. Haake. Designing object-oriented synchronous groupware with coast. In *Proc. of the 1996 ACM Conference on Computer supported cooperative work*, 1996.
27. H. Shen and C. Sun. Flexible notification for collaborative systems. In *Proc. of the 2002 ACM Conf. on Computer supported cooperative work*, 2002.
28. H. S. Shim, R. W. Hall, A. Prakash, and F. Jahanian. Providing flexible services for managing shared state in collaborative systems. In *Proc. of the* 5th *European Conf. on Computer Supported Cooperative Work (ECSCW'97)*, 1997.
29. C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. on Comp.-Human Interaction (TOCHI)*, 9(1), 2002.
30. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. on Comp.-Human Interaction (TOCHI)*, 5(1), 1998.
31. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the 2000 ACM Conf. on Computer supported cooperative work*, 2000.
32. Y. Yang, C. Sun, Y. Zhang, and X. Jia. Real-time cooperative editing on the internet. *IEEE Internet Computing*, 4(3):18–25, 2000.