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 analyze the requirements for supporting this common work practice in typical collaborative activities and applications. This analysis shows that, for some applications, it is necessary 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. We exemplify the use of our system 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 that allow multiple users to synchronously manipulate the shared data. During synchronous manipulation, all users are *immediately* notified about the updates produced by other users. At the data management level, it is 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 [37, 41, 39].

Asynchronous applications support loosely-coupled interactions that allow users to modify the shared data without having *immediate* knowledge of the modifications that are being or have been produced by other users. At the data management level, it is usually necessary to support a model of temporary divergence among multiple, simultaneous streams of activity [9] and to provide some mechanism to automatically merge these streams of activity. Some general-purpose (e.g. [27, 10]) and application-specific (e.g. [26] 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, individuals can

* This work was partially supported by FSE.

coordinate themselves and create joint contributions. These closely-coupled periods may involve all elements of the group or smaller subgroups. Between two periods of close interaction, individuals tend to work in a loosely-coupled way, by producing their individual contributions in isolation.

In this paper, we address the problem of supporting this type of work practice by integrating synchronous and asynchronous support in a single platform that can be used in a mobile computing environment. To this end, we have added support for synchronous sessions in the DOORS system [33], a replicated storage system designed to support asynchronous groupware. DOORS manipulates coobjects: data objects structured according to the DOORS object framework, allowing different data-sharing strategies to be used by different applications.

To support synchronous sessions, the following mechanisms have been implemented. First, it has been added support for manipulating coobjects during synchronous sessions: several replicas of a coobject can be maintained synchronized in realtime. Second, relying on the object framework, a different reconciliation technique can be used in each setting for the same coobject. This property is important for some applications (e.g. in a text editor, operational transformation can be used to in synchronous mode, and versioning can be used in asynchronous mode). Finally, we have added support to use different operations in each setting. This property is also important for some applications (e.g. in a text editor, *insert/remove character* operations are used in synchronous settings, and *update text element/region* 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 two applications that exemplify the use of our approach. Section 6 discusses related work and section 7 concludes the paper with some final remarks.

2 Analyzing synchronous and asynchronous requirements

In this section we analyze the data managements requirements of synchronous and asynchronous interactions using 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 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 Messengerand Yahoo Messenger) with sophisticated interfaces, advanced management tools, and integration of new features (e.g. integration with messaging systems from wireless phone networks).

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[1]. 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 sufficient).

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 and 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 [8, 20] that guarantee eventual consistency.

Although synchronous and asynchronous conferencing tools have the same functionality (and it is possible to extend a message board system that uses a

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

single server to be used in a synchronous setting), 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 it 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 has 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 [25]), 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 synchronous editors (e.g. Grove [14], REDUCE [47]), it is common to allow multiple users to modify the shared document concurrently.

In both cases, each participant maintains a copy of the shared data and all updates are propagated to all participants. In the last case, applications must also handle updates that may conflict with other concurrent updates. Operational transformation [14, 43, 2] have 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 [32, 16, 26, 7, 5]. A common model for data access is the copy-modify-merge paradigm, in which a users gets its own private copy of the document, modify it in isolation and later his changes are merged with the mod-

ifications produced by other users. This approach has been implemented using either a centralized (e.g. CVS [7]) or a peer-to-peer architecture (e.g. Iris [26]).

Asynchronous editing systems usually merge updates produced in different regions of the document and create multiple versions for updates that modify the same region[2]. 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 [45] 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 [10], allowing users to continue accessing the document without the need to merge the multiple versions immediately (unlike the usual approach in distributed file systems [24] 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 [44, 29]. The following example (from [44]) 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 corrections (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.

[2] Older systems (e.g. the original Lotus Notes [20]) 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 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.

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 — these problems have been partially addressed in [40]. 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.

Graphics editing Several applications for collaborative synchronous edition of graphics have been implemented [38, 31, 35, 42, 6]. Some applications use lock-based concurrency control strategies that prevent conflicts. Some recent solutions [42] 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 some object (line, square, etc.) is concurrently moved to two different locations, two objects are created. In the user in-

terface, 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 [15, 17]. In some of these applications [15], asynchronous interaction is limited to edit the same graphics at different times. In this case, a single stream of activity exists.

In other applications [17], 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 if each update conflicts with other updates that have been concurrently executed — in case of conflicts, this approach is 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 [22].

From the above 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 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 objects [22]. This approach is unnecessarily complex for synchronous settings.

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 [4, 13] and in commercial products [28, 27, 17].

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, 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 [16] 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 summarizes the previous analysis focusing on two important characteristics: the granularity of update notification and the reconciliation techniques used. We also present a strategy to integrate synchronous and asynchronous interactions. In the previous subsections, we have made the case for the use of these techniques, although it is possible to use different approaches with success.

This analysis allows 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

		Conferencing system	Multi-user editing tool with structured document	Group calendars
synchronous	updates	technical: messages social: small size	insert/remove character add/remove element to the structure	decision-making tools for time agreement add/remove appointment
	reconciliation / concurrency control	causal order	operational transformation for elements merge structure ops. using total order	merge updates using total order – alternatives for conflict resolution
asynchronous	updates	technical: messages social: large size	update region (e.g. section, paragraph) add/remove element for document structure	add/remove appointment
	reconciliation / concurrency control	causal order	versioning for elements merge structure ops. using total order	merge updates using total order – alternatives for conflict resolution
integrating synchronous and asynchronous	updates	convert sequence of small messages into a single long message	compress character ops. into a single update element op.	use decision-making log as appointment information
	reconciliation / concurrency control	use different techniques		same technique

Table 1. Analysis of synchronous and asynchronous applications.

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 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.

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 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.

3 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. In this paper we only consider issues related with data management.

3.1 Basic requirements and design options

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

Synchronous interaction In synchronous applications, users access and modify the shared data in realtime. To this end, the system must 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.

Asynchronous interaction In asynchronous interactions, users collaborate through the access and modification of shared data. Therefore, one fundamental requirement to maximize the chance for collaboration is to allow users to access and modify 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 in different settings (e.g. the use of undo-redo [21], versioning [7], operational transformation [14, 43, 46], 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 [13, 7, 27] 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 [11, 18]. 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 [11], or combining both styles. Further details on the requirements and design choices for asynchronous groupware in mobile computing environments are presented elsewhere [33].

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 specific requirements for implementing this strategy.

Updates with a different granularity As discussed in the previous section, in some applications, updates are propagated between replicas using operations with a different granularity for synchronous and asynchronous interactions. To address this problem, our system includes a mechanism to compress the log of operations submitted by users.

During a synchronous interaction, the *small* operations executed by users are incrementally converted and compressed in 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.

Different reconciliation techniques As discussed in the previous section, in some applications, it is interesting to use different reconciliation techniques in different setting to handle updates executed concurrently. To address this problem, 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 data type to be used in a collaborative application, she can specify different reconciliation techniques to be used in synchronous and asynchronous settings.

The same approach can be 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.

4 DOORS

In this section, we present the DOORS system. We start by briefly presenting the system architecture and the object framework. A more detailed description of these elements of the system, and how they can be used to support only asynchronous groupware can be found in [34]. Then, we detail the mechanisms used for integrating synchronous sessions in the overall asynchronous activity.

4.1 Architecture

DOORS is a distributed object store based on a ”extended client/replicated server” architecture. It manages coobjects: objects structured according to the DOORS object framework. A coobject represents a data type designed to be shared by multiple users, 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 describe each component and how they work together later). Sets of related coobjects are grouped in volumes that represent collaborative workspaces and store the data associated with some workgroup and/or collaborative project.

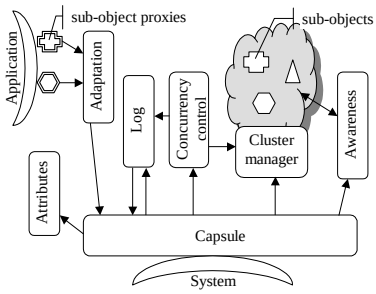

Fig. 1. DOORS object framework.

The DOORS architecture is composed by servers and clients, as depicted in figure 2. Servers replicate volumes of coobjects to mask network failures/partitions and server failures. Server replicas are synchronized during pairwise 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 use a ”get/modify locally/put changes” model of data access. First, the application obtains a private copy of the coobject (from the DOORS client). Second, the application invokes subobjects' methods to query and modify its state (as it would do with common objects). The update operations are transparently logged (and compressed) in the coobject. Finally, if the user chooses to save her changes, the logged sequence of operations is (asynchronously) propagated to a server.

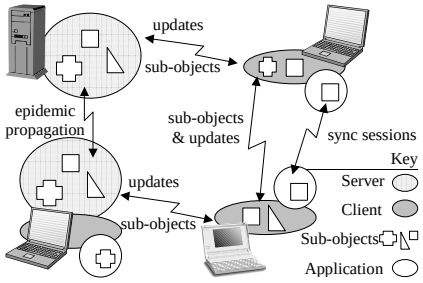

Fig. 2. DOORS architecture composed by four computers with different configurations. Coobjects are replicated by servers, partially cached by clients and manipulated by users' applications.

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

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

4.2 DOORS object framework

The outlined design imposes a heavy burden on coobjects, which must handle several aspects that are usually managed by the system. To alleviate programmers from much of this burden and to allow the reuse of *good* solutions in multiple data types, we have defined an object framework that decomposes a coobject in several components that handle different operational aspects (see figure 1). In this subsection we outline the complete object framework, introducing each component in the context of the local execution of an update 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 coobjects' data through sub-objects' proxies. When an application invokes a method on a **sub-object proxy**, the proxy encodes the method invocation (into a simple object that includes information to trace its causal dependencies and to order them) 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. But other implementations are available, allowing to execute operations in a server or to change the execution location depending on the connectivity.

Local execution is controlled by the **capsule component**. Query operations are executed immediately in the respective sub-object and the result is returned to the application. Update operations are logged in the **log component**. When-

ever 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 (showing the expected result) [13], as an update only affects the *official* state of a coobject when it is finally executed in the servers. To guarantee that all (server) replicas evolve in a consistent way and that users intentions are respected, different concurrency control/reconciliation components implementing different strategies may be used in the server (this problem is discussed extensively in [33]).

During the execution of the operations some awareness information may be produced. This information is handed over to the **awareness component** that immediately processes it (storing it to be later presented in applications and/or propagating it to the users using the systems' notification services).

Besides controlling the local execution of operations, the capsule component defines the coobject's composition and aggregates its components. The composition described in this subsection represents a common coobject, but it is possible to define different compositions — for example, it is possible to maintain a tentative and a committed version of the sub-objects using two different reconciliation components to execute updates stored in a single log using an optimistic and a pessimistic total order strategy respectively. The capsule component also defines the interface with the system for exposing the logged operations and processing the operations received during epidemic synchronization sessions. Finally, the **attributes component** stores the system and type-specific properties of the coobject.

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

Second, the programmer must define the coobject composition selecting the adequate 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. As these components encode most of the data-sharing semantics, different data-sharing approaches can be obtained using different pre-defined components.

4.3 Integration of synchronous sessions

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

Manipulate coobjects in synchronous sessions: As we have seen in section 2, each site that participates in a synchronous section 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 an adaptation component (called synchronous adaptation component) that propagates updates executed in any replica to all replicas that participate in the synchronous session. This component relies on a group communication sub-system for managing communications among participants of the synchronous (GCSS) session (two different implementations exist, one relying on JGroups [1] and other relying on the Deeds event-dissemination system [12]).

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[3]. In particular, the adaptation component must be a version of the synchronous adaptation component. 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 the entrance process, the application receives the current state of the coobject (including all instantiated sub-objects) from a designated primary in the group (relying on the state transfer mechanism of the GCSS). A private copy of the coobject is created (instantiated) using the received state. Any user is allowed to **leave the synchronous session** at any moment.

In DOORS, the sub-objects are only instantiated by the cluster manager when they are accessed. When a coobject is being manipulated in a synchronous session, the initial state of a sub-object is obtained from the designated primary

[3] 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 starts by replacing 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.

in the group to guarantee that all replicas instantiate all sub-objects in a coherent way[4].

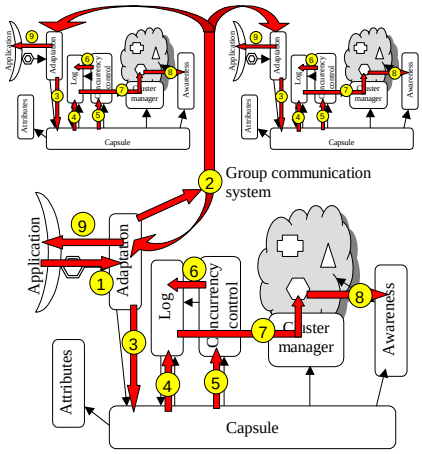

Fig. 3. Synchronous processing of an update operation in three replicas of a coobject.

Applications manipulate coobjects as usual, i.e., by executing operations in sub-objects' proxies. The proxy encodes the operation and delivers it to the adaptation component for processing. Query operations are processed locally as usual (see details in section 4.2).

For an update operation, the adaptation component propagates the operation to all elements of the synchronous session, as depicted in step 2 of figure 3. The GCSS delivers all operations in the same order in all replicas. When the operation is received in (the adaptation component of) a replica, including the replica where the operation 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). Using this approach, it is very simple to maintain all replicas consistent: as all update operations are received in all replicas in the same order, the concurrency control component just has to execute all operations in the order they are received[5].

[4] Whenever a replica needs to instantiate a sub-object, it sends a request to the group. The primary replies to this request by sending the initial state of the sub-object (as obtained from the DOORS client) to all replicas — all replicas cache the initial state of the sub-object for future use.

[5] Note that even using this approach it is possible to preserve users' intentions executing operational transformation techniques before executing an update.

An alternative approach has also been implemented: the operation is propagated using a FIFO order (only guaranteeing per-sender ordering). This order guarantees that the operation is delivered immediately in the local replica, thus imposing no delay on local execution of operations (local execution proceeds as usual). In this case, as operations are delivered in different orders in different replicas, it is usually necessary to rely on operational transformation to guarantee that all replicas remain consistent.

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 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 section 3.1, for some applications, it is necessary to convert the *small* operations used during synchronous interaction into the *large* operations used for asynchronous interaction.

```
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. 4. Algorithm used for log-compression.

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 applied to the local replicas of the sub-objects. 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 just before being

stably executed (and after the reconciliation component executes the last undo or transformation to the operation) using the algorithm presented in figure 4.

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 added to the sequence of operations in the log (e.g. an insert/remove operation in a text element can be converted into an operation that sets a new value – the current value of the text modified by the operation – to the text element).

To use this approach it is necessary to define the following methods used in 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* operations — this operation has access to the current state of the coobject. 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 stored in the log. Only the designated primary can save the result of the session. In respect to the overall evolution of the coobject, the compressed sequence of operations that is the result of the synchronous sessions is handled in the same way as the updates executed asynchronously by a single user. Thus, the sequence of executed operations is propagated to the servers, where it is integrated according to the reconciliation policy that the coobject uses in the server.

Using different reconciliation strategies: As discussed in section 3.1, for some applications, it is important to use different reconciliation techniques during synchronous and asynchronous interactions. In the DOORS system, it is possible to use different techniques by specifying that a coobject is composed by different components in the server and during shared access in the client.

The reconciliation component used during shared access in the client controls how updates are applied to each replica maintained by the participants of the synchronous session. Thus, this component defines the reconciliation strategy for synchronous interaction.

The reconciliation component used in the server controls how updates are applied to the stable replicas maintained by the servers. Thus, this component defines the reconciliation strategy for asynchronous interactions.

A similar approach is used for the awareness component, allowing to use different approaches to handle information created during synchronous and asynchronous interactions.

5 Applications

In this section, we present two applications that exemplify our approach to integrate synchronous and asynchronous interactions. These applications have been implemented in Java 2. The DOORS prototype has also been implemented in Java 2 (the pre-processor is implemented using JavaCC).

5.1 Multi-synchronous document editor

The multi-synchronous document editor allows users to produce structured documents collaboratively — these documents are represented as coobjects. A document is a hierarchical composition of two types of elements: containers and leaves. Containers are sequences of other containers and/or leaves. The complete structure of a document, including all containers, is stored in a single subobject. Leaves represent atomic units of data that may have multiple versions and that may be of different types. Each leaf is represented by a sub-object.

For example, a LaTeX document has a root container that may contain a sequence of text leaves and/or scope containers. A scope container may also contain a sequence of text leaves and/or scope containers. There is no direct association between these elements and LaTeX commands/elements. Users are expected to use scope elements to encapsulate the document structure. For example, a paper can be represented as a sequence of scope elements, each one containing a different section (see figure 5). The file to be processed by LaTeX is generated by serializing the document structure - all text is contained in text leaves.

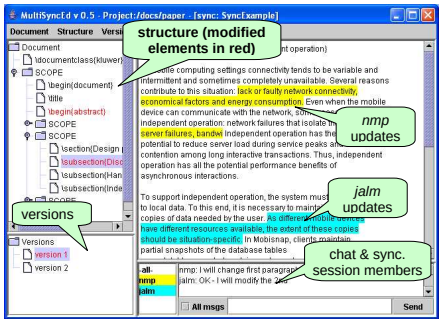

Fig. 5. Multi-synchronous document editor with a LaTeX document, while synchronously editing one section.

Asynchronous edition: When editing the document asynchronously, users are allowed to change the same elements independently. The coobject manages concurrent modifications automatically, maintaining syntactic consistency, as follows. Concurrent modifications to the same text leaf are merged using the pre-defined strategy implemented in the multi-version sub-object (text leaves are defined as a sub-type of this sub-object): two versions are created if the same version is concurrently modified; a remove version is ignored if that version has been concurrently modified; otherwise, both updates are considered. Users should merge multiple versions into a single version later. Concurrent modifications to the same container are merged executing all updates in a consistent way in all replicas (using an optimistic total order reconciliation component 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 [43] 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. Updates to the structure, versions or current contents of a text version executed locally or remotely are reflected in the editor's user interface using the callback mechanism provided by the adaptation component. For example, figure 5 shows a synchronous session with two users. The updates produced by each user to the selected text version are presented using different colors. In the structure and versions windows, elements that have been modified during the current session are presented in red.

For converting *synchronous* operations into *asynchronous* 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[6]. An insert/remove operation can be *converted to a large* update version

[6] 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.

operation, with the new value of the version equals to the result of applying the given operation to the current state of the version.

For supporting synchronous edition it would also be possible to implement the following alternative approach: the multi-synchronous editor maintains the document's coobject synchronously synchronized as explained before using only *large* operations and it relies on an external editor for synchronous edition of the same text element. An update version operation is submitted when the participants decide to finish the edition of a text element. In this case, the small operations, typical of a synchronous setting, are executed outside of the coobject's control. Thus, the coobject does not need to convert synchronous operations into asynchronous operations — this conversion is executed implicitly by the editor.

5.2 Multi-synchronous conferencing tool

In this section we describe a *conferencing* tool that integrates synchronous and asynchronous interactions using the approach described in section 2.1. 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 of discussion is stored in a single sub-object. In each shared space, there is an additional sub-object that is used to index all threads of discussion.

Two operations are defined: create a new thread of discussion with an initial message and post a message to a thread of discussion (as a reply to a previous message). 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[7].

Our tool also allows users to maintain several replicas of a shared space synchronously synchronized. This is achieved using the synchronous adaptation component, as in our previous example. 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.

At the data management level, the thread sub-object defines an additional operation to add a message to a previous message. When the user decides to

[7] This property is usually considered sufficient in this type of application. For guaranteeing the same order in all replicas, a component that implements an optimistic total order could be used instead of the causal order component).

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 incorporating the new messages). In this case, the other rules needed for the log compression algorithm 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.

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 [27], Bayou [13], BSCW [5], Prospero [10], Sync [30], Groove [17]). 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). Moreover, it is the base for supportting the integration of synchronous sessions. Second, most of those systems (excluding BSCW) concentrate their attention on the reconciliation problem and do not address awareness support. Our system allows to integrate a solution for handling awareness information. From these systems, at least three can provide some integration between synchronous and asynchronous interactions.

In Prospero [10], it is possible to use the concept of streams (that log the sequence of operations executed) to implement synchronous and asynchronous applications (depending on how often streams are synchronized). This mechanism can be used to implement the integration of synchronous and asynchronous sessions when the same operations can be used in both styles of cooperation. However, the authors do not address the problem of applications that need to use different operations 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" [13] by connecting to the same server. However, without implementing a notification mechanism that allows applications to easily update their interface, it is difficult to support synchronous interactions efficiently. Moreover, relying on a single replica for synchronous sessions may lead to unacceptable latency.

In Groove [17], while some applications can only be used in asynchronous mode (e.g. Notepad), others can be used in synchronous and asynchronous (off-

line) modes (e.g. Sketchpad). In Sketchpad, the same reconciliation strategy seems to be used (execute all updates by some coherent order leading to a solution similar to the *last-writer wins*). However, as discussed in section 2, 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 [42, 22].

Other groupware systems have presented solutions to integrate synchronous and asynchronous interactions. In [15] the authors define the notion of a room, where users can store objects persistently. Applications also run inside the room. A user may connect to the central server that stores the room to observe and modify the room state (using the applications that run inside the room). Users can work in a synchronous mode if they are inside the room at the same time. Otherwise, they work asynchronously. In [19] the authors present a hypertext authoring system that allows users to work synchronously and asynchronously. A tightly coupled synchronous session, with shared views, should be established to allow more than one user to modify the same node or link simultaneously (a locking mechanism prevents any other concurrent modification of those elements). In [36], 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 while using the system). Furthermore, either they do not support divergent streams of activity to occur (besides very short-time divergence during synchronous sessions) or they solve the problem through a single solution (versioning). Our solution is more general, allowing to use the appropriate reconciliation solution for each setting.

In [40], the authors propose a general notification system that can be used to support synchronous and asynchronous interactions by using different strategies to propagate updates. They also present a specific notification component for group editors that implements an operational transformation algorithm (in both settings) that solves some of the problems for using this approach in asynchronous settings. However, as discussed in section 2, when used in asynchronous settings, this technique 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 [29], the authors present a brief overview of SAMS, an environment that supports synchronous, asynchronous and multi-synchronous interactions using an operational transformation algorithm extended with a constraint-based mechanism to guarantee semantic consistency [3]. Although addressing the problem of semantic consistency is important for integrating synchronous and asynchronous interaction, the solution does not seem to allow the use of different strategies to merge updates executed in different settings (as it is important for integrating synchronous and asynchronous interactions in some applications).

7 Final remarks

In this paper, we have analyzed the requirements for supporting the integration of synchronous and asynchronous interactions in different types of applications. Based on this analysis, 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 support asynchronous groupware relying on optimistic server replication and client caching.

To integrate synchronous sessions in the overall asynchronous activity we address three main problems (identified as important in the analysis of section 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 executed operations to all replicas.

Second, our system addresses the problem of using different reconciliation strategies in different settings. To this end, the programmer may use the DOORS object framework to specify that a different reconciliation component should be used in each setting.

Finally, it addresses the problem of using operations with a different granularity 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.

More information about the DOORS system is available from [blanked out]. DOORS code is available on request.

References

1. JGroups. `http://www.jgroups.org` .
2. G. O. Abdessamad Imine, Pascal Molli and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work (ECSCW'03)*, Sept. 2003.

3. H. S.-M. ans Pascal Molli and G. Oster. Semantic consistency for collaborative systems. In *Proceedings of the Fifth International Workshop on Collaborative Editing*, 2003.
4. D. Beard, M. Palaniappan, A. Humm, D. Banks, A. Nair, and Y.-P. 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.
5. 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.
6. J. 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.
7. P. Cederqvist, R. Pesch, et al. Version management with CVS, date unknown. `http://www.cvshome.org/docs/manual` .
8. A. Demers, D. Greene, C. Hauser, W. Irish, and J. 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.
9. P. 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, Sept. 1995.
10. P. Dourish. Using metalevel techniques in a flexible toolkit for cscw applications. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(2):109–155, 1998.
11. P. Dourish and V. 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.
12. S. Duarte, J. L. Martins, H. J. Domingos, and N. Preguiça. Deeds - a distributed and extensible event dissemination service. In *Proceedings of the* 4^{th} *European Research Seminar on Advances in Distributed Systems (ERSADS)*, May 2001.
13. W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. 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.
14. 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.
15. S. Greenberg and M. Roseman. Using a room metaphor to ease transitions in groupware. Technical Report 98/611/02, Department of Computer Science, University of Calgary, Alberta, Canada, Jan. 1998.
16. I. Greif and S. Sarin. Data sharing in group work. *ACM Transactions on Information Systems (TOIS)*, 5(2):187–211, 1987.
17. Groove. Groove workspace v. 2.5. `http://www.groove.net` .
18. C. Gutwin and S. 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.
19. J. M. Haake and B. 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.
20. L. K. 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.

21. A. Karsenty and M. 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.
22. R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):375–409, 1990.
23. A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. 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.
24. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.
25. M. J. Knister and A. 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.
26. M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work*, 3(3-4):359–378, 1995.
27. Lotus. IBM Lotus Notes. `http://www.lotus.com/notes` .
28. Microsoft. Microsoft outlook. `http://www.microsoft.com/outlook` .
29. P. Molli, H. Skaf-Molli, G. Oster, and S. Jourdain. SAMS: Synchronous, Asynchronous, Multi-Synchronous Environments. In *Proceedings of the 2002 ACM Conference on Computer supported cooperative work in design*, 2002.
30. J. P. Munson and P. Dewan. Sync: A java framework for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, June 1997.
31. 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.
32. F. Pacull, A. Sandoz, and A. 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.
33. N. Preguiça, J. L. Martins, H. Domingos, and S. 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.
34. N. Preguiça, J. L. Martins, H. Domingos, and S. Duarte. Supporting groupware in mobile environments. Technical Report TR-04-2002 DI-FCT-UNL, Dep. Informática, FCT, Universidade Nova de Lisboa, 2002.
35. D. Qian and M. D. Gross. Collaborative design with NetDraw. In *Proceedings of Computer Aided Architectural Design (CAAD) Futures '99*, 1999.
36. C. Qu and W. 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.
37. M. Roseman and S. Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(1):66–106, 1996.
38. D. W. Saul Greenberg, Mark Roseman and R. 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.
39. C. Schuckmann, L. Kirchner, J. Schümmer, and J. 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.
40. H. Shen and C. Sun. Flexible notification for collaborative systems. In *Proceedings of the 2002 ACM Conference on Computer supported cooperative work*, pages 77–86. ACM Press, 2002.

41. H. S. Shim, R. W. Hall, A. Prakash, and F. 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, Sept. 1997.
42. C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(1):1–41, 2002.
43. C. Sun and C. 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.
44. 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 Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.
45. W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.
46. N. Vidot, M. Cart, J. Ferrié, and M. 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.
47. Y. Yang, C. Sun, Y. Zhang, and X. Jia. Real-time cooperative editing on the internet. *IEEE Internet Computing*, 4(3):18–25, 2000.