Published in *Proceedings of the Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, April 1999.

FLEXIBLE DATA STORAGE FOR MOBILE COLLABORATIVE APPLICATIONS

Nuno Preguiça, J. Legatheaux Martins, Henrique J. Domingos
Department of Computer Science
Faculty of Sciences and Technology - New University of Lisbon
Quinta da Torre, 2825 Monte da Caparica, Portugal
{nmp,jalm,hj}@di.fct.unl.pt

KEY WORDS

Mobile computing; asynchronous collaborative applications; distributed data storage; object framework.

ABSTRACT

In this paper we describe a flexible object storage system aimed at supporting collaborative applications in large-scale environments that include mobile computers. We present an integrated solution to two major problems that arise in such environments: data availability and concurrency control. The first is tackled by the flexible combination of weakly consistent server replication and client caching. The second is tackled through an open object framework that enables easy object development using type specific conflict detection and resolution. This object storage serves as a supporting platform to produce new distributed and mobile collaborative applications.

INTRODUCTION

Mobile computing is characterized by some intrinsic constraints related with available connectivity, power and hardware resources [14]. Despite the impressive progress in hardware and communication technology [6], mobile hosts have to face lower and highly variable bandwidth capabilities when compared with those of stationary computers. Moreover, mobile computers incur in periods of complete disconnection.

As users must be able to access data to perform useful work, the utility of any computer depends largely on the efficiency of the underlying storage system. In mobile environments, where periods of complete disconnection are frequent, data availability must be provided relying on local replicas of data. To support collaborative applications effectively, users must be allowed to perform their contributions in any mobile host without any restrictions, even when disconnected. These concurrent updates must be subsequently merged, and their intended effects taken into account, to produce the final state of the shared data.

In this paper we present the DAgora storage system, which has been designed to support asynchronous collaborative applications in large-scale settings that include mobile computers. It uses server replication and client caching with a *read any / write any* model of data access to maximize availability. Log propagation has been adopted to promote type-specific concurrent updates merging.

To simplify the development of mobile collaborative applications, DAgora also provides an object framework that allows new data types to be composed from reusable predefined components and regular object classes. This object framework hides from application programmers much of the complexity associated with data distribution and concurrency control issues related with the read any/write any model of data access. Different policies exist to apply concurrently made updates to different replicas and new ones may be defined as required by new applications.

In the remainder of this paper we present: the DAgora requirements; a global overview of the system; the object framework; comparison with related work; and conclude with some final remarks.

MOTIVATION

Consider three different asynchronous collaborative applications that can be used in a mobile environment: a conferencing system, a multi-user editor and a group scheduler. All these applications require some sort of data repository to manage their shared data. 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 document editor, different users should be able to modify the same structured document. All modifications should be reflected in the final document. Multiple versions of each document element (e.g. chapter, sections) must be created if concurrent updates to the same element have been produced. In a group scheduler application, users should be allowed to enter new appointments, which must be considered tentative until being committed by some form of automatic global agreement.

From the above scenarios we note that users cooperate by accessing and modifying (or applying operations to modify) some shared data. Moreover, all those applications allow different users to concurrently modify data without restrictions. To provide high availability, different users may have access to inconsistent data replicas. To support these properties, DAgora is based on weakly consistent data

replication with a *read any / write any* model of data access.

Our goal in designing DAgora was to provide system support to ease the development of asynchronous collaborative applications for mobile environments. To this end, providing data availability is just one of the problems involved. Another one, perhaps more difficult to solve, is the handling of concurrent updates in a weakly connected system based on weakly consistent replication. Several problems are involved: interpretation of results in user application; consistency among servers; respect by the user's intentions when concurrent updates are merged.

Distributed file systems, such as Coda [10] and Ficus [12], use system and user defined conflict resolution programs to merge divergent replicas. These systems work very well in environments with few conflicts and their strategy is quite effective for objects with simple semantics – e.g. file directories. They have proven the value of semantic conflict detection and resolution. However, experimental results (30% unsolved update/update file conflicts [12]) suggest that the resolution of conflicts based on simple state propagation may be very difficult for complex objects.

We believe that the observed shortcomings can be overcome executing conflict resolution at the granularity of individual operations and further exploiting domain-specific knowledge (thus extending the principles applied in Coda and Ficus). Several systems, such as Bayou [15], Rover [7] and Sync [11], use different approaches based on the above principles. In related work section we discuss the reasons why we believe our system is more suitable for the target environment.

In the DAgora storage system, updates performed by users are propagated to a server and among all the servers as method invocations – log propagation model [4]. The effect of each update in each data replica is determined by the execution of the associated method in each server. Due to the availability of precise updates' information, this model simplifies conflict detection and enables the implementation of different conflict resolution policies.

From the applications that we have briefly described in the beginning of this section, we can see that distinct applications handle concurrent updates in different ways. To support this characteristic, DAgora allows each data type to define specific policies to handle concurrent updates.

To face the complexity associated with the DAgora model, we have defined an open object framework. This object framework consists of several components that manage the inherent complexity associated with new data types (notably, the logging and ordering of updates). For each of these components several predefined semantics are available and others may be defined. Therefore, the development of a data type for a new application is greatly simplified through the reuse of available solutions.

In the next sections we will present an overview of the DAgora storage system and detail the DAgora open object framework.

SYSTEM OVERVIEW

The DAgora distributed storage system manages objects, known as coobjects – from collaborative objects. These coobjects may be rather complex (such as documents or scheduler calendars) and be implemented as arbitrary compositions of regular objects. Sets of related coobjects are grouped in volumes representing collaborative workspaces and storing the data associated with a given workgroup and/or cooperative project.

To provide high availability of data and support for workgroups that are distributed across several physically disjoint locations, volumes of coobjects are replicated by groups of servers. The location of servers must be selected to decrease users' connectivity requirements and nothing prevents a powerful mobile computer from hosting a DAgora server.

Since traditional replication schemes providing one copy serializability and strict consistency yield unacceptably low write availability in partitioned networks or in the presence of disconnected computers [1], weak consistency of replicated data is desirable. Consequently, DAgora has adopted a model in which clients can read and write to any replica independently – read any / write any model.

Updates are propagated among servers during occasional, pair-wise communications known as anti-entropy sessions [4], thus taking into consideration the connectivity characteristics of mobile environments. This epidemic scheme guarantees that each server eventually receives all updates from every other, either directly or indirectly. Therefore, consistency among data replicas may be eventually achieved in a quiescent state when all updates have been propagated to all replicas.

To increase data availability and system usefulness for mobile users, DAgora implements a caching mechanism in clients. Therefore, users that work on computers with reduced hardware resources, such as PDAs, may have access to data even while disconnected. To face the inherent heterogeneity of mobile environment, server replication and client caching mechanisms may be used and combined to implement different storage system configurations. In figure 1, we depict the DAgora storage system architecture with a configuration that presents a static computer, a laptop computer and a small PDA.

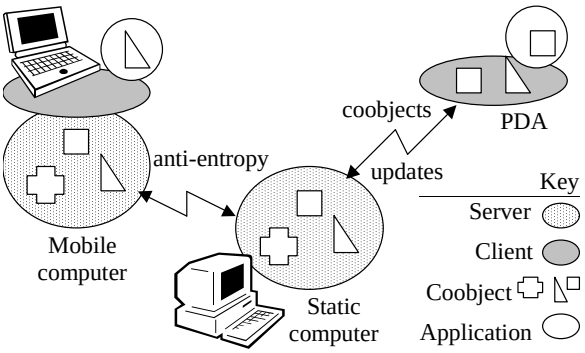

Figure 1 – DAgora architecture composed by three computers with different configurations. Coobjects are replicated by servers, cached by clients and manipulated by users' applications.

Applications employ a *get / modify locally / put changes* model of data access. Private copies of coobjects are obtained through the client component and they are modified by usual method invocations. Updates are exported to a server using a *store-and-forward* model: the client component stores the updates until incremental propagation is possible.

The DAgora storage system is based on a clear division between system core and coobjects implementations. The system core is responsible to provide high availability of data and to guarantee that updates performed by users are propagated to all replicas. It is composed of server and client components as depicted in figure 1. Coobjects implementations are responsible to handle updates. In clients, they have to log executed updates. In servers, they must store updates delivered by the system core, expose them for server replication and apply them to the replica's state. As all these actions are under programmer control, specific solutions may be developed for different data types.

OBJECT FRAMEWORK

The management of updates imposes a heavy burden on the coobjects. To alleviate programmers from much of the associated complexity we have defined an object framework. The DAgora object framework structures each coobject in five disjoint components (objects), each with a well-defined interface. These components are: capsule, data, attributes, log, and log-ordering (figure 2).

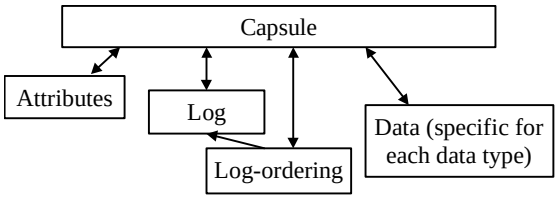

Figure 2 –The DAgora open object framework.

This framework allows inexperienced programmers to create coobjects by relying on predefined components to impose consistency among replicas. New components, with different semantics, may be implemented as required for new applications.

Attributes

The component "attributes" 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 (see "log-ordering" section). It stores information about sequencer identity and defines methods for its management. The simple implementation should be used otherwise. These classes may be extended to defined type-specific attributes.

Log

The component "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, the log adds additional information – a version vector and a timestamp – necessary to order updates. With this information it is possible to trace the update precedence graph.

Similar to the component "attributes", two implementations are available: a simple and an extended one. The extended one should be used with sequencer-based orderings (see "log-ordering" section). Both log implementations execute compression while logging updates if update properties – commute and mask [8] – are available (masked updates are discarded).

Log-ordering

The component "log-ordering" 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. The component "log-ordering" uses the information added by the "log" to establish an order among updates.

Currently, several implementations 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 [8], and 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 replica consistency using these orderings, **total order** is often required. Several techniques were implemented to guarantee total order.

When no sequencer is used to commit updates (**stability-based techniques**), 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.

Alternatively, a **sequencer-based ordering** is available, allowing updates to be committed provided that 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).

Capsule

The component "capsule" aggregates the components of a coobject and determines its composition. It servers as interface between system core and coobjects. Usually, a "capsule" just coordinates and redirects invocations to the appropriate components. A common "capsule" is implemented and aggregates one instance of each component.

Previous research has concluded that the definition of two states for an update, committed and tentative, is very useful in mobile environments [7,15]. For instance, in a scheduler application, reservations executed by users must be considered tentative until they are committed. Users should be allowed to see tentative data to avoid possible conflicts (tentative data represent a foresight of the coobject's state). In the DAgora system, a programmer may easily create a coobject that stores a tentative and a committed version of the data, relying on simple data objects and using the extended "capsule" implementation. This "capsule" is composed by two instances of the components "log-ordering" and "data" and transparently maintains both states – committed data results from the execution of stored updates using a pessimistic total order, while tentative data results from the execution of unstable updates to the committed state using causal order.

Data

The component "data" implements the real data type being created, with its associated state and operations. With current log implementations, which are based simply on update ordering, the code of each operation is responsible for detecting and solving conflicts among concurrent updates.

For some applications, it is impossible to solve conflicts automatically. For instance, if a base element (e.g. section) of a structured document is modified concurrently by two users, the system usually can neither decide which modification is the best, nor merge both modifications. In such cases, two versions of the conflicting element must be created and resolution must be left to users. In DAgora, we have created a component "data" that implements a set of generic objects with multiple versions. Concurrent modifications of the same object are detected and solved automatically creating multiple versions. Programmers may extend this component and define automatic merging procedures or leave this work to users. Another component implements a generic tree-structured organization on top of the above set of objects and can also be extended by programmers. These base components have been used for implementing several structured documents manipulated by a collaborative editor [13].

When the *multi-version* "data" components are not used, the programmer must take into consideration the DAgora model of operation, when implementing the component "data". However, to guarantee that users' intentions are respected when updates are applied in each server and eventual conflicts are detected and solved, some simple techniques must be followed and DAgora provides the necessary support for their implementation. First, the existence of concurrent updates may be tested using the timevector associated with each operation. Second, the defined preconditions for the execution of an update may be checked. Third, the definition of alternative actions to be executed dependent on the coobject's state is possible. Fourth, the definition of state-independent operations is also possible. More complex techniques, such as updates' transformations [3], may also be implemented using the updates stored in the "log" component.

Our experience with some implemented applications [13] suggests that most applications will use one or two simple techniques (e.g., a careful operation definition associated with a regular precondition checking has been used in a scheduler application, while our *multi-version* components rely uniquely on the information associated with each update).

Using The Object Framework–Scheduler example

The scheduler application enables users to reserve resources, such as meeting rooms, projectors, etc. Users interact with a graphical interface, presented in figure 3, 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. To enter a new reservation users must indicate the set of alternative times for which they intend to reserve the resource and give a brief description of the reason.

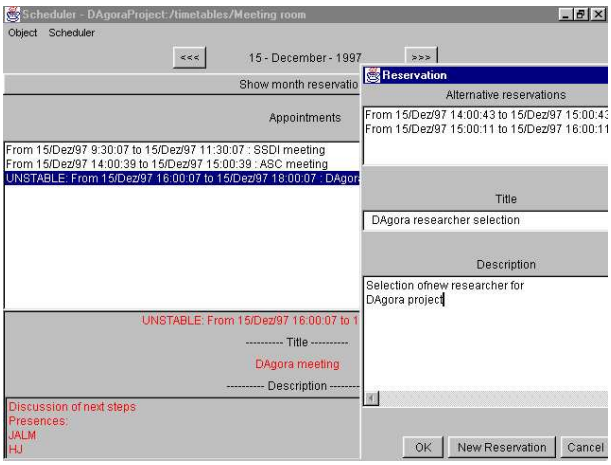

Figure 3 – Scheduler application. Main window presents known reservations. Reservation window is used to set a new appointment.

To implement this application we have developed a coobject based on an extended capsule – thus, transparently providing a tentative and a committed version of each calendar. To guarantee that appointments are committed as soon as possible in a consistent way across different replicas, even in the presence of disconnected replicas, updates are applied using a causal total order algorithm based on a primary replica.

In figure 4, we present the code needed to implement the scheduler coobject, which is preprocessed to generate standard Java code. As it can be seen, to create a new coobject type, a programmer only has to define the component "data" and to select the desired component implementations. This simplifies data-type construction, through massive code reuse.

```
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 4 – Scheduler coobject implementation. *SchedulerCapsule* defines the components used in the coobject, and extends the selected capsule. *SchedulerData* implements a simple scheduler object, as it would usually be implemented (besides defining which invocations should be logged).

RELATED WORK

Several systems have been developed to manage data in large-scale environments including mobile computers. Notably, some *mobile* database systems [2,5] are based on "mobile" transactions. However, as these systems have been implemented for different purposes they usually define a model of concurrency control that is too restrictive for collaborative applications (discarding executed contributions is usually unacceptable).

Coda [10] 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 update conflicts. Ficus distributed file system [12], although presenting a different architecture, relies on similar conflict detection and resolution mechanisms. As we have already referred, although these systems work very well for its intended environment, experimental results suggest that the resolution of conflicts based on simple state propagation may be very difficult for complex objects. Odyssey, Coda's successor, presents a model for application-aware adaptation in presence of mobility based on collaboration between system and applications. It is particularly

interesting to support multimedia applications, where data fidelity may be selected according to available connectivity.

Bayou [15] 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 perform update commitment. The combination of these features reveals itself quite adequate for large-scale mobile system. The DAgora system may present Bayou's main characteristics through adequate coobject definition. However, DAgora enables different concurrency control schemes and specific data types definition – it does not require data to fit the available relational model. Therefore, in some circumstances, DAgora enables the implementation of more flexible and suitable solutions.

Rover [7] 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 servers by calling type-specific methods. RDOs are also used to export computations to servers. QRPCs 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 eases the creation of new data types.

Several distributed object systems have been previously developed and present some form of concurrent update handling. Some of them [9] even provide object frameworks decomposing object operation. However, these systems are usually real-time, designed for low granularity objects with different requirements, and present solutions unsuitable for mobile large-scale settings.

Sync [11], a framework for mobile collaborative applications, presents an interesting model to handle concurrent updates and to create new object. However, we believe that lack of server replication makes it less suitable for large-scale asynchronous settings.

FINAL REMARKS

The DAgora data storage presents an architecture that allows adaptation to specific environments using a range of system configurations. We believe that it provides a suitable solution for data availability in setting with mobile computers.

The associated DAgora open object framework allows programmers to develop specific solutions for their problems. As there is no single solution that solves all problems, we believe that the flexibility that is provided by this open object framework is fundamental to support different types of applications. Moreover, the object framework simplifies the task of programmers allowing them to reuse several predefined components that handle most of the complexity associated with data distribution.

REFERENCES

[1] Coan B., Oki B., Kolodner E. Limitations on database availability when networks partition. In *Proceedings 5th ACM Symposium on Principles of Distributed Computing*, August 1986.

[2] M. Dunham, A. Helal, S. Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2, 1997.

[3] Ellis C., Gibbs S. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, June 1989.

[4] R. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4), 1992.

[5] R. Gruber, F. Kaashoek, B. Liskov, L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, December, 1994.

[6] Imielinski T., Korth H. Introduction to Mobile Computing. *Mobile Computing*, ed. T. Imielinski and H. Korth, Kluwer Academic Publisher, 1996.

[7] Joseph A., DeLespinasse A., Tauber J., Gifford D., Kaashoek M. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[8] Karsenty A., Beaudouin-Lafon M. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[9] A. Kermarrec, I. Kuz, M. Steen, A. Tanenbaum. A Framework for Consistent, Replicated Web Objects. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, May 1998.

[10] Mummert L., Ebling M., Satyanarayanan M. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

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

[12] Page Jr. T., Guy R., Heidemann J., Ratner D., Reiher P., Goel A., Kuenning G., Popek G. Perspectives on Optimistically Replicated, Peer-to-Peer Filing. *Software-Practice and Experience*, vol. 28(2), February 1998.

[13] Preguiça N., Martins J., Domingos H., Simão J. System Support for Large-Scale Collaborative Applications. Technical Report, TR-01-98 DI-FCT-UNL, Dep. Computer Science, New University of Lisbon,1998.

[14] Satyanarayanan M. Fundamental Challenges in Mobile Computing. In *Proceedings of the 15th ACM Symposia on Principles of Distributed Computing*, 1996.

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