Data Management Support for Asynchronous Groupware

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

Abstract
In asynchronous collaborative applications, users usually collaborate accessing and modifying shared information independently. We have designed and implemented a replicated object store to support such applications in distributed environments that include mobile computers. Unlike most data management systems, awareness support is integrated in the system. To improve the chance for new contributions, the system provides high data availability. The development of applications is supported by an object framework that decomposes objects in several components, each one managing a different aspect of object "execution". New data types may be created relying on pre-defined components to handle concurrent updates, awareness information, etc.

Keywords
Asynchronous groupware, mobile computing, awareness, object framework, development support.

INTRODUCTION

The ubiquity of the Internet has opened opportunities for collaboration among people on different geographical locations. Several general-purpose services, such as e-mail and news, have been used to support basic interaction and collaboration. However, enhanced support for groups of users collaboratively seeking common goals requires specialized applications such as multi-user editing tools, cooperative schedulers and calendars, workflow-based applications, conferencing systems and others [8]. Most of these applications rely heavily on a data storage sub-system to enable information sharing, distribution and composition. Some support systems have been implemented, either, for general use (e.g. Lotus Notes [16]), for specific domains (e.g. Vortex [11] for workflow) or for specific applications (e.g. Iris [15] for document editors).

In asynchronous groupware, users usually collaborate accessing and modifying shared information without immediate knowledge about the actions of other users (either because users work at different times or simply because they do not have access to each other's actions). To improve the chance for asynchronous collaboration, users should be allowed to perform their contributions independently without restrictions (besides coordination and access control restrictions). To accomplish this requirement, a replicated data management system with a "read any/write any" model of data access is often used. The increasing popularity of mobile and disconnected computing, with its inherent characteristics [21], seems to further strength the above approach - mobile users expect to be able to access and modify shared information even while disconnected. Using this model, users may execute concurrent updates. To synchronize these concurrent streams of activity [5] adequately, it is often necessary to rely on application-specific semantic information [12,14,7].

Awareness information is often essential to the success of collaborative activities [4]. In asynchronous groupware, although users have no immediate knowledge of each other's actions, overall information about the evolution of the collaborative activity (e.g. evolution of the shared data, users' actions,…) may improve each user's contributions. Different collaborative activities, applications and users will demand different forms of awareness information (e.g. a scheduler application may actively notify users of new appointments while a collaborative writing system may simply maintain a log of modifications).

In this paper we present the data management approach of the DAgora project [1] to support asynchronous groupware. It has two main components: (1) a replicated object store that integrates awareness mechanisms; and (2) a framework that eases the creation of new data types that are specially tailored to be used in collaborative applications.

The DAgora distributed object store (named DOORS) is based on a group of servers that replicate sets of related objects with a "read any/write any" model of data access. It also includes a client caching mechanism that allows clients to cache frequently used objects. The combination of these mechanisms provides high data availability even in the presence of voluntary disconnection and network and/or server failures. To maximize the semantic information available to synchronize divergent streams of activity, the system propagates operations instead of data values.

DOORS objects are structured according to an object framework that decomposes object “operation” in different

aspects: concurrency control, awareness-support, etc. This framework eases the development of collaborative applications because it allows programmers to define new shared data types using, in each data type, the adequate pre-defined components for each aspect of data sharing.

The remainder of this paper is organized as follows: section 2 discusses requirements and design choices; section 3 outlines the DOORS architecture; section 4 presents the object framework; section 5 describes our experience using the DOORS system and section 6 discusses related work; section 7 concludes the paper with some final remarks.

REQUIREMENTS AND DESIGN CHOICES

In this section we present the requirements and design choices that lead to the DAgora data management approach to support asynchronous collaborative activities. To illustrate some of the requirements of asynchronous groupware we will use examples from three well understood applications: a conferencing system, a group scheduler and a multi-user editing tool.

In an asynchronous conferencing system, users collaborate through the exchange of messages posted in a shared space (newsgroups are a simple example). These messages create threads of discussion related with different subjects. Users should be allowed to independently post new messages, which should be displayed in a consistent way (at least, message dependencies should be taken into account - a reply should not be displayed before the original message).

In a group scheduler, multiple users should be allowed to request new appointments independently. These new appointments should be considered tentative [7] until being committed by some form of automatic global agreement. If requested, users should be notified when their requests are committed or aborted.

A multi-user editing tool allows a group of users to collaboratively edit some structured document (for example, this paper). Different users should be allowed to independently modify the document (e.g. two users can modify different sections). Concurrent modifications should be merged taking into consideration all changes (e.g. the final document must include the new versions of the two sections). Syntactic consistency[1] [5] should be preserved even when handling *semantic* conflicts (e.g. if two users modify the same section, two versions of that section should be created and maintained).

From the previous brief descriptions we note that users collaborate through the access and modification of shared information. Therefore, data management plays an important role in the support of groupware. In this paper we restrict our discussion to the data-management support. However, in the DAgora project [1] we are also addressing other problems, such as the coordination of collaborative activities [3], the support of synchronous applications [23] and the dissemination of awareness information relative to the collaborative activity [6].

High data availability

One important requirement to enable collaboration is to allow users to access shared information. In distributed settings, it is impossible to guarantee the permanent reachability of a single storage site (due to network and machine failures). Therefore, systems that intend to provide high data availability usually rely on data replication - data can be accessed if some replica is available. In DOORS, the information associated with a collaborative activity is grouped in a volume [14] that is replicated by a group of servers.

Mobile users are frequently disconnected from the network, either due to economical factors, energy saving or unavailable connectivity. Disconnected users have to rely on local data replicas to access the shared information. However, as it is often impossible and/or undesirable to allow all disconnected computers to manage a full unit of replication, clients usually rely on a caching mechanism. In DOORS, clients cache a set of key objects to allow disconnected users to continue their work.

From the previous examples we note that users contribute to a collaborative activity through the modification of the shared information. Therefore, to promote collaboration, users should be allowed to modify data without restrictions. In a distributed setting that includes mobile/disconnected computers, pessimistic concurrency control mechanisms based on locks or tokens lead to low write availability (for example, to allow a single disconnected client to modify data it may be necessary to prevent all other clients from updating it). In DOORS, we have adopted an optimistic replication strategy with a "read any/write any" model of data access - all clients are allowed to modify data independently. We have also adopted an epidemic scheme of update propagation [2], where every server eventually receives all updates from every other, either directly or indirectly. This scheme requires only occasional pair-wise communication between computers, thus imposing minimum connectivity requirements among computers. The DOORS architecture is detailed in the next section.

Multiple concurrency-control strategies

In the optimistic replication scheme adopted in DOORS, as in other systems previously presented in literature [7,16,14,15], users may independently perform their updates. This situation leads to the need to handle concurrently performed updates. Moreover, due to the lazy replication strategy, updates are not propagated to all servers at the same time - different servers may have received different subsets of the performed updates. Although similar approaches have been identified as necessary to support large-scale distributed environments

[1] A system is syntactically consistent if the underlying data store is structurally sound, allowing the activity to proceed.

that include mobile computers [7,9,14], they pose an important data management problem - how to handle the concurrently performed updates?

Consider the following examples from the previously presented applications. In a conferencing system messages should be displayed taking into consideration their dependencies. However, there is no need to display all messages in the same order in all replicas - a causal order is sufficient. On the other hand, in a scheduler application all appointments must be committed in the same order in all replicas - all updates should be applied using a total order. Uncommitted updates may be presented as tentative.

Many algorithms have been proposed to handle concurrent updates (based on undo-redo techniques [13], operation transformations [24], exploitation of data types semantic properties [12],…). However, it seems that no single method is adequate to all situations. Instead, different groups of applications will use different mechanisms. Nevertheless, the use of semantic information has been identified [7,14,18] as a key element to merge the concurrent streams of activity.

In DOORS we allow each application to define its own concurrency control method (see "application development support" for reuse support). To maximize the flexibility in the handling of concurrent updates, we have based our system in the propagation of updates as operations. Consequently, the concurrency control mechanism may use not only the semantic information associated with the data type but also the semantic information associated with each performed operation.

Integrated awareness support

Awareness has been identified as important in the development of collaborative activities because individual contributions may be improved by the understanding of the activities of the whole group [4,19]. To this end, it is important to have information about the evolution of shared data, users' actions and motivations,… In asynchronous collaborative activities, awareness information plays a central role in collaboration allowing each user to take notice of new contributions from other users.

Consider the previously presented multi-user editing tool: it is important that each user takes notice of updates performed to the shared document. To this end, information about updates should be automatically collected and maintained with the document. This information may be displayed to users using different user interface metaphors (log of changes, multiple colors in the document,…). This form of awareness is usually called shared feedback [4]. Consider now the group scheduler application: it seems interesting to allow the affected persons to be notified of the commitment or abortion of any requested appointment (a log with the results of requests may also be maintained). In our architecture, updates are processed asynchronously by the data management system and users will usually not be connected to the system when their updates are committed[2]. Consequently, we believe that the support for handling awareness information relative to the evolution of the shared data should be tightly integrated with the data management system. This integration allows each processed update to produce the adequate awareness information (and eventual notifications to be propagated immediately).

In DOORS, we have made this integration. Each processed update may produce a piece of awareness information. Each data type may define the way this information is handled, thus supporting different awareness models (e.g. awareness data may be stored with the data object or/and may be immediately propagated to users). We will detail the awareness support in the next sections.

Application development support - reusing pre-defined base solutions

In the previous subsections we have presented some of the data management requirements that lead to the DAgora approach to support asynchronous collaborative applications. It is important to notice that the fulfillment of those requirements demands type-specific solutions. The DOORS design choices attempt to offer the maximum flexibility in the implementation of different solutions - different groups of applications require different solutions. However, to fulfill the objective of providing support for the development of new applications, this flexibility is not sufficient – application programmers should be assisted in the implementation of their specific data types. To this end, we have created a data management object framework that decomposes each object in several components, each one responsible for a different aspect of the object "operation".

Moreover, we have implemented a set of pre-defined components that execute different policies, notably related with concurrency control and awareness support. Using the DOORS open object framework, application programmers may create new data types composing these pre-defined components with regular object classes. If necessary, programmers may create new components or extend any pre-defined one. The DOORS object framework does not restrict reuse to concurrency control and awareness support. For example, we have defined a base component that acts as a surrogate of a relational database system. This component allows programmers to store their data using a relational model while relying on DOORS facilities. We will detail the object framework and some of the implemented base components in the section "object framework".

As it has already been mentioned, the DOORS replication mechanism is based on the propagation of the operations performed by users. To this end, the invocation of

[2] Note that this operational pattern (updates being processed when users are no longer connected to the system) is common in many data management systems that support disconnected users [7,12,14].

operations should be logged in clients. In DOORS, we use a preprocessor that transforms the object classes implemented by the application programmer so that the invocation of operations could be logged transparently when user applications call object methods.

ARCHITECTURE

DOORS is a distributed object store based on a "extended client/replicated server" architecture. It manages objects structured according to the DOORS object framework (named coobjects - from collaborative objects). These coobjects may represent rather complex data objects, such as documents or calendars, and be implemented as arbitrary compositions of common objects. Sets of related coobjects are grouped in volumes that represent collaborative workspaces and store the data associated with a given workgroup and/or collaborative project.

The DOORS architecture is composed by servers and clients, as depicted in figure 1 - any machine may act as both a client and a server. Servers replicate volumes of coobjects to mask network and/or server failures. Clients cache key coobjects to allow users to continue their work, even while disconnected. Applications that use DOORS to store their data run on client machines and modify coobjects through the invocation of coobjects' methods - users collaborate through the modification of shared coobjects.

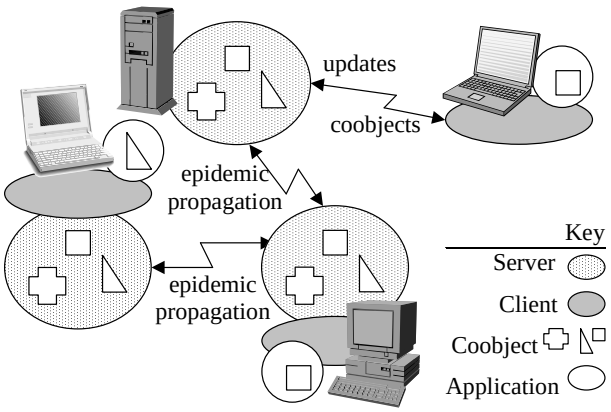

Figure 1 – DOORS architecture composed by four computers with different configurations. Coobjects are replicated by servers, cached by clients and manipulated by users' applications.

Applications usually use a "get/modify locally/put changes" model of data access. When an application requests a coobject, if it is not available in the local cache, it is fetched from any server (if connectivity is available). A private copy of the coobject is created in the client component and it is handed over to the application. Applications use coobjects as common objects, i.e., applications invoke coobjects' methods to query and modify their state. Updates performed by applications are logged internally by coobjects without any intervention of the DOORS system. These updates are recorded as sequences of method invocations (properties of operations are used to compress these sequences - for example, only the last operation that sets new text to a section is recorded). Finally, if the user chooses to save her changes, the logged sequence of updates is transferred to the client, where it is persistently stored until it is propagated to a server. Alternatively, DOORS supports a remote invocation mechanism that immediately processes on a server any method invocation performed on a coobject (if connectivity is available). This mechanism is mainly used to support access to large coobjects that can not be instantiated in clients, in particular coobjects that act as surrogates of RDBMS (see next section).

Upon arrival of updates from a client machine, the server hands them over to the coobject local replica. It is up to the coobject replica to store and process these updates - updates are propagated from clients to servers and among servers as sequences of method invocations. Servers propagate new updates among them in an epidemic way [2] - pairs of servers establish occasional communications to synchronize received updates. Therefore, all servers will eventually receive all updates, either directly or indirectly. DOORS allows pair-wise communications to be established over multiple transports, including asynchronous ones such as e-mail. This property, combined with the epidemic strategy of update propagation, imposes minimum connectivity requirements among servers. As a consequence of the lazy strategy of update propagation, multiple replicas may differ at a given moment, but they will eventually converge (at least all replicas will reflect the same set of updates). These temporary inconsistencies may be reduced by increasing the frequency of epidemic propagation sessions.

The group of servers that replicate each volume may vary as a result of users (system administrators) explicit orders. The protocols implemented for membership management are light-weight and impose only pair-wise communications. To support push-based awareness models, each server contains an "awareness service". This "awareness service" is responsible to propagate messages through the defined mechanisms - e-mail, SMS/pager gateways or other (to cope with temporary impossibilities of propagation, several retries may be necessary). To support coobjects that act as database surrogates, each server may have an associated database system to maintain local replicas.

OBJECT FRAMEWORK

In the previous section we have outlined the DOORS system. As it has been described, the responsibilities of the system core are almost restricted to provide high data availability: it should maintain updated copies of coobjects in clients and propagate updates between clients and servers and among servers. The rationale behind this design is to allow flexible support of collaborative activities: "operational" aspects of data management, such as concurrency control and awareness support, are controlled in a type-specific way and are defined in the implementation of the shared data types (coobjects).

Therefore, the system core is limited to minimal services that represent the common aspects of data management.

On the other hand, a heavy burden is imposed on the implementation of coobjects, which must handle several aspects that are usually managed by the system. To alleviate programmers from much of this burden and to allow the reuse of "good" solutions in multiple data types, we have defined an object framework that decomposes coobjects in several components that handle different operational aspects. In this section we will present the object framework and some of the reusable pre-defined base components that have been implemented. In the applications presented in the next section we will exemplify the use of this object framework.

The DOORS object framework structures each coobject in six components, each one with a well defined function and interface (see figure 2). We will now briefly present these components and outline (with some simplifications) how they work together in the clients and in the servers. The *capsule* aggregates the other components of the coobject and serves as an interface to the DOORS system. The *"attributes"* component stores general-purpose information about the coobject. The *log* stores the updates performed by users. The *concurrency control* component is responsible to execute the logged updates. The *data* component defines the conventional data type represented by the coobject, with its internal state and operations. The *awareness* component handles the awareness information.

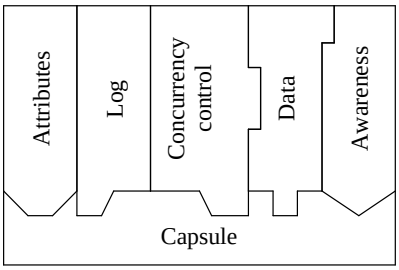

Figure 2 – DOORS open object framework.

In clients, applications manipulate coobjects using the methods defined in the *data* component. Updates performed are transparently logged (in the *log*) and executed. When the user decides to save her changes, the system extracts the sequences of updates performed from the *log*, and later propagates these updates to a server.

In servers, the updates received from clients and from other servers are stored in the *log* – during epidemic sessions these stored updates are propagated among replicas. The *concurrency control* component is responsible for executing logged updates according to the defined strategy. When an update is executed some awareness information may be produced: this information is handled by the *awareness* component. Next, we will detail the components of the DOORS open object framework and present some pre-defined base solutions.

Capsule component

The *capsule* component defines the composition of the coobject and aggregates its components. It implements the interface used by the system core to interact with coobjects and coordinates coobjects' operation. Commonly, coobjects are composed by one instance of each component and work as it has been described. However, we have defined a *two-version capsule* that allows programmers to implement coobjects that maintain two data versions relying on common object classes. Usually one version stores the committed state (result of the execution of all stable updates) and the other stores a tentative state (reflecting all known updates). In the next section, we present the example of a scheduler application that uses this capsule to allow users to observe both the tentative and committed appointments – this behavior has been identified as interesting for applications that use replicated data in large-scale settings that include mobile computers [7].

The *two-version capsule* contains two *data* components, two *awareness* components and two *concurrency control* components. Each data component stores a different version of the data. These versions are easily maintained executing the stored updates resorting on different *concurrency control* policies - the committed (tentative) version is obtained using a pessimistic (optimistic) strategy. The two *awareness* components allow programmers to handle stable and unstable awareness information in different ways.

"Attributes" component

The *"attributes"* component is used to store general-purpose information relative to the coobject, such as creation and modification dates, owner, etc. It also stores meta-information about the coobject state and the replication process, such as summaries (timevectors) of received, executed, discarded and delivered updates [20]. Two base implementations are available: a simple and an extended one. The extended implementation maintains additional information about a primary replica and defines methods to modify it. This information can be used by other components in their operation - e.g. concurrency control mechanisms based on a sequencer use this information (see "*concurrency control* component" for details). These base classes may be extended to define type-specific attributes.

Log component

The *log* component stores the updates performed by users. In clients, these updates are temporarily logged until they are requested by the system (when the user saves his changes) to be propagated to a server. In servers, updates are permanently stored (until they are discarded) and they are propagated during epidemic synchronization sessions. The *log* adds, to each sequence of updates, the necessary information to order them and to trace their dependencies, i.e., to identify the multiple streams of activity.

Two base implementations are available: a simple one and an extended one. The extended one should be used with *concurrency control* components based on a sequencer (see

details in the next subsection). The sequence of updates produced in clients is automatically compressed if operations' properties are available - e.g. if multiple consecutive updates (performed in the same session by the same user) set a new value to the same section of a document, only the last update needs to be recorded.

Concurrency control component

The *concurrency control* component is responsible for executing the updates stored in the log. In clients, intercepted updates are usually immediately executed to allow users to observe the expected results from their actions. In servers, multiple strategies may be used to synchronize the concurrent streams of activity depending on the semantics of the shared data-type. Two inter-related problems must be taken into consideration: how to guarantee that all data replicas evolve as expected and how to guarantee that users' intentions are respected.

The first problem can be handled constraining the execution order of updates. For example, executing all (deterministic) updates by the same order in all replicas leads all replicas to the same state. However, as different replicas may have received different subsets of updates, it is usually necessary to postpone the execution of some updates to guarantee that property. In some applications it is not necessary to achieve exactly the same state or due to semantic properties it is possible to rely on weaker orderings to achieve the same state. We have implemented several components with different policies (in the next section we present examples of their usage).

The *no order* component executes updates as soon as they are received for the first time - therefore, different replicas may apply the same updates in a different order. The *causal order* component executes updates in a causal order (when a user performs a new update, he observes a coobject state that reflects a given set of previous updates - causal order guarantees that the new update is executed in all replicas after that set of updates). Due to the DOORS epidemic propagation model, this constraint does not usually impose any delay on the execution of updates - therefore, replicas usually reflect all known updates.

We have also implemented several components that execute updates in the same order in all replicas using different techniques - all components rely uniquely in information propagated during normal epidemic sessions. The *stability-based total order* component implements a fully distributed algorithm to establish the total order (based on the ReplicatedStateMachine algorithm, see [17]). As this algorithm relies on information gathered in all replicas, a permanently disconnected computer may prevent new updates from being executed. The *sequencer-based total order* component uses a replica designated as sequencer to order updates. These updates are executed in all replicas in the order defined by the sequencer. The identity of the sequencer may be modified using the methods defined in the *"attributes"* component. This component allows new updates to be (ordered and) executed if they can be propagated to the sequencer replica (even in the presence of multiple permanently disconnected computers).

Both total order components postpone the execution of updates until the order of updates can be positively established. However, in some situations it is preferable to use an optimistic approach - updates are immediately applied assuming that no previous updates are unknown. This allows users to observe the expected results from all known updates. Subsequently, if some previous update is received the updates executed in the wrong order are undone and later redone in their correct position. We have implemented versions of the previous total order components that use this undo-redo optimistic strategy [13].

The second problem is to guarantee that users' intentions are respected when concurrent streams of activity are synchronized. To tackle this problem, three main approaches have been proposed in literature. First, the use of transactions - an update is committed if data values are equal to those observed by the user, otherwise it is aborted. This model is too restrictive because the strict identity of values is not necessary in many situations - some transactions could be executed if data honors a weaker condition. Moreover, the abortion of users' contributions is not usually acceptable in asynchronous groupware where contributions may represent large amounts of work (in synchronous groupware several systems have used this approach successfully - e.g. [22]). Second, updates are transformed taking into consideration the updates executed after the state observed by the user [24]. Third, semantic information is used in synchronization [7,12]. Although we have extended the *optimistic stability-based total order* component to execute update transformations (that must be specified by programmers), we expect that most applications will rely on semantic information to guarantee the fulfillment of users' intentions. As updates are applied in servers executing coobjects' methods, it is possible to express the expected semantics in the code of operations - pre and post-conditions can be checked and alternative actions may be executed depending on the state of the coobject (see the next section for examples).

Awareness component

The *awareness* component is used to handle the awareness information generated in the execution of updates. Two main implementations are available. The *notification-based* component propagates information to users using their preferred transport - e-mail messages, SMS/pager messages, etc. The *shared-feedback* component stores the awareness information, so that it can be presented to users in applications - e.g. a multi-user editor may present the log of updates. We have also implemented some wrappers that guarantee the expected semantics despite the multiple concurrency control policies (e.g. awareness information may not be propagated as the result of unstable updates in optimistic concurrency control strategies, although updates

are executed in all servers only one message is propagated,…).

In the future, we expect to provide an *enhanced notification-based* component that propagates information to users through our new event-dissemination architecture (see [6] for a preliminary design). This architecture will allow users to specify the way awareness information is propagated to themselves – for example, some user may request to be immediately notified using SMS/pager messages for important information, daily digests for information about activities that he is not directly involved on, and e-mail for other messages.

Data component

The *data* component implements the data type being created, with its associated state and operations. It is implemented as a common class that is preprocessed, so that updates performed by users can be transparently logged. We have implemented some base components that can be extended to easily create data types with specific approaches. The *structured multi-version* component defines a hierarchy of sub-objects that may have multiple versions. Two types of operations are available: operations that modify the sub-objects and operations that manage the hierarchic organization. Concurrent modifications of the same sub-object are automatically detected and solved creating multiple versions - the information added to the operations by the *log* component is used to detect concurrent modifications without the need for any special *concurrency control* component (as in [14,16]). Programmers may define automatic merge procedures or let this work to users. Concurrent modifications of the hierarchic organization are automatically merged in a coherent way. This component is suitable for situations where it is impossible to automatically solve conflicts but syntactic consistency must be maintained to enable users to continue their work (e.g. in a structured document two versions should be created when the same section is concurrently modified by two users). We have used this component to implement several structured documents (as it is detailed in the next section).

The *database surrogate* component implements a surrogate of a database. It provides basic methods to query and update the database using a Java JDBC-like interface (i.e. queries and updates may be performed using SQL). To allow different servers to hold local data replicas using different database systems, only SQL standard statements must be used. This component may be extended to create new data types that use the relational data model - these new data types should define high-level operations that use those basic methods. Coobjects that use such data-types are structured as all other coobjects and rely on the same DOORS mechanisms (e.g. updates are propagated and executed in all replicas using the common mechanism). Similarly, we have also defined a *file-system surrogate* that allows programmers to store their data in files.

EXPERIENCE

In the previous sections we have described the DOORS system. In this section we will present some applications that illustrate the DOORS support for asynchronous groupware. In particular, we will focus on the use of the DOORS object framework to ease the development of new applications. The applications presented in this section have used a DOORS prototype implemented in Java 1.2. The DOORS pre-processor has been implemented using the JavaCC parser generator.

Scheduler

The scheduler application enables users to manage a shared calendar. This calendar may be used to schedule the reservation of a shared resource (e.g. meeting room) or as a personal date book managed by more than one person (e.g. the owner and his secretary). In this application multiple users may independently request reservations, thus imposing a high data availability requirement. As only a single reservation may be granted for the same period of time, a global agreement mechanism must be used to commit requests and decide possible conflicts. However, users must be able to observe not only committed requests but also those that have not been committed yet. As these tentative updates represent the expected state of the shared calendar, users should also avoid conflicts with those updates. To reduce the likelihood of having rejected requests, users may provide alternative periods of time for their requests. Additionally, users may want to be notified when the results of their requests are determined.

To develop this application using DOORS it was necessary to implement the shared calendar as a new coobject (and then to implement the application that manipulates this coobject). First, we have implemented the coobject's data component - we have used a common calendar class, as it would be implemented for a local application. In figure 3 we present the method that processes new requests. As it can be seen, this method sequentially checks the possibility to schedule the appointment in alternative periods. The result of the operation is reported using the method *newAwarenessMsg* (and it will be processed by the *awareness* component).

Next, this data component has been composed with some pre-defined components to create the shared calendar coobject. In this application we wanted to be able to present the committed and tentative updates using different colors. In DOORS, we can maintain two data versions using normal data classes relying on the *two-version capsule*. The committed state is maintained executing the stored updates by a pessimistic total order (e.g. the *sequencer-base total order*) - therefore all replicas will execute all updates in the same order, thus deciding possible conflicts in concurrently performed requests. The tentative state is maintained executing all other updates by any order to a copy of the committed state (or executing updates by the correspondent optimistic total order). To provide notifications of the final

result of requests, we have associated the *notification-based* awareness component to the committed data (and a wrapper so that a single notification is propagated for each request). With the tentative data version we have associated a null awareness component, so that no notification is propagated as the result of tentative updates. Finally, we have also used a *log* and a *"attributes"* components (the extended versions must be used with the *sequencer-based total order* concurrency control).

```
public update void processRequest(RequestInfo request){
    for( int i = 0; i < request.period.length; i++) {
        if( availablePeriod( request.period[i]) {
            insertRequest( request, request.period[i]);
            newAwarenessMsg( request.user,
                "Reservation confirmed at " +
                request.period[i] + "\nDetails:\n" +
                request.detailedInfo() );
            return;
        }
    }
    newAwarenessMsg( request.user,
                "Impossible request.\nDetails:\n" +
                request.detailedInfo() );
}
```

Figure 3 – Method *processRequest* for scheduler coobject. The method *newAwarenessMsg* is defined in the base *data* component class and it is a simple redirection to the awareness component. The "update" keyword is used by the DOORS preprocessor.

As outlined, the shared calendar coobject was implemented using a simple calendar class and relying on several pre-defined components to manage the complexity associated with data sharing among multiple users. High data availability is provided by the core of the DOORS system.

Multi-user document editor

The editor application allows users to produce structured documents collaboratively. A document is a hierarchical composition of two types of elements: containers and leaves. Containers are sequences of other containers and/or leaves. Leaves represent atomic units of data that may have multiple versions and that may be of different types. In figure 4, we present an example of a LaTeX document. Users are allowed to change the same document independently and the system must manage these changes. If two users modify the same atomic element, two versions of that element should be created, thus maintaining syntactic consistency - the system can not decide which version is the best one and no work should be discarded by the system. Users should merge both versions later. Concurrent modifications of the document structure should be merged applying both modifications in the same way in all replicas - for example, if two users are adding a new section they are probably adding different sections. Additionally, the application should provide awareness information to users presenting the modifications performed by other users.

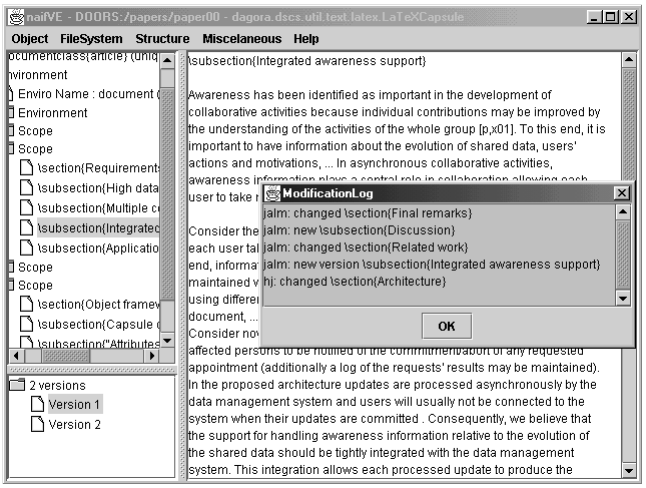

Figure 4 – Multi-user editor with a LaTeX document.

To create a shared document coobject in DOORS, we have used the *structured multi-version* data component to automatically manage the document structure and multi-versioning of atomic document elements. This component had to be extended to define the allowed configuration of elements and the type of atomic elements. To guarantee that all modifications in the structure of a shared document are executed in the same way in all replicas we have used an *optimistic total order* concurrency control component. This component guarantees, not only, that the shared document replicas will converge, but also that users can immediately observe all contributions performed by users.

We have used the *shared-feedback* awareness component to store awareness information about all updates performed. This information is used by the editor application to provide shared-feedback awareness information to users. The shared document coobject has also used a simple *capsule* component, a *log* component and a *"attributes"* component. As outlined, we have used and extended DOORS pre-defined components to create multiple shared document coobjects - data-management problems related with data sharing are managed by the pre-defined components.

Musical shared database

The musical shared database is an application that manages information about music. Multiple users are allowed to introduce information about new albums, their songs, authors, producers, on-line pointers, etc. Associated with each album there is a discussion board where users may produce their comments on the album and reply to previous comments. Users may also classify albums regarding several characteristics. Additionally, users may request to be notified when albums with specific characteristics are introduced. As usual in database applications, a set of possible queries is also provided to users.

This application has been developed using DOORS to manage data distribution - multiple database replicas may be distributed in different computers and may be accessed independently (e.g. different intranets may have different

database replicas that are synchronized during e-mail-based epidemic sessions, a portable computer may hold a replica, etc). The data component has been implemented extending the *database surrogate* - application-level operations (such as *insert a new album* and *insert a comment*) have been defined using basic SQL statements. The *causal order* concurrency control component has been used to guarantee that the dependencies among operations are respected (e.g. replies are always posted after the original comments) - this property is sufficient in this case. This coobject relies on the *notification-based* awareness component to implement notifications to users. As usual, a *log* and a *"attributes"* components have been used.

RELATED WORK

Several systems have been designed to manage data in large-scale distributed environments. Notably, database systems usually rely on transactions to manage concurrency control. However, as we have already discussed, we believe that transactions are too restrictive for asynchronous groupware. Distributed file systems, such as Coda [14], support data sharing among distributed and disconnected users. Coda supports automatic conflict detection and resolution relying on application-defined programs to merge multiple file versions. As these programs have no information about executed operations, their task is difficult and sometimes impossible. This is not a problem in the targeted environment where conflicts are expected to be the exception - however this problem is important for general groupware support.

Lotus Notes [16] is a replicated document database based on epidemic update propagation. Documents have a record-like structure composed by typed fields defined in forms. Notes propagates field values, handling concurrent updates by the creation of multiple field versions that must be manually merged. Although this approach is suitable in some circumstances, automatic merging of concurrent streams of activity is often possible and desirable.

Bayou [7] is a replicated database system based on epidemic operation-based update propagation. Bayou updates (writes) include information to allow generic automatic conflict detection and resolution through dependency checks and merge procedures. Bayou data presents two values: tentative and committed. A primary replica scheme is used to fasten update commitment. This set of data management characteristics is interesting for many collaborative applications, as we have already discussed when we have presented similar DOORS features. DOORS differs from Bayou in three main aspects. First, DOORS includes integrated awareness support about data evolution, which is important for many asynchronous groupware applications. Second, as DOORS allows specific data types definition it does not impose data to fit the relational model (as in Bayou). DOORS also allows the implementation of applications that use the relational data model extending the *database surrogate* data component. However, these applications must define high-level operations in Java using a JDBC-like interface, while Bayou applications may use a higher-level mechanism (based on TCL). Third, the DOORS object framework allows the reuse of different strategies (such as update transformations) - this situation may allow better and simpler solutions for some applications, without imposing additional complexity for programmers.

Several groupware systems are based on a traditional client/server architecture. Sync [18], a framework for mobile collaborative applications, presents a model of concurrent update merging based on the definition of merge matrixes. These matrixes define the operations that must be executed in the server and in clients for each pair of operations. The Prospero toolkit [5] presents a model for data management based on the synchronization of divergent streams of activity - divergence may be constrained using promises and guarantees (an extension of locks). It allows type-specific customization through its open implementation (e.g. synchronization procedures may be redefined). However, the lack of server replication makes these systems less suitable for large-scale settings. Additionally, they do not present integrated awareness support (e.g. no mechanism is defined to notify users of the result of their action - a desirable characteristic in some applications designed for mobile environments).

Shared workspace systems (such as BSCW [10]) allow multiple users to share a common workspace where they can store documents. They usually use simple concurrency control mechanisms based on locking (check-in/check-out) or version management. We believe that it is often possible and desirable to automatically merge concurrent updates. Awareness information is usually pull-based (users must log to the system and poll for new information).

CONCLUDING REMARKS

The DOORS replicated object store provides data management support for asynchronous groupware. Asynchronous collaborative applications, as illustrated in the applications reported in this paper, require high (read and write) data availability to maximize the opportunity for collaboration among users - users perform their contributions modifying shared information. To this end, DOORS combines server replication and client caching to provide high data availability in a distributed environment that includes mobile computers. While DOORS shares goals and approaches with several other systems, it also stresses several different directions that look very promising according to our experience.

First, DOORS is fully built around the notion of operation-based update propagation. This approach maximizes the flexibility in the synchronization of divergent streams of activity relying on both data type and operation semantic information. According to our experience, this flexibility is essential to create specially tailored solutions.

Second, DOORS provides integrated support for handling awareness information that is generated in the execution of operations during the synchronization process.

Third, the DOORS system core is almost restricted to the common functions of data management: propagate updates between clients and servers and among servers. DOORS delegates on coobjects most of the aspects related with the management of data sharing, such as concurrency control, handling of awareness information, etc. Additionally, DOORS defines an object framework that decomposes coobject "operation" in several components that handle those different aspects. This framework eases application development allowing programmers to create new data types relying on several pre-defined solutions to organize data, to manage concurrency control, to handle awareness information, etc. - the applications reported in this paper illustrate its use. Due to this model and to the open implementation of the object framework, DOORS provides the necessary flexibility to manage different collaborative applications.

Some problems also need more attention and further research. First, we intend to enhance our support for small mobile devices introducing support for partial replication and adequate mechanisms to handle variable connectivity. Second, we intend to research more advanced tools to support our component-based object framework. Third, we intend to pursue our research in the event-dissemination architecture and use it to propagate awareness information.

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

Acknowledgments

We would like to thank our students for helpful feedback on the system and for supporting us with the implementation of several applications (in particular, we would like to thank Inês Vicente and Filipe Leitão for the implementation of the musical shared database).

REFERENCES

1. DAgora project homepage - http://dagora.di.fct.unl.pt
2. Demers, A., Green, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D. Epidemic Algorithms for Replicated Database Maintenance. *Operating Systems Review*, 22(1), 1988.
3. Domingos, H., Preguiça, N., Legatheaux Martins, J. Coordination and Awareness Support for Adaptive CSCW Sessions. In *Proceedings of CRIWG'98*, 1998.
4. Dourish, P., Bellori, V. Awareness and Coordination in Shared Workspaces. In *Proceedings of CSCW'92*, 1992.
5. Dourish, P. Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Transactions on Computer-Human Interaction*, June 1998.
6. Duarte, S., Legatheaux Martins, J., Domingos, H., Preguiça, N. DEEDS - An Event Dissemination Service for Mobile and Stationary Systems. In *Actas do 1º Encontro Português de Computação Móvel*, 1999.
7. Edwards, W., Mynatt, E., Petersen, K., Spreitzer, M., Terry, D., Theimer, M. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of UIST'97*, Oct. 1997.
8. Ellis, C., Gibbs, S., Rein, G. Groupware: Some Issues and Experiences. *Commun. ACM* 34(1), 1991.
9. Gray, J., Helland, P., O'Neil, P., Shasha, D. The Dangers of Replication and a Solution. In *Proceedings of SIGMOD'96*, 1996.
10. Horstmann, T., Bentley, R. Distributed Authoring on the Web with the BSCW Shared Workspace System. *ACM Standards View*, Mar. 1997.
11. Hull, R., Llirbat, F., Simon, E., Su, J., Dong, G., Kumar, B, Zhou, G. Declarative Workflows that Support Easy Modification and Dynamic Browsing. In *Proceedings of WACC'99*, 1999.
12. Joseph, A., deLespinasse, A.,Tauber, J., Gifford, D., Kaashoek, M. Rover: A Toolkit for Mobile Information Access. In *Proceedings of 15th SOSP*, Dec. 1995.
13. Karsenty, A., Beaudouin-Lafon, M. An Algorithm for Distributed Groupware Applications. In *Proceedings of 13th ICDCS*, May 1993.
14. Kistler, J., Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, Feb. 1992.
15. Koch, M. Design Issues and Model for a Distributed Multi-user Editor. *Computer Supported Cooperative Work – An International Journal*, 3(3-4), 1995.
16. Lotus Notes. http://www.lotus.com
17. Lynch, N. Distributed Algorithms. *Morgan Kaufmann Publishers, Inc.*, 1996.
18. Munson, J., Dewan, P. Sync: A Java Framework for Mobile Collaborative Applications. *IEEE Computer*, June 1997.
19. Pankoke-Babatz, U., Syri, A. Collaborative Worspaces for Time Deferred Electronic Cooperation. In *Proceedings of GROUP'97*, 1997.
20. Preguiça, N., Legatheaux Martins, J., Domingos, H., Simão, J. System Support for Large-Scale Collaborative Applications. Technical Report, TR-01-98 DI-FCT-UNL, available from [1].
21. Pitoura, E., Samaras, G. Data Management for Mobile Computing. *Kluwer Academic Publishers*, 1998.
22. Schuckmann, C., Kirchner, L., Schümmer, J., Haake, J. Designing Object-Oriented Synchronous Groupware with COAST. In *Proceedings of CSCW'96*, 1996.
23. Simão, J., Preguiça, N., Domingos, H., Legatheaux Martins, J. DAgora: A Flexible, Scalable and Reliable Object-Oriented Groupware Platform. In *Proceedings of ECSCW'97 OOGP Workshop, 1997*, available from [1].
24. Sun, C., Ellis, C. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Prooceedings of CSCW'98*, 1998.