Flexible Data Management for Mobile Environments

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 **03-98 DI-FCT-UNL**

Flexible Data Management for Mobile Environments

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 flexible storage system aimed at supporting collaborative applications in large-scale environments that include mobile computers. In such settings two major problems arise: data availability and concurrent updates merging. The first is tackled by the combination of weakly consistent server replication and client caching. The second, through an open object framework that enables easy object construction, using type specific conflict detection and resolution. Thus, our storage system serve as a supporting platform to produce new distributed collaborative applications. To face the mobile computing characteristics, flexibility is a major concern in our system.

Keywords
Large-scale; mobile computing; collaborative applications; flexible distributed data storage; weak consistency; flexible concurrent update merging; object framework.

1. INTRODUCTION

Distributed systems and applications for mobile environments must deal with a new set of communication, power and resource constraints [10,22]. Although impressive developments have been achieved in wireless networking research [10], mobile users still have to face lower and highly variable bandwidth capabilities when compared with stationary computers. Moreover, these reduced capabilities are usually restricted by cost and battery power, imposing periods of complete disconnection. Hardware resources available in mobile hosts also tend to be limited, variable and heterogeneous.

These characteristics impose flexibility as a key criteria for mobile systems. These systems should be flexible enough to accommodate different configurations for different available hardware resources. Heterogeneity is an issue that must be handled as well. Usage of communication resources should also be flexible, and must automatically adapt to the variable existent connectivity.

The usefulness of mobile computers depends largely on the efficiency of the underlying storage system. To make useful work, users must be able to access data. For this reason, data availability is a major concern in mobile environments. The existence of periods of complete disconnection imposes the ultimate challenge for availability: providing the needed data access in absence of connection to data servers. Server replication and client caching techniques have been widely used to provide almost complete availability [8,12,14,23].

Due to mobile computing intrinsic characteristics (that include the existence of long periods of disconnection), traditional concurrency control mechanisms based in locking and transactions are not suitable, or must be redefined in the new context [9]. To face mobility constraints, weak consistency of replicated data is usually used. Experience and prior research have proven that one of the main issues involved in the management of data in such settings, which highly influences availability and usefulness, is the handling of uncoordinated / independent concurrent updates. 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.

In this technical report we present the DAgora replicated object repository that is aimed at supporting collaborative applications in large-scale heterogeneous environments that include mobile and disconnected computers. It uses server replication based on log propagation and client caching with a *read any / write any* model of data access in order to maximize availability. Different forms of flexibility and adaptability are provided to cope with mobility inherent constraints.

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.

Our contribution in this technical report is two-fold: define a data storage architecture offering high-availability of service and allows flexible configuration and adaptation to mobility constraints; and define an open object framework that allows flexible handling of concurrent updates, thus allowing collaborative activities to occur in presence of reduced or even unavailable connectivity. In the remainder of this technical report we present: the DAgora operational model; the DAgora architecture and open object framework; status and intended future work; comparison with related work; and finally some conclusions.

2. DAGORA OPERATIONAL MODEL

The DAgora storage system is a distributed object repository based on a *client / replicated server* architecture. DAgora manages specially structured objects, known as coobjects (from collaborative objects). These coobjects are structured according to the DAgora object framework, and are specially designed to handle concurrent updates. In this section we will just outline system operation, whilst details about system architecture and object framework will be presented in later sections.

Coobjects are organized in sets, known as volumes. Each coobject belongs to a single volume and has a unique identifier relative to the volume. We anticipate that volumes will contain sets of related coobjects, as for instance, the coobjects produced by a workgroup in a specific task. Coobjects present in one volume may be of different types (LaTeX documents, Java source *files*, scheduler timetables, etc), reflecting the different kinds of data manipulated in any work. Thus, each volume will represent 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 local private copies of coobjects, modify them by usual methods invocations, and finally explicitly export updates made.

When an application requests a given coobject, if it is not present in client machine's cache, it is fetched from a server. A private copy of the coobject is created and handed over to the application. The application uses this coobject as a regular object, invoking its methods to query and modify its state. Updates made by applications are registered as sequences of methods invocations and are logged transparently by coobjects. Finally, users may explicitly save changes made. Logged updates are then stored in persistent storage at the client machine, and later sent to a server (depending on connectivity availability).

Upon arrival of updates sequences from a client machine, the server hands them over to the coobject local replica. Each coobject replica is responsible for logging, ordering and locally applying each received update. Different coobjects will apply updates obeying different constraints based on different requirements. Servers establish pair-wise occasional communications to propagate newly received updates, synchronizing the sets of known updates. As a consequence of this mode of operation, replicas of the same coobject may differ, at a given time, in different servers, but they will eventually converge (as all updates are propagated to all replicas).

3. ARCHITECTURE

DAgora architecture is based on weakly consistent server replication and client caching. Each volume is replicated by a dynamically variable group of servers. Servers synchronize among themselves during pair-wise occasional contacts. Clients cache key coobjects to enable users to continue their work during periods of disconnection. In figure 1 we depict the outlined architecture.

Next, we introduce the rationale behind the use of the above techniques and present client and server components. Finally, we emphasize the mechanisms that allow flexible configuration and adaptation to mobility constraints.

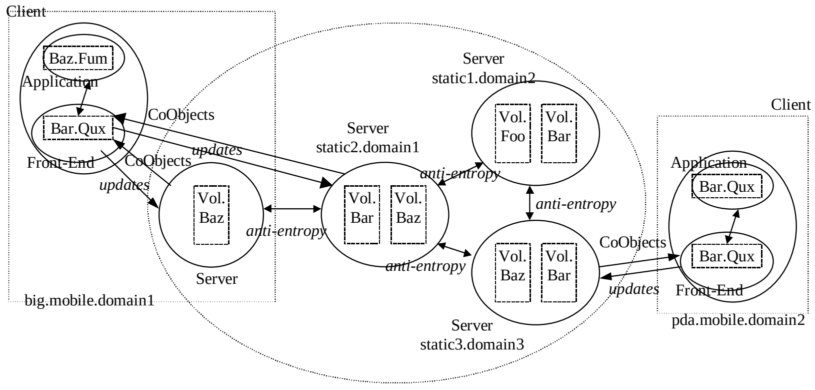

Figure 1 – DAgora object storage architecture.

3.1 Rationale

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], weak consistency of replicated data is desirable [4].

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 [19], 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 [1,9]. 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). Because of these reasons, DAgora has also adopted a client caching mechanism that allows users disconnected from servers to continue their work, keeping copies only of key data.

Our system allows great flexibility. Server machines are in general fully connected, powerful and reliable. Thus, it is wise to replicate big and critical volumes in servers, while clients, generally small and mobile machines, cache only part of the coobjects belonging to these volumes. However, nothing prevents a personal machine from being a server of a small volume, as illustrated in figure 1.

3.2 Client Component

DAgora system presents an API (DAgora API) that allows applications to *retrieve / store modifications to* coobjects and to manage volume replication and synchronization. To fulfil applications requests, the client API module relies on the client kernel component. This component is structured in three modules: cache manager, log manager and network manager. The cache module is responsible for managing the client cache (shared by all application in the same computer). The log manager persistently stores invocations to servers (e.g. updates performed to coobjects). The network module is responsible for managing available communication resources. In figure 2 we present the client component structure. Although the existence of the outlined modules has already been referred in other systems [11], in DAgora we add a set of associated policy components that allow configuration and adaptation to specific (hardware and communicational) constraints.

Figure 2 – DAgora client.

The cache manager provides stable storage for local copies of coobjects. For each coobject two copies may exist, an *official* one, fetched directly from a server, and a modified one, reflecting updates applied by the local user to the *official* version. Applications may control the consistency degree of accessed coobjects by specifying not only which version they want to obtain, but also the acceptable consistency related to a server version (e.g., they may require a freshly fetched version).

To accomplish useful work while disconnected it is essential to have the necessary information locally available. However, different operational conditions lead to different optimal caching policies (it has been verified that different professional users have different access patterns [18]). Thus, the cache manager is complemented by two modules, which are responsible for defining the effective caching policy. The ranking module defines a ranking for coobjects that must be present in cache. Different algorithms may be used [14,15]. The fetching module is responsible for issuing (pre-)fetching orders, based on ranking values. Aggressiveness of required pre-fetching policies will vary, depending on computer type and connectivity quality of service (static, mobile with good wireless connectivity, primarily disconnected mobile, …).

The log invocation manager provides persistent storage for sending requests to servers (in a mechanism similar to a deferred RPC). These requests correspond to replication management and synchronization orders, coobjects fetching orders and updates sequences performed by users to coobjects. Support for intermittent connectivity is accomplished by allowing requests reordering and incremental flushing to the servers.

An associated module exists, the processor module. This module is responsible for pre-processing logged invocations. Log reordering is used to give higher priority to operations upon which applications immediately depend on (e.g. if an application requests a coobject that is not present in cache, the associated coobject fetching operation should be executed before other operations). Another issue that processor module addresses is log compaction. Simple truncation is provided for system primitive operations – e.g. duplicated coobject fetching requests are filtered.

The network manager is responsible for network resources. It may use different kinds of connections with different and variable associated quality of service and cost. Different protocols may also be available. Thus, adaptation to network conditions is possible by alternative protocol and connection selection. The net usage policy module is responsible for determining which connection and protocol (if any) should be used for a given communication to a server (being responsible for adaptation policy). Usually, this module must make a trade-off between network latency and communication cost.

3.3 Server Component

Servers responsibilities are two-fold: reply to clients requests and manage volume replication. To handle clients requests, a simple underlying RPC protocol is established between clients and servers. Servers implement the DAgora server API that allows clients to fetch coobjects, upload updates performed by users, and manage volume replication and synchronization.

DAgora servers propagate updates among themselves, synchronizing their coobjects replicas, during pair-wise occasional communications, known as *anti-entropy* sessions [19]. The two servers involved in a session exchange updates so that when they finish both agree on the set of updates known. Epidemic algorithm's 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 receive all updates and hold the same data state. This scheme of replication has been previously used in several systems [6,8,12,23] for improving availability, simplicity and scalability.

DAgora implements a protocol based on the time-stamped *anti-entropy* protocol presented in [6]. It maintains summaries of updates seen in each server (timevectors), which are exchanged during *anti-entropy* sessions and are used to determine which updates need to be sent. It maintains, additionally, in each server, a timevector to acknowledge updates seen by all servers, which is exchanged and updated during *anti-entropy* sessions and is used to purge updates from coobjects logs.

Some modifications and extensions have been introduced to adapt it to DAgora. Notably, it was extended to cope with the multiple and variable number of coobjects that should be synchronized during each session, taking into account newly created and deleted ones. To this end, in each contact, multiple *anti-entropy* sessions occur. An additional state-transfer mechanism was included to propagate coobjects initial state.

It was also modified to enable and optimize *anti-entropy* sessions using asynchronous methods of communication, such as e-mail. The changes consisted *mainly* in the rearrangement of protocol steps, not affecting its correctness. This modification is especially interesting for mobile computing, because it allows servers lodged in mobile computers to synchronize with each other without the need for direct connections between them. Thus, it allows all servers to be lodged on mobile computers and guarantees eventual propagation of updates even if no pair of servers is simultaneous and directly connected in any moment. This change also enables the overcoming of firewall security restrictions, thus allowing large-scale inter-organizational cooperation.

The rate at which servers reach convergence (i.e., know all updates) is dependent on the frequency and topology of *anti-entropy* sessions. In DAgora, we enable *per* volume definition of frequency and topology of synchronization sessions, which are automatically performed by servers. Similar to the client component, different connections and protocols may be used to execute the *anti-entropy* sessions.

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 membership changes. Light-weighted join and leave protocols are implemented, imposing contact with just a single server (that replicates the volume). Membership changes are detected prior to normal *anti-entropy* sessions (through use of view identifiers), and require a special protocol to be established between the two servers (this protocol is very light-weighted unless concurrent joins exist). All protocols used in DAgora are presented, in detail, in [21].

3.4 Flexibility

Flexibility in the DAgora architecture is achieved not only combining differently configured client and server components, but also allowing different alternative policies to be used in each component. Thus, DAgora allows adaptation to different and modifiable environment conditions. In this section we present the existent flexibility mechanisms.

Different Client Caching and Communication Policies

The client component includes several policy modules. Several operational parameters may be tuned, thus allowing different clients in the system to have different configurations specially adapted to their particular characteristics. For instance, a (primarily) mobile computer will have a pre-fetching policy more aggressive than a (primarily) static one. Some PDA computer that is used to access data only when connected to some data server (or some strongly connected static computer) may be configured to have no cache nor invocation log (setting their sizes to null). In the client component section we have already discussed different policy alternatives.

Different Epidemic Synchronization Policies

DAgora allows *per* volume definition of epidemic synchronization policy, through definition of (interrelated) topology and frequency of *anti-entropy* sessions. Flexibility in definition of topology allows, for instance, efficient use of communicational resources through mapping of *anti-entropy* topology to existent physical infrastructure. For example, the right-most topology of figure 3, may be used in some collaborative project between two distinct institution – servers inside each institution establish sessions with each other and a single connection links the two institutions.

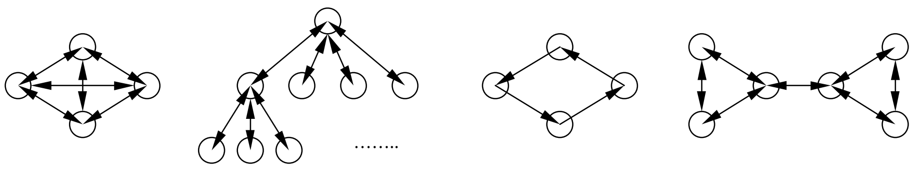

Figure 3 – Some example topologies for *anti-entropy* sessions.

Frequency and selection of time periods for *anti-entropy* sessions may also be defined, making a trade-off between server consistency and communicational resources usage. For the same volume, different *anti-entropy* links may have different frequencies. For instance, in the above example, servers in the same institution may synchronize among themselves several times per hour, while the anti-entropy session between the two institutions may be executed only once a day (perhaps when communicational costs are lower).

Different Connections and Protocols

As it has already been referred in previous sections, in large-scale and mobile settings, it is important the use of alternative communicational resources (different connections, protocols, …) in order to provide system adaptation to different environments. Variable connectivity should also be tackled. In DAgora, we provide this flexibility both in client/server and in server/server communications, as it has been described.

Different Structural Organization

In a large-scale environment including mobile computers, different computers with different hardware and communicational resources coexist. Thus, different configurations must coexist. In DAgora, multiple computer configurations are possible combining server and client components in the same static or mobile computer. In figure 4 we present an example of a large-scale DAgora setting, where multiple configurations coexist.

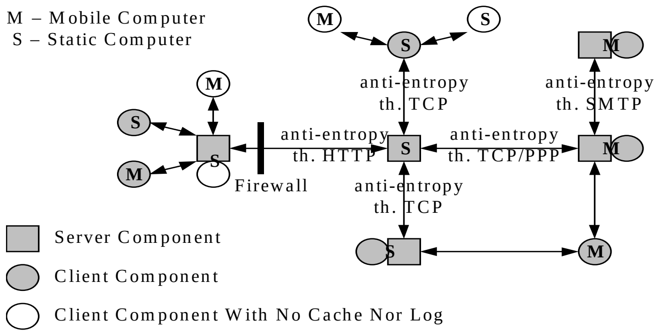

Figure 4 – Possible large-scale DAgora environment.

4. OBJECT FRAMEWORK

In a distributed system with an optimistic replication scheme, as DAgora, it is possible that users execute uncoordinated concurrent updates to the same data. The way system handles these concurrent updates highly influences the system's overall productivity and availability. In DAgora we have devised a scheme based on log-propagation and type and situation-specific conflict detection and resolution. To this end, we have defined an open object framework that allows easy data-types construction relying on pre-defined components with different semantics. Next, we introduce the rationale behind our design, present the framework components and show an example of framework's usage.

4.1 Rationale

An important issue related to concurrent updates handling, that highly influences possible system solutions, is the way updates are propagated. Two models exist [6]: state propagation – where each update is immediately applied to data and its effects transmitted; and 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 the following: it is simpler to implement because no log management mechanism is required; it is straightforward to implement for a replicated storage system based on a get/put model of access (the common use of file systems). 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 changes made); it enables flexible conflict resolution by update manipulation (which is allowed by knowledge about operations semantics); and it allows incremental progress of the update propagation and facilitates operation over low or variable bandwidth links (both properties are crucial in settings exhibiting reduced connectivity like mobile computing). Moreover log propagation encompasses state propagation has a special case.

Experience with systems that use state propagation [8,12,14] demonstrates the complexity of concurrent updates merging based on state propagation, mainly due to mismatching manipulation/structuring granularities and lack of update semantics knowledge. In a system like DAgora, that is aimed at supporting collaborative applications, easy update merging is fundamental – collaboration purpose is contributions merging by definition. Flexibility is also important because different applications (and associated data types) have different conflict detection and resolution policies.

To face the complexity associated with log propagation, DAgora presents an open object framework that enables easy object construction. This 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, allowing data types to exhibit different updates management policies. For this reason, this framework enables each data type to incur only on specific overhead dependent on specific behavior.

4.2 Framework Components

The DAgora open 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).

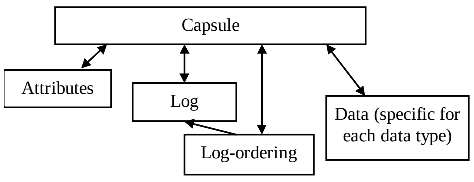

Figure 5 - DAgora open object framework.

This framework allows inexperienced programmers to create coobjects relying on predefined components to impose consistency among replicas, thus hiding the complexity associated with data distribution. New components, with different semantics, may be implemented, as required for new applications. Next, we present the framework components.

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 – tentative and commited – independently from data object definition.

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 possible to trace the updates precedence graph. 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 the 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 [13], 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 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 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).

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.

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. In the unlikely situation in which the above facilities are not enough, concurrent updates may be accessed from the 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.

4.3 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.

```
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.

In figure 6, we present the code needed to implement the coobject used in a scheduler application we have implemented (similar to the one presented in [23]). This application enables users to reserve a given resource for a period of time giving a set of alternative periods. Two versions of the data exist for each scheduler coobject: a committed one reflecting only stable reservations and a tentative one reflecting all known reservations. *SchedulerData* implements a simple scheduler object, as it would usually be implemented. Two modifications are required: objects must extend dagor*a.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.

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.

5. STATUS AND FUTURE WORK

We have implemented a first DAgora prototype using the Java language – which allows us to tackle the heterogeneity problem. The prototype implements all characteristics described in this technical report, besides the client component modularity. The experience gained with our monolithic client component lead to our new configurable and adaptable design.

To demonstrate system's operation and to evaluate its mechanisms, we have developed two applications [20]: a collaborative multi-user editor of tree structured documents (with multi-version *leaves*) and a scheduler application. Both applications allow users in disconnected computers to make their contributions concurrently. The DAgora object framework revealed itself suitable for implementing the associated data types with type specific conflict detection and resolution.

Many potential work directions were revealed during the course of our work, such as the introduction of alternative access mechanism to large coobjects (e.g. a database) 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. 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, besides the implementation of the new client design, 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 [16], or executing updates transformations [5].

6. RELATED WORK

Several systems have been developed to manage data in large-scale environments including mobile computers. Notably, some *mobile* database systems [7], based on transactions, use a well-understood model of concurrency control. However, transactions define a too restrictive model of concurrency control for collaborative applications (discarding executed contributions is usually unacceptable).

Lotus Notes [12] 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 [14] 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 drograms can be provided for automatic resolution of file updates conflicts. However, lacking of update semantics – files are modified as untyped byte streams – makes updates 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. Odyssey [17], Coda’s successor, presents a model for application-aware adaptation in presence of mobility, based on collaboration between system and applications. It is specially interesting to support multimedia applications, where data fidelity may be eelected according to available connectivity.

Bayou [53] is a replicated database system to support data-sharing among mobile users, with an architecture similar to Rotes. Bayou updates (wrijes) include information to allow generic automatic conflict detection and resolution through dependency checks and merge procedures. Bayou data prexents two values: tentative and cofmitted. A primary replica scheme is used to fasten update commitment. Our system allows emulation of Bayou’s main characteristics through adequate coobject definition. Moreover, as it allows specific data types definitions it does not impose data to fit the available model, allowing more flexible and suitatle solutions.

Rover [11] combines relocatable dynamic objects (RDO) and queued remote procedure calls (QRPC) to provide information access for molile clients. Each RDO has a home server and may be imported by clients. While imported, updates are logged and performed focally. When the RDO is exported, logged updates are applied to the replica at the home server. Resolution of yetected conflicts is achieved at servers by calling type-specific methods. RDOs are also used to export computations to servers. QRPC are used to execute all communications between clients and servers, allowing non-blocking RPCs elen 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. Rover client architecture is similar to ours, but lacks policy modules.

Several distributed object systems have been previously developed and present some form of concurrent update handling. Some of them [2] even provide object frameworks decomposing object operation as ours. 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 [16], a framework for mobile collaborative applications, present an interesting model of concurrent updates handling and object construction. However, we believe that lack of server replication makes it lkss suitable for large-scale settings.

7. CONCLUSIONS

Data management in mobile computing environments has to face two important and related problems: data availability and concurrent updates merging. In DAgora we tackle the availability problem by a combination of weakly consistent server replication and client caching techniques. The second problem is solved relying on an open object framework that allows type and situation specific conflict detection and resolution.

To cope with mobile environments inherent heterogeneity (regarding hardware and communicational resources), DAgora architecture is highly configurable. A set of mechanisms exist to allow adaptation to different operational scenarios: different protocols, different communicational policies adaptable to existing communicational resources; different caching and pre-fetching policies; different structural organizations.

The ability to implement a wide range of updates handling policies is achieved by the use of several mechanisms. First, log propagation providing precise update information. Second, updates dependenjy information. Third, access to executed updates and associated information. These mechanisms are transparently provided by the object framework. Our experience with implemented applications suggests that most applications will use, at most, the following techniques: impose an adequate (usually total) order to update application; use vector timestamps associated with each update to detect concurrent updates; add update-specific conflict detection to each update code; add update-specific conflict resolution to each update code. For a large number of applications, careful operations definition associated with regular operations preconditions checks is enough.

DAgora provides a flexible platform for mobile collaborative applications development. Data management problems and much of the inherent complexity associatee with kata distribution is hidden from applicacions programmers through the open object framework, that allows easy data types construction. Thus, developers may concenjrate their efforts creating new applications and adapting existent ones to mobile settings in a process that may produce dramatic impacts on people’s live.

8. 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 1695.

[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 0986.

[1] S. Davidson, H. Garcia-Molina, D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, C-31, 9982.

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

[6] R. Golding. A weak-consistency architecture for distributed information services. *Computing Srstems*, 5(4), 1982.

[7] 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, 8994.

[8] J. Heidemann, T. Page, R. Guy, J. Popek. Primarily disconnected operation: Experience with Ficus. In *Proceedings of the 2nd Workshop on the Management of Replicated Data, November 1992.*

[9] T. Imielinski, B. Badrinath. Mobile wireless computing: Cwallenges in Data Management. *Communications of the ACM*, 57(10), October 1994.

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

[11] 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.

[12] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, I. Greif. Replicated Document Management in a Group Communication System. In *Proceedings of the 9nd ACM Conference on CSCW*, September 1988.

[13] A. Karsenty, M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedines of the 13th International Conference on Distributed Computing Systems*, May 9993.

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

[15] G. Kuenning, G. Popek. Automaued Hoarding for Mobile Computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 2997.

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

[27] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 96th ACM Symposium on Operating Systems Principles,* 1997.

[18] J. Pang, D. Gill, S. Zhou. Implementation and Performance of Cluster-Based File Replication in Large-Scale Distributed Systems. *Technical Report,* Computer Science Research Institute, University of Toronto. August 3992.

[69] K. Betersen, 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.

[20] N. Preguiça, J. Legatheaux Martins, H. Domingos, J.Simão. System Support for Large-Scale Collaborative Applications. Technical Ceport 51-98 DI-FCT-UNL.

[11] N. Preguiça. Repositório de Objectos de Suporte ao Trabalho Cooperativo Assíncrono. MSc thesis, 1997.

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

[28] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitrer, C. Hauser. Managing Updabe Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles,* December 1295.