System Support for Large-Scale Collaborative Applications

Nuno Preguiça, J. Legatheaux Martins
Henrique J. Domingos and Jorge Simão

Departmento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Quinta da Torre, 2825 Monte da Caparica, Portugal

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

Technical report **01-98 DI-FCT-UNL**

System Support for Large-Scale Collaborative Applications

Nuno Preguiça, J. Legatheaux Martins
Henrique J. Domingos and Jorge Simão
Departmento de Informática
Faculdade de Ciências e Tecnologia– Universidade Nova de Lisboa
Quinta da Torre, 2825 Monte da Caparica, Portugal
{nmp,jalm,hj,jsimao}@di.fct.unl.pt

ABSTRACT

In this technical report we describe a storage system aimed on supporting collaborative applications in large-scale environments that include mobile computers. To maximize availability, it uses weakly consistent server replication and client caching with a *read any/write any* model of data access. To allow easy management of concurrent updates it provides an object framework that isolates the programmer from the inherent complexity of data replication through code reuse. Type and situation specific conflict detection and resolution are implemented based on support provided by the system and the object framework.

Keywords

Large-scale asynchronous groupware, mobile computing, replicated storage system, log propagation, object framework.

INTRODUCTION

The widespread use of computer networks opened new opportunities for collaboration among people on different geographical places. Several general-purpose Internet-based services have been developed and are in use, notably the *Internet news* and *mail* systems. However, enhanced support for groups of users seeking a specific common objective require specialized applications such as multi-user editing tools, cooperative schedulers and calendars, conferencing systems, workflow systems and others [8,10].

While some of these applications are synchronous, requiring *only* some sort of group-messaging support [30], most of them rely heavily on a storage system to enable information sharing, distribution and composition. Some support systems have been implemented, either for general use (e.g., Lotus Notes [16]), for some specific domain (e.g., LinkWorks [26] for workflow) or for some specific applications (e.g., Iris[18]).

The increasing popularity of mobile and disconnected computing poses new requirements on existing support systems [1,13]. It also opens opportunities for new forms of work – cooperative telework – which in turn exacerbates the need for (primarily asynchronous) collaborative applications.

In this technical report we present a replicated object repository aimed on supporting large-scale heterogeneous computing systems including mobile and disconnected computers. This system can be used as a basis to build asynchronous cooperative applications for distributed groups, managing the inherent complexity associated with data replication and concurrent updates merging.

Our storage system, named DAgora, is based on groups of servers that replicate sets of related objects with a *read any / write any* model of data access in order to maximize data availability. A client caching mechanism is also provided to cope with mobile and disconnected computing.

DAgora also provides an object framework that allows new data types to be composed from reusable predefined components and regular object classes, thus hiding from application programmers the complexity associated with data distribution. Different policies exist to apply concurrently made updates to different replicas, thus allowing each data type to incur only in specific overhead. Flexibility in concurrent updates handling is achieved by our object framework data types composition and DAgora open implementation, that allows new policies to be defined as required.

In the remainder of this technical report we present the requirements we believe essential to support distributed asynchronous groupware and the design choices we have made to fulfil them, some applications we have implemented to evaluate and demonstrate DAgora flexibility, the DAgora repository architecture and the associated object framework, status of the DAgora prototype and intended future work, comparison with related work, and finally some conclusions.

REQUIREMENTS AND DESIGN CHOICES

Consider three different and well-understood asynchronous collaborative applications: a conferencing system, a multi-user editing tool and a group scheduler. All these applications require some sort of data repository to manage their shared data. Due to scale, mobility and disconnection, traditional concurrency control methods are unsuitable. Moreover, we believe that ideally all these applications should allow every user to modify any data concurrently without any restriction (besides coordination and access control mechanisms). Concurrently made updates should be automatically merged in a type- and situation-specific way. Bellow we present some operational scenarios that illustrate these claims.

In a conferencing system, any user should be allowed to reply to a previously existing statement. All concurrent replies should be displayed in a consistent way across different conferencing replicas. In a multi-user editing tool, different users should be able to modify the same structured document. All modifications should be reflected in the final document, creating versions of document elements (e.g. chapter, sections) if concurrent updates to the same elements have been produced. In a group scheduler application, users should be allowed to require new appointments, which must be considered tentative [32] until being committed by some form of automatic global agreement.

From the above scenarios we note that in asynchronous applications users cooperate by accessing and modifying (or applying operation to modify) some shared data. Thus, the storage system should always be a central piece of any support system. Two major and interrelated problems should be tackled: high-availability of data and concurrent updates handling. For some applications other supports should also be provided, such as a notification and a coordination sub-systems [5]. In this technical report we present exclusively the storage system support.

High-availability

In large-scale settings, connectivity among system components is often limited (due to low bandwidth and expensive connections), and at times, even unavailable (due to network and/or machine failures and disconnected computers). Since a single storage site may not be *permanently* reachable from some client machine, replication is required in order to provide high-availability of service. To avoid low write availability in presence of partitioned networks [3] and due to intrinsic groupware properties, weak consistency of replicated data is desirable.

For the above reasons, we have adopted a *read any / write any* model, in which updates can be applied to any replica independently. We have also adopted an epidemic scheme of update propagation among servers [24], 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 taking into consideration connectivity constraints. Some consistency across replicas will eventually be reached (in absence of new updates) as all updates are propagated to all replicas.

Mobile computers, with its inherent reduced connectivity, only exacerbate the above constraints [1,13]. Moreover, the reduced hardware resources available (as presented for instance by personal digital assistants – PDAs), often make impossible and/or undesirable for clients to manage a full unit of replication (that usually corresponds to large amounts of data). By these reasons, DAgora has also adopted a client caching mechanism [17] that allows users in disconnected machines to continue their work, keeping copies only of key data.

Concurrent Updates Handling

Experience and prior research have proven that one of the main issues involved in the management of data in large-scale and mobile computing environments is the handling of uncoordinated/independent concurrent modifications. In database systems, where it has been most actively studied, concurrency control is usually achieved through transactions [4]. Transactions may be implemented either using pessimistic (e.g. locks) or optimistic (e.g. timestamps) techniques. While using optimistic techniques, several transactions are allowed to proceed concurrently until commit time, being aborted if ACID properties have been violated. As ACID properties in conjunction with conventional reliance on primitive read/write semantics are often too restrictive, several techniques have been proposed in order to reduce high abort rates, either exploiting type [29] or application semantics [19].

Several (primarily synchronous) groupware systems have adopted locking mechanisms [27] and optimistic transactions [28]. However, we believe that these mechanisms are unsuitable for asynchronous groupware due to two main reasons:

- In presence of partitioned networks and disconnected computers strong locking mechanisms impose a too restrictive data access model (locks held by some disconnected computer may avoid data access for too long periods). On the other hand, weak locking mechanisms usually lead to abortion.
- Updates granularity in asynchronous applications is usually large, representing a semantically consistent unit (e.g. a new version of a chapter). For this reason, abortion of such amounts of work is often unacceptable.

For the above reasons, we believe that unrestricted concurrency should be the rule, i.e., no restrictions should be imposed to users work (besides coordination and access control mechanisms). Additionally, some merging mechanism must be devised to allow automatic merging of the multiple parallel activity streams [6], taking into account all concurrently made updates (collaboration purpose is contribution merging by definition).

It seems incontestable that, in absence of conflicting concurrent updates, automatic merge should be done. However, the definition and detection of conflicting updates is not trivial. Moreover, whilst there are many actions that can be taken in presence of conflicting updates, the adequate one seems to be type and situation specific. Flexibility should be a key criteria of the mechanisms needed to handle these updates.

Two models exists to store and propagate modifications [9]: state propagation – where each update is immediately applied to data and its effects transmitted; log propagation – where each update, besides being applied to data, is stored in a log which is used to propagate modifications.

State propagation main advantages are: simpler implementation because no log management mechanism is required; and straightforward implementation for a replicated storage system based on a get/put model of access (the common use of file systems). However, log propagation has also several advantages, namely: it enables easy merging of concurrent updates (in absence of conflicts, merging concurrent updates is reduced to applying all updates sequentially); it enables precise conflict detection, based on precise update definition (state propagation often leads to false conflicts detection since it is hard to exactly determine the changes made); and enables flexible conflict resolution by update manipulation (which is allowed by knowledge about operations semantics). Moreover, state propagation may be regarded as a special case of log propagation. Finally, log propagation provides the base (information) support to implement notifications on update operations.

Several systems, like Coda [17], Ficus [11]and Lotus Notes [16], have previously used state propagation. However, experience with those systems demonstrates the complexity of concurrent updates merging based on state propagation, mainly due to mismatching manipulation/structuring granularities and lack of update semantics knowledge. For instance, in Lotus Notes, this complexity leads to a single mechanism of update merging – concurrent versions of the same fields are reflected in the final document as different versions (original implementation of Notes consistently chose one version, discarding others). While this strategy is adequate for several applications, it poses (unsolvable) problems to others.

Mismatching manipulation/structuring granularities are illustrated by the example presented in figure 1. In this example, each document would usually be stored as a single file or Notes rich-text field, but users will usually modify only part of the document. Thus, to produce the expected document not trivial processing would be required to detect differences between different document versions. Moreover, based only on middle versions of the document, i.e., without any further information, it is impossible to determine which versions of the chapters correspond to the new ones, thus making it impossible to automatically produce the expected document.

Due to the above reasons, DAgora object repository is based on log propagation, like Bayou [32] and Rover [14]. However, unlike Bayou that presents a relational database data model, Rover and DAgora allow generic object definition. Thus, they enable definition of complex and *suitable* data types, without imposing data to fit the available data model. Restricting all applications to a single data model lead to unnecessary and complex exercises of data manipulation, trying to squeeze data structure to the available model, which sometimes turns out to be an unsolvable situation.

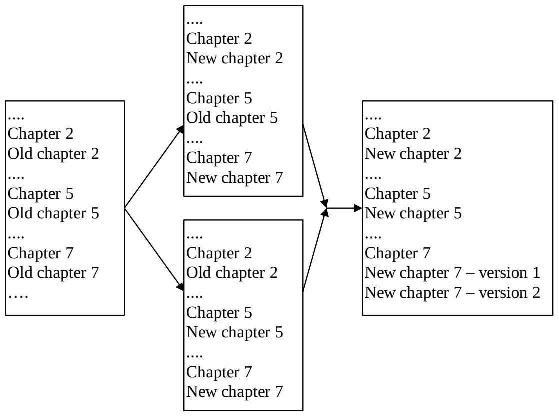


Figure 1 – Expected evolution of a concurrent editing session. The middle versions represent two concurrent modifications to the same document.

Moreover, and unlike Rover, DAgora presents an object framework that enables easy and flexible object construction. This open object framework is constituted by several components that manage the inherent complexity associated with data types implementations (notably, updates logging and ordering), thus restricting work involved in data type construction almost to common object definition. For each of these components several predefined semantics are available and others can be defined, thus allowing data types to exhibit different updates management policies. For this reason, and unlike other previous systems, this framework enables each data type to incur only on specific overhead dependent on specific behavior.

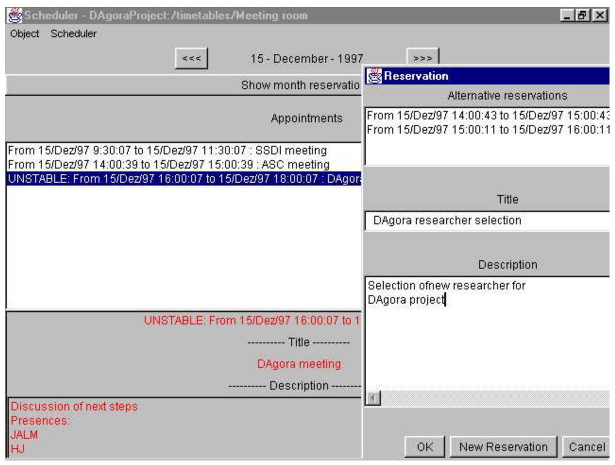
EXAMPLE APPLICATIONS

To demonstrate system's operation and to evaluate its requirements and mechanisms, we have developed two applications: a scheduler and a multi-user document editor. We believe that these application, although very simple and implementing only a subset of functionalities required for a production-level version, highlight the storage system and concurrent update management requirements for different types of applications.

Scheduler

The scheduler application enables users to reserve resources, such as meeting rooms, projectors, etc. At most one reservation may be granted for the same period of time. This scheduler is only intended for use after people have decided the set of acceptable periods of time for which they want to reserve the given resource. It does not help people to agree, for instance, in a mutual agreeable period of time for a meeting.

Users interact with a graphical interface, presented in figure 2, observing which periods are already reserved. Two kinds of reservations exist: committed and tentative. While for committed reservations displayed times are unchangeable (unless reservation is deleted), for tentative reservations displayed times are dependent on the existence of other reservations, yet unknown, that reserve the same times.



```
15 - December - 1997

From 15/Dez/97 9:30:07 to 15/Dez/97 11:30:07 : SSDI meeting
From 15/Dez/97 14:00:39 to 15/Dez/97 15:00:39 : ASC meeting
UNSTABLE: From 15/Dez/97 16:00:07 to 15/Dez/97 18:00:07 : DAgor

UNSTABLE: From 15/Dez/97 16:00:07 to 1
---------- Title ----------
DAgora meeting
---------- Description --------
Discussion of next steps
Presences:
JALM
HJ

Alternative reservations
From 15/Dez/97 14:00:43 to 15/Dez/97 15:00:4
From 15/Dez/97 15:00:11 to 15/Dez/97 16:00:11
Title
DAgora researcher selection
Description
Selection ofnew researcher for
DAgora project
```

Figure 2 – Scheduler application. Main window presents known reservations. Reservation window is used to set a new appointment.

Users make their reservations filling a form indicating the set of alternative times for which they intend to reserve the resource and giving a brief description of the reason (for a meeting room, it may be the description of the meeting that will take place).

The underlying storage system of the scheduler application should transparently support the existence of tentative and committed versions of the same resource calendars. Moreover, it should be able to commit new appointments as soon as possible – in presence of mobile computers, this calls for a primary replica calendar. Although different calendar replicas may, in a given moment, present different values (due to a different set of known and committed updates), these values should be coherent – a committed appointment should be scheduled for the same period of time in all replicas.

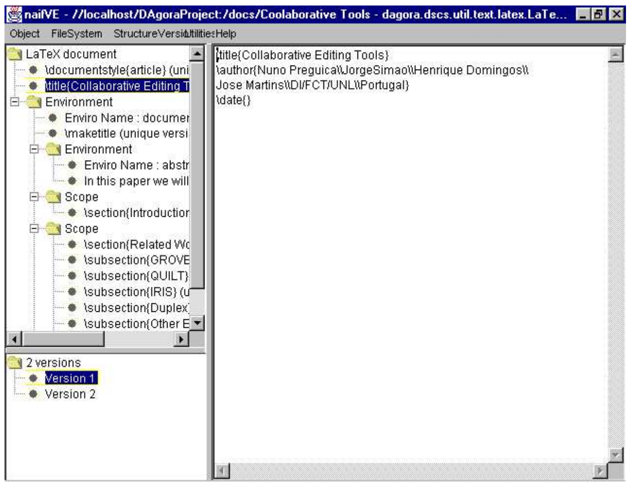
Multi-user Document Editor

The editor application allows users to produce documents cooperatively. Users may modify documents without any restriction, though we expect coordinated users to modify different parts of the same documents. Although documents are stored as single data units in our repository, they are structured in independent components, as proposed by several collaborative editing tools [18,23].

Two base components are used to build documents: containers and leaves. Containers are sequences of other containers and/or leaves and define documents hierarchic structures. Leaves represent atomic units of data that may have multiple versions and may be of different types. A document is a hierarchical composition of these components, representing the document structure.

For instance, a LaTeX document has several containers and several leaf types, as illustrated in figure 3. Each leaf component defines its type and default associated editor to be used. Each container defines its initial composition and possible new components.

Currently a small set of document types is implemented, including a generic hierarchic structured document, a LaTeX document and a Java source document. To allow the same document editor to manipulate different documents types (based on the same base components), reflection information is provided in the derived components. Export and import functions allow interoperability with file system based tools.


naifVE - //localhost/DAgoraProject:/docs/Coolaborative Tools - dagora.dscs.util.text.latex.LaTe...
Object
FileSystem
StructureVersiUtilitiesHelp

```
LaTeX document
  \documentstyle{article} (uni
  \title{Collaborative Editing T
  Environment
    Enviro Name : documer
    \maketitle (unique versi
    Environment
      Enviro Name : abstr
      In this paper we will
    Scope
      \section{Introductior
    Scope
      \section{Related Wc
      \subsection{GROVE
      \subsection{QUILT}
      \subsection{IRIS} (u
      \subsection{Duplex
      \subsection{Other E

2 versions
  Version 1
  Version 2

\title{Collaborative Editing Tools}
\author{Nuno Preguica\\JorgeSimao\\Henrique Domingos\\
Jose Martins\\DI/FCT/UNL\\Portugal}
\date{}
```

Figure 3 – DAgora editing tool with a LaTeX document.

In figure 3 we present the edition of a LaTeX document. The editor window is divided in three areas: structure, version and editing areas. This allows easy document structure and leaf version navigation

Concurrent updates to document structure are merged sequentially applying both modifications – the rationale being that if two users (for instance) add a new chapter, they are probably adding two different chapters. When users intend to add the same chapter, they should merge them later. Deletion is handled through a pair of operations *mark as deleted / discard* in order to avoid delete / modify conflicts and to guarantee that no component of a document is deleted while is considered of interest by some user. Updates to same leaf elements are merged through version creation.

To allow the outlined concurrent updates handling policy, the repository should be able to order all updates – so that all replicas could evolve coherently. However, all updates known in each replica should be immediately applied to maximize awareness about users contributions. Moreover, the storage system should provide a mechanism to detect concurrently made updates so that, version creation could be correctly handled.

As can be seen from these two examples, the programmer should be able to select very flexibly the way concurrent updates are handled in each case. Besides the above-mentioned requirements, the need for high-availability is implicit in both applications. Users must be able to access the same data objects, even in presence of network and/or server failures. When disconnected, users must be able to access and modify data objects. Updates will be reintegrated as soon as possible (i.e., when user reconnects to system).

In the next section we will present the structure of the repository and the way it operates. Later we will present the object framework allowing flexible concurrency handling.

REPOSITORY STRUCTURE / OPERATION

DAgora storage system is a distributed object repository based on a *client / replicated server* architecture. DAgora manages objects, known as coobjects (from collaborative objects). Coobjects are organized in sets, known as volumes. Each coobject belongs to a single volume and has a unique identifier relative to the volume. Each volume represents a collaborative workspace, containing coobjects relative to a given workgroup and/or cooperative project.

DAgora applications run on client machines, allowing users to collaborate through concurrent modification of the same coobjects. Coobjects may be rather complex (such as a document or a scheduler calendar) and be implemented as an arbitrary set of regular objects. Applications employ a *get / modify locally / put changes* model of data access: they obtain private and local copies of coobjects, modify them by usual methods invocations, and finally explicitly export updates made.

DAgora architecture is depicted in figure 4. Servers replicate volumes of coobjects in order to guarantee high-availability in presence of networks and/or server failures. Clients cache key coobjects so that users may continue their work, even while disconnected.

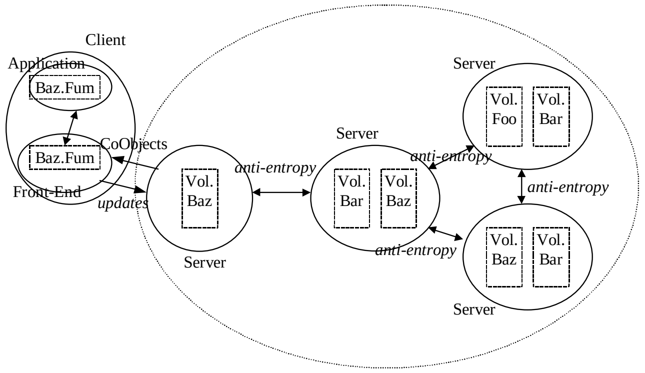


Figure 4 - DAgora object repository architecture.

When an application requests a given coobject, if it is not present in client cache, it is fetched from a server. A private copy of the coobject is created and handed over to the application. Application uses the coobject as a regular object, invoking its methods to query and modify coobject's state. Finally, users may record explicitly their changes (or not). Updates made by applications are registered as sequences of methods invocations (logged internally and transparently by coobjects). These sequences are stored in stable storage at clients machines, and later sent to a server.

As outlined, clients are responsible to fulfil users requests. To this end, they manage a local cache, in order to guarantee that coobjects requested by applications will be mostly available. Coobjects are obtained from any server that replicates them. Clients make their best effort to guarantee that copies handed over to application are up-to-date with some server replica.

Clients also manage a stable log of invocations to volumes. These invocations represent sequences of updates performed by users to coobjects. Updates are forwarded to servers that replicate each volume as possible. This mechanism is similar to a deferred RPC (with no return parameters).

Upon arrival of sequences of updates from a client machine, the server hands them over to the coobjects local replica. Coobjects implementations are responsible for storing and applying them. Different coobjects will apply updates obeying different constraints, usually guaranteeing that all replicas will eventually converge (as all updates are propagated to all replicas). Servers establish pair-wise occasional communications to propagate newly received updates, which are obtained from and delivered to coobjects local replicas. As a consequence of this mode of operation, replicas of the same coobject may differ, in each moment, in different servers, but they will eventually converge.

As outlined, servers, besides interacting with clients, are responsible to manage volume replication. Each volume is replicated by a variable set of servers. DAgora servers propagate updates among themselves, synchronizing their coobjects replicas, during pair-wise communications, known as *anti-entropy* sessions [24]. The two servers involved in a session exchange updates so that when they finish, both agree on the set of updates known, for each coobject. Epidemic algorithms theory guarantees that as long as servers and communication paths form a connected graph (i.e., as long as servers are not permanently partitioned or failed) each update will eventually reach all servers. In absence of new updates performed by clients, all servers will eventually know all updates and hold the same data.

DAgora protocol [25] maintains and exchanges summaries of updates seen in each server for each coobject (represented as timevectors) in order to minimize updates exchanged during *anti-entropy* sessions. Additional acknowledgment summaries are used to purge updates from coobjects logs. DAgora protocol enables anti-entropy sessions to occur over multiple transports, including asynchronous methods of communications, such as e-mail. Thus, it allows servers lodged in mobile computers to synchronize with each other without need for direct connections between them.

The group of servers that replicate each volume may vary as a result of users (system administrators) explicit orders. To this end, DAgora uses a well-known coobject in each volume to track and propagate volume membership changes. Join and leave protocols are light-weighted imposing communication with only one server. Membership changes are propagated during usual *anti-entropy* sessions.

In order to promote tailorability and flexibility we have made a clear division of responsibilities between the system core (stable and unmodifiable) and the coobjects implementations (which are under programmer control) System core is responsible to: fetch copies of coobjects from servers and cache them in clients machines; create private copies of coobjects to be handed over to clients applications; send to a server the sequences of updates made by applications to coobjects (after explicit save), storing them temporarily in clients machines, if necessary; establish communications between servers to propagate coobjects updates to all replicating sites.

Coobjects implementations are responsible to: log updates made by applications (in clients machines); store updates delivered by system core (in servers machines); expose logged and stored updates; order and apply known updates. As said before, all these actions are under programmer control, through an open (co)object framework provided in DAgora. This framework will be presented in next section.

OBJECT FRAMEWORK

Updates management imposes a heavy burden on coobjects implementation. To alleviate programmers from much of the associated complexity we have defined an object framework. This framework allows inexperienced programmers to create coobjects relying on predefined components (sub-objects) to impose consistency among replicas, thus hiding its inherent complexity.

This object framework structures each coobject in five disjoint components (objects), each one with a well-defined interface. These components are the following: capsule, data, attributes, log, and log-ordering (figure 5). Our open implementation allows new components with different semantics to be implemented, independently from each other.

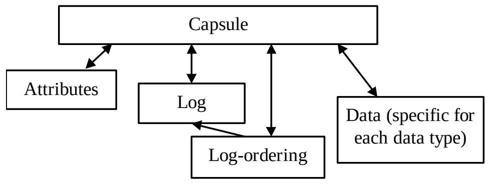


Figure 5 - DAgora object framework.

Capsule

Capsules aggregate the components of a coobject. They implement the interface used by the system core to interact with coobjects. Usually, a capsule just coordinates and redirects invocations to the appropriated components.

Two capsule implementations are available. One, is the normal capsule that aggregates an attributes object, a data object, a log object and a log-ordering object. This is the usual configuration of a coobject. The second one aggregates an attributes object, two data objects, two log-ordering objects and a log object. This second capsule is used to implement coobjects that store two versions of the data, independently from data type definitions. Our scheduler application uses this capsule to store the tentative and committed calendars.

Attributes

The attributes component is used to store general-purpose information relative to the coobject and meta-information relative to the replication process. Two implementations are available: a simple and an extended one. The extended implementation should be used with sequencer based orderings. It stores information about sequencer identity, and defines methods for its management. Simple implementation should be used otherwise. These classes may be extended to defined type-specific attributes.

Log

The log is used to log and store updates performed by users. It has a dual function: in clients, it logs updates temporarily; in servers, it stores updates received directly from clients and/or from *anti-entropy* sessions. For each sequence of updates logged or stored, log adds additional information necessary to order updates. With this information it is also possible to trace the updates precedence graph. This information is used in our multi-user document editor to execute version management.

Similar to the attributes component, two implementations are available: a simple and an extended one. The extended one should be used with sequencer based orderings. Both log implementations execute compression while logging updates if updates properties – commute and mask – are available (masked updates are discarded).

Log-ordering

The log-ordering component is used to determine the order by which updates should be applied to the coobject. It has a dual function: in clients, it determines if updates should be applied immediately to coobject’s private copy (usually, updates are applied immediately to allow users to observe the expected results from their actions); in servers, it orders the application of stored updates. Log-ordering component uses the information added by log to establish an order among updates.

Currently, several log-ordering components are available, namely: no order, causal order, total order based on a sequencer replica, total causal order based on stability tests, total causal order using undo/redo [15], total causal order based on a sequencer replica. **No order** and **causal order** impose almost no delay on update application, thus enabling immediate commitment of updates in servers. However, as it is often hard to guarantee replicas consistency using these orderings, **total order** is often required. Several techniques were implemented to guarantee total order.

When no sequencer is used (**stability based techniques**) to commit updates, each server must gather enough information about other servers to establish the total order. This information is propagated during *anti-entropy* sessions. Unfortunately, as it requires feedback from all replicas, one simple disconnected replica may prevent any update from being committed. To mitigate this problem, an **optimistic undo/redo** implementation is available, where all updates are applied immediately, being undone and redone later, if a new update is received that should have been ordered prior to an already executed one. This implementation is used in our multi-user document editor, thus allowing users to be aware of all known and executed updates.

Alternatively, a **sequencer based ordering** is available, allowing updates to be committed since the sequencer replica is reachable (even in presence of multiple disconnected replicas). With this implementation, a coobject replica is responsible for defining the official commit order for all received updates (which are propagated as usual, during normal anti-entropy sessions). Our scheduler application uses this ordering to commit appointments.

Data

The data component implements the real data type being created, with its associated state and operations. With current log implementations, which are based simply on updates ordering, operations are responsible for detecting and solving conflicts among concurrent updates. Our experience suggests that for most applications careful operations definition associated with regular operations preconditions check is enough (our scheduler application is an example).

Some others may require more complex updates conflict detection and resolution. Detecting the existence of concurrent updates is easy, based on information added to updates by the log component and the summaries of applied updates (our multi-user document uses this facility). In the unlikely situation in which the above facilities are not enough, concurrent updates may be accessed from log to determine existing conflicts and to execute update-specific conflict resolution. The above characteristics allow very flexible management of concurrent updates, although we expect that most applications will not need to resort to all those possibilities.

Using The Object Framework

To create a new coobject type, a programmer must define the data component and select the desired components implementations. This allows easy data-type construction, through massive code reuse.

In figure 6, we present the code needed to implement the coobject used in our scheduler application. *SchedulerData* implements a simple scheduler object, as it would usually be implemented. Two modifications are required: objects must extend *dagora.dscs.DagoraData* and implement *java.io.Serializable* (which requires no new method definition); public methods that may modify the object state must have a new qualifier – *loggable*. *SchedulerCapsule* defines the components used in the coobject, and extends the selected capsule.

```
public class SchedulerCapsule
        extends dagora.dscs.TwoVersionsCapsule
        implements java.io.Serializable
{
    public SchedulerCapsule() {
        attrib = new dagora.dscs.AttribSeq();
        logcore = new dagora.dscs.LogCoreSeqImpl();
        commitData = new SchedulerData();
        commitlogorder = new dagora.dscs.LogTotalSeqCausal( false);
        tentativeData = new SchedulerData();
        tentativelogorder = new dagora.dscs.LogNoOrder( true);
    }
}

public class SchedulerData
        extends dagora.dscs.DagoraData
        implements java.io.Serializable
{
    public Vector appointments( int year, int month, int day) {
        /* method code here */
    }
    public loggable void insertReservation( ReservationEntry[] altRes) {
        /* method code here */
    }
    public loggable void removeReservation( ReservationEntry res) {
        /* method code here */
    }
}
```

Figure 6 – Scheduler coobject implementation.

Coobjects definitions are preprocessed to generate *standard* Java code, which is later compiled using standard development tools. Coobjects using undo/redo orderings are required to define undo methods. Ordering information associated with each update may be accessed by parameters implicitly added to *loggable* methods.

STATUS AND FUTURE WORK

The implementation of the DAgora storage system has two distinguishable (yet complementary) components: the system core and the object framework. While system core is a rigid component, the object framework requires the ability to evolve dynamically (while system is running). Moreover, as the system has been designed for large-scale settings, heterogeneity is also a requirement. To fulfil the above requirements we have decided to implement DAgora using Java [31]. The security mechanisms available were an additional motivation, enabling some control over coobjects implementations – as coobjects execute on server machines some security restrictions are necessary.

We have a complete implementation of the DAgora storage system based on Java JDK 1.1. DAgora applications were also implemented in Java (using Java JDK 1.1.2 and Swing). We expect to allow our implemented applications to be tested from [12] using an adequate browser, as soon as a web version of our client is implemented.

Such as other distributed systems, DAgora performance depends on several factors, namely, the location of servers and clients, the communication infrastructure and the amount of data processed. Specifically related with DAgora implementation, preliminary results [25] have shown that its performance is highly influenced (and dominated) by the Java object serialization process. To improve performance we intend to investigate alternative methods to store objects automatically. Nevertheless, we believe that system enhanced functionality advantages overcome its performance drawbacks (compared to traditional storage systems).

Many potential work directions were revealed during the course of our work, such as the determination of optimal caching and anti-entropy policies, the introduction of session guarantees and resource consumption restrictions to coobjects, alternative access mechanism to large coobjects based on partial replication or remote access. Other issues that we intend to explore in the future include the creation of generic notification mechanisms to provide users with shared feedback of activities related with coobjects (log component provides this information). Suitable access control and security mechanisms must also be addressed. Coordination among users is other issue that requires further investigation in large-scale settings. However, the next step in DAgora evolution will be the creation of new applications and associated coobjects and components to further refine our basic model. We are specially interested in using the updates precedence graph to deal, automatic and transparently from data objects, with concurrent updates, either discarding conflicting updates, creating multiple version, merging conflicting updates [21], or executing updates transformations [7].

RELATED WORK

Several systems have been developed to manage data in large-scale environments. Notably, database systems [4], based on transactions, define a widely used and well-understood model of concurrency control. Some systems [20] have even introduced extensions to support disconnected operation. However, as we have already discussed, transactional techniques are not suitable for asynchronous groupware.

Lotus Notes [16] is a replicated document database. Documents have a record-like structure composed by typed fields defined in forms. Notes architecture is composed by a group of servers that replicate databases (sets of documents) using epidemic techniques and by clients that cache documents. Notes propagates fields values, handling concurrent updates by creation of multiple versions of data that must be manually merged. We believe that this approach is rather inflexible and often inadequate, being automatic conflict resolution preferable and often possible.

Coda [17] is a replicated file system with support for disconnected clients. It also supports low bandwidth networks and intermittent communication. While disconnected, clients log all updates to the file system, which are replayed on reconnection. System executes automatic update conflict resolution for directories. Application-specific programs can be provided for automatic resolution of file updates conflicts. However, lacking of update semantics – files are modified as complete untyped byte streams – makes update merging rather difficult and sometimes impossible. Concerning Coda's architecture, we believe that requiring clients to synchronize all accessible server replicas imposes an excessive overhead to clients on large-scale settings.

Ficus [11] is a replicated file system that uses similar conflict resolution policies, but uses an epidemic scheme to propagate updates among servers. The shortcomings presented by the above-mentioned systems to handle concurrent updates are the result of state propagation strategy. Next we present two systems that use an update propagation strategy: Bayou and Rover.

Bayou [32] is a replicated database system to support data-sharing among mobile users, with an architecture similar to Notes. 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. Our system allows emulation of Bayou's main characteristics through coobject definition. Moreover, as it allows specific data types definition it does not impose data to fit the available model, allowing more flexible and suitable solutions – for instance, implementing our editor applications with Bayou would have been rather cumbersome.

Rover [14] combines relocatable dynamic objects (RDO) and queued remote procedure calls (QRPC) to provide information access for mobile clients. Each RDO has a home server and may be imported by clients. While imported, updates are logged and performed locally. When the RDO is exported, logged updates are applied to the replica at the home server. Resolution of detected conflicts is achieved at server by calling type-specific methods. QRPC are used to execute all communications between clients and servers, allowing non-blocking RPCs even while disconnected. We believe that our system is more suitable for large-scale settings due to server replication (in conjugation with client caching). The object framework also allows easier data types definition and more flexible handling of concurrent updates.

Several distributed object systems have been previously developed and present some form of concurrent update handling. Some of them [2] even present object frameworks decomposing object operation. Some real-time collaborative systems [7,21,27,28] also present concurrency control mechanism to handle concurrent updates. However, these systems are usually real-time, designed for low granularity objects with different requirements, and present solutions unsuitable for asynchronous large-scale settings.

Iris [18] present an architecture for large-scale collaborative editing. The associated storage system has many similar design choices when compared to DAgora. However, it is not a general-purpose data storage.

Sync [21,22], a framework for mobile collaborative applications, present an interesting model of concurrent update handling and object construction. However, we believe that lack of server replication makes it less suitable for large-scale settings.

CONCLUSIONS

The DAgora storage system is an object repository for large-scale environments that include mobile computers. Our architecture has been designed with the goal of maximizing data availability. It combines two major techniques: server replication and client caching. Servers replicate volumes of coobjects with a read any/write any model of data access. Coobjects are modified through method invocation, and updates are propagated among servers using an epidemic scheme requiring only pair-wise occasional communications. Clients cache key coobjects to allow users operation even while disconnected (and to improve performance).

Although architecture is important to achieve availability, experience has proven that usability of the system is largely dependent on efficiency of concurrent updates handling - in some systems, concurrent updates that can not be automatically merged lead to normal access failure and thus to a even lower availability. Thus, automatic conflict resolution is not just desirable, it is necessary and fundamental.

DAgora presents a set of characteristics that allows users to implement a wide range of updates handling policies. First, we use log propagation instead of state propagation. This provides precise update information, allowing precise conflict detection and update semantics usage. Second, coobjects automatically add to each update enough information that allows the precedence graph of updates to be traced. This allows the precise determination of concurrent work paths. Third, coobjects store updates allowing *users* access. This enables very flexible handling of concurrent updates, allowing updates transformation.

Moreover, DAgora provides an open object framework that divides coobjects operation in several independent and reusable components, thus alleviating programmers from most of the inherent complexity associated with the above characteristics. Common coobject creation can be reduced almost to regular object implementation and selection of the desired semantics for the other components.

Although DAgora characteristics enable very complex updates handling, experience with implemented coobjects indicates that most data types will require only simple (and easy to implement) techniques of concurrent updates merging. These techniques consists in: imposing an adequate (usually total) order to update application; using vector timestamps associated with each update to detect concurrent updates; adding update-specific conflict detection to each update code; adding update-specific conflict resolution to each update code.

REFERENCES

1. R. Alonso, H. Korth Database system issues in nomadic computing. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, May 1993.
2. G. Brun-Cottan, M. Makpangou. Adaptable Replicated Objects in Distributed Environments. *INRIA Rapport de recherche nº 2593*, May 1995.
3. B. Coan, B. Oki, E. Kolodner. Limitations on Database Availability when Networks Partition. In *Proceedings 5th ACM Symposium on Principles of Distributed Computing*, August 1986.
4. S. Davidson, H. Garcia-Molina, D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, C-31, 1982.
5. H. J. Domingos, J. Legatheaux Martins, J. Simão, N. Preguiça. Coordination and Flexible Synchronicity in Large Scale CSCW. In *Proceedings CRIWG97*, Madrid, October 1997.
6. P. Dourish. The Parting of the Ways: Divergence, Data Management and Collaborative Work. *Proceeding of the 4th European Conference on CSCW*, 1995.
7. C. Ellis, S. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, June 1989.
8. C. Ellis, S. Gibbs, G. Rein. Groupware - Some Issues and Experiences. *Communications of the ACM*, 34(1):38-58, January 1991.
9. R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4), 1992.
10. J. Grudin. Computer-Supported Cooperative Work: History and Focus. *IEEE Computer*, May 1994.
11. R. Guy, J. Heidemann, W. Mak, T. Page Jr., G. Popek, D. Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceedings*, June 1990.
12. http://dagora.di.fct.unl.pt
13. T. Imielinski, B. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communications of the ACM*, 37(10), October 1994.
14. A. Joseph, A. DeLespinasse, J. Tauber, D. Gifford, M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
15. A. Karsenty, M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

16. L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, I. Greif. Replicated Document Management in a Group Communication System. In *Proceedings of the 2nd ACM Conference on CSCW*, September 1988.

17. J. Kistler, M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

18. M. Koch. Design issues for a distributed multi-user editor. *Computer Supported Cooperative Work – An International Journal*, 3(3-4):359-378, 1995.

19. H. Korth, G. Speefle. Formal model of correctness without serializability. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, 1988.

20. B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, L.Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1996.

21. J. Munson, P. Dewan. A Concurrency Control Framework for Collaborative Systems. In *Proceedings of the 1996 ACM Conference on CSCW*, November 1996.

22. J. Munson, P. Dewan. Sync: A Java Framework for Mobile Collaborative Applications. *IEEE Computer*, June 1997.

23. F. Pacull, A. Sandoz, A. Schiper. Duplex: A Distributed Collaborative Editing Environment in Large Scale. In *Proceedings of the 1994 ACM Conference on CSCW*, October 1994.

24. K. Petersen, M. Spreitzer, D. Terry, M. Theimer, A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

25. N. Preguiça. Repositório de Objectos de Suporte ao Trabalho Cooperativo Assíncrono. MSc thesis, 1997 (in portuguese).

26. W. Prinz, S. Kolvenbach. Support for Workflows in a Ministerial Environment. In *Proceedings of the 1996 ACM Conference on CSCW*, November 1996.

27. M. Roseman, S. Greenberg. GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications. In *Proceedings of the 1992 ACM Conference on CSCW*, 1992.

28. C. Schuckmann, L. Kirchner, J. Schummer, J. Haake. Designing Object-Oriented Groupware with COAST. In *Proceedings of the 1996 ACM Conference on CSCW*, 1996.

29. P. Schwartz, A. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, August 1984.

30. J. Simão, J. Legatheaux Martins, H. J. Domingos, N. Preguiça. Supporting Synchronous Groupware with Peer Object-Groups. In *Proceedings* of 3rd COOTS, June 1997.

31. Sun Microsystems. The Java Language Environment – A White Paper. October 1995.

32. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.