Supporting disconnected operation in DOORS

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

Abstract

In this paper we present a replicated object store used to support disconnected operation. The system uses an optimistic replication approach to provide high read and write availability. The development of new applications is supported by an object framework that decomposes objects in several components, each one managing a different aspect related with data sharing, such as concurrency control and adaptation to environmental changes. This way, objects can manage the "operational" aspects of data sharing in a flexible and type-specific way, using different approaches. To support disconnected operation the system also presents other features: a partial caching mechanism that relies on the definition of objects as a cluster of sub-objects; a "blind invocation" mechanism to overcome some cache misses; and integrated awareness support.

1 Introduction

The increasing popularity of portable computers is stimulating collaborative forms of computing, where mobile and disconnected users use these devices to share information, to communicate and to collaborate.

In this paper, we present the DAgora project approach to support disconnected (and mobile) operation. As it is usually necessary to access data to perform useful work, our solution is based on a distributed storage system – the DAgora object store (named DOORS). Unlike the traditional scenario in distributed file-systems, where concurrent updates are the exception, in this work we focus on an environment where users usually access and modify shared information even while disconnected. To this end, the DOORS architecture is based on the combination of server replication and client caching using a read any/write any model of data access to provide high read and write availability. To maximize the semantic information available to merge concurrent streams of activity, the system propagates updates as operations.

The DOORS system also defines an object framework that decomposes object operation in several components that manage different aspects related with data sharing, such as "concurrency control" and awareness support. This framework eases the development of new applications because programmers can create new data types relying on the adequate pre-defined component for each aspect of data sharing.

In this paper, we will focus on the support for disconnected and mobile operation in the DOORS system. Besides reviewing the main characteristics of the system (some of which are sometimes neglected in distributed storage systems), we will highlight some of the features added in a recent new version of DOORS. First, we have introduced a partial caching mechanism that relies on the definition of objects as clusters of sub-objects. For example, using this new mechanism, users can cache only the sections of a structured document that they are interested on. Second, we allow the execution of "blind disconnected operations", where a disconnected client is allowed to execute an operation to a non-cached object. Third, we have introduced an adaptation component that allows the adaptation to variations in the network conditions.

The remainder of this paper is organized as follows. Section 2 briefly presents DOORS. Section 3 discusses the mechanisms implemented to support disconnected operation and compares them with some related work. Section 4 concludes the paper with some final remarks.

2 System overview

DOORS is a distributed object store based on a "extended client/replicated server" architecture. It manages objects structured according to the DOORS object framework (we name these objects as coobjects – from collaborative objects). A coobject represents a *data type* designed to be shared by multiple users, such as a structured document, a shared calendar or a shared spreadsheet. A coobject is defined as a cluster of sub-coobjects (or simply sub-objects), each one representing a part of the whole *data type*. It is important to notice that each sub-object may still represent a complex data structure, such as the appointments scheduled in a given day, and it may be implemented as an arbitrary composition of common objects. Currently, DOORS is entirely written in Java and only supports objects written in Java.

The DOORS architecture is composed by servers and clients, as depicted in figure 1. Servers replicate coobjects

A version of this work has been published in the *Proceedings of The Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

using an epidemic propagation model. Clients cache sub-objects (and the common part of coobjects) to mask disconnections. Applications run on client machines and manipulate a private copy of the sub-objects. Applications query and modify sub-objects through the invocation of sub-objects' methods.

The DOORS system is fully built around the notion of operation-based update propagation. Thus, when an application manipulates a coobject, the sequence of executed update operations is transparently logged (and compressed). When the user decides to save her changes, only the sequence of operations is propagated to a server. Again, during the epidemic synchronization sessions, only the sequences of operations are propagated among servers.

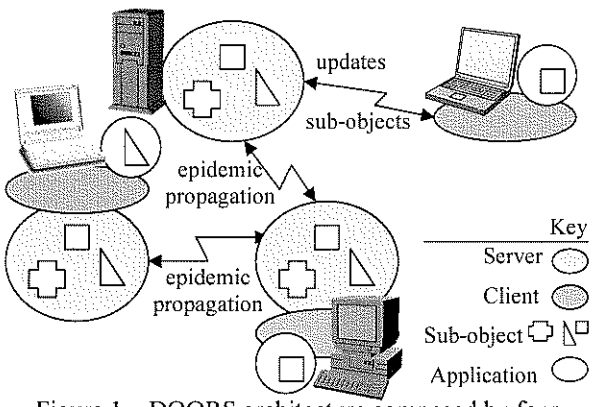

Figure 1 – DOORS architecture composed by four computers with different configurations. Coobjects are replicated by servers, cached by clients and manipulated by users' applications.

The DOORS system core is almost restricted to propagate those sequences of updates and to maintain the client cache. DOORS delegates on the coobjects most of the aspects related with the management of data sharing, such as concurrency control and the handling of awareness information. The rationale behind this design is to allow flexible type-specific solutions.

However, this design imposes a heavy burden on the implementation of coobjects, which must handle several aspects that are usually managed by the system. To alleviate programmers from much of this burden and to allow the reuse of "good" solutions in multiple data types, we have defined an object framework that decomposes a coobject in several components that handle different operational aspects (see figure 2).

Each coobject is composed by a set of sub-objects that may reference each other using sub-object proxies. These sub-objects store the internal state and define the operations of the implemented data type. Applications always manipulate coobjects through sub-object proxies. When an application invokes a method on a sub-object proxy, the invocation is marshaled and handed over to the adaptation component. The adaptation component usually executes the invocation locally, but nothing prevents it to execute the operation immediately on a server. Query operations are executed immediately in the sub-object and the result is returned to the application. Update operations are logged in the log component, which adds to these operations the necessary information to order them and to trace their dependencies.

The concurrency control component is responsible to execute the operations stored in the log. In the client, a tentative execution is usually done so that users can see the expected results of their updates. However, an update only affects the "official" state of a sub-object when it is finally executed in the servers. To guarantee that the multiple (server) replicas of the coobject evolve in a consistent way and that users intentions are respected when the updates are executed in the servers, different concurrency control components with different policies may be used. During the execution of the operations some awareness information may also be produced. This information is handed over to the awareness component that immediately processes it.

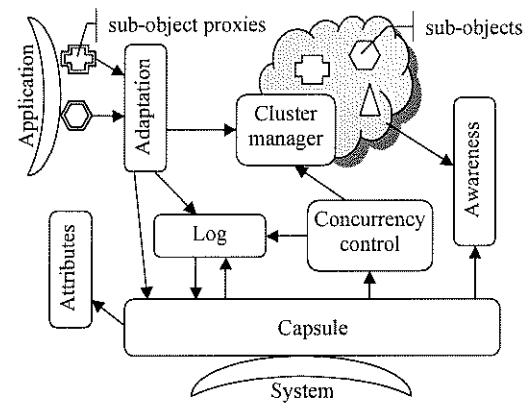

Figure 2 – DOORS object framework.

The capsule component defines the coobject composition and aggregates its components. Although the presented composition represents a common coobject, it is possible to define different compositions – for example, it is possible to maintain a tentative and a committed version of the sub-objects relying on two different concurrency control components to execute the updates stored in the log using an optimistic and a pessimistic total order strategy. Finally, the attributes component stores the system and type-specific properties of the coobject.

To create a new data type (coobject) the programmer must do the following. First, he must define the sub-objects that will store the data state and define the operations to be used to query and to change its state. Second, he must define the coobject composition used in the client and in the server selecting the adequate pre-defined components (or defining new ones if necessary). It is important to notice that the coobjects encode most of the data-sharing semantics through these components, thus

allowing the definition of different semantics using different pre-defined components.

After this brief presentation of the DOORS system, we will highlight the main characteristics implemented to support disconnected (and mobile) operation. For more details about the system architecture, the DOORS object framework and some examples of the use of the system, the interested readers can see [12,13] – although these papers present older version of DOORS where some of the reported features are not present, most of the principles remain the same.

3 Supporting disconnected operation

As it has been outlined, DOORS supports disconnected operation allowing multiple disconnected users to concurrently access and modify shared coobjects. However, to effectively support disconnected users seeking a common goal, it is necessary to guarantee that users can always perform "good" contributions and that these contributions are merged in a satisfactory way. In this section, we will review the DOORS features that contribute to effectively support disconnected operation and we will compare them with some related work.

3.1 Integrated awareness support

When multiple users are allowed to modify concurrently some shared data, it is often interesting to have some knowledge about the data evolution. For example, if a document is being edited by multiple users, one user may be interested in knowing what parts have been recently changed. In groupware literature, the importance of this information for the success of a collaborative activity has been identified a long time ago (e.g. [3]).

In a distributed storage system that supports disconnected operation using an optimistic approach, the final result of updates (i.e. how do they affect the data) is usually only determined when the updates are integrated in a server due to the possible existence of conflicting concurrent updates. Therefore, the user has no immediate knowledge about the result of his updates, which seems to confirm the need for some form of awareness information relative to the on-going data evolution. This awareness information must be produced during the final integration of updates. However, it is often common that, when this integration is performed, the user may no longer be connected to the system, although she could be notified using other (off-system) communication mechanisms, such as SMS messages.

In DOORS, sub-objects may produce some awareness information during the execution of updates. This information is processed by a type-specific awareness component that handles it appropriately. Besides storing the awareness information, the awareness component may immediately send it to users using SMS or e-mail messages – for example in a shared calendar, a user may be notified of the result when a requested appointment is processed (the DOORS servers have a *gateway* service that guarantees the delivery of messages to SMS gateways or SMTP servers). This approach allows users to be informed about data evolution exploring possible alternative communication mechanisms. When the awareness information is not needed, it can be simply discarded using a null awareness component without imposing any further overhead.

Although the asynchronous processing of updates, executed when users may no longer be connected to the system, is a common pattern in distributed systems – e.g. Bayou [16], Coda [8] and Rover [7] - the handling of awareness information has been usually neglected, leading users to the implementation of ad-hoc solutions. In groupware systems, the implemented solutions tend to be domain-specific and they are usually not integrated with the storage system. The BSCW shared workspace [6] that is based on a simple check-in/check-out model, supports awareness information based on the actions performed on the shared workspace – check-in, check-out, etc. In the Placeless Documents project [9], the authors have addressed the use of active properties to provide awareness information in some applications, but they did not integrate it as a *first-class citizen.*

3.2 Multiple concurrency control/reconciliation strategies

Distributed storage systems usually present some form of concurrency control to guarantee the correctness of each users execution and the consistency of the system. The most common approaches are based on the restriction of users actions and are implemented relying on some kind of locking mechanism (that may use a central server, tokens, quorums, etc). However, in distributed environments with disconnected computers, this type of approach is too restrictive imposing low (write) availability. To overcome this problem, it is possible to use an optimistic concurrency control policy, where users are allowed to perform concurrent updates that are later merged.

Using an optimistic approach, different strategies have been proposed in different systems – Bayou [16], IceCude [14], Coda [8], Rover [7] and operational transformations [15] are just a few of those. These strategies can usually be customized by users in a type-specific way. However, it seems that no single method is the best for all situations. Instead, different groups of applications require different mechanisms. Anyway, the use of semantic information has been pointed as the key to merge the concurrent streams of activity.

In DOORS, to maximize the semantic information available we propagate the sequence of operations executed by users – this way, it is possible to use the semantic information associated with the data type and with the operations. Using the concurrency control component it is possible to define for each data type the adequate strategy to handle and execute the updates stored in the log – for example, we have implemented different orders and op-

erational transformations. To tailor a given strategy, if necessary, it is possible to use the code of operations (that is executed in the servers). This way, it is very simple to test pre-conditions, post-conditions and even define multiple alternatives.

If the propagation of updates as operations is common (even in the Coda file-system this approach has been recently tested [10], although the objective had been to reduce the size of updates), the possibility of using different strategies and even define new ones is unusual (customization of a given strategy is what is usually available). However, our experience with some applications [12] suggests that this approach is very important to support different applications, allowing programmers to define the adequate concurrency control / reconciliation policy in the simplest way.

3.3 Partial caching mechanism

To allow disconnected operation it is necessary to cache data items. In Coda [8], as well as in other distributed file-systems, the unit of caching is the complete file. This approach allows a simple and clean file access semantics – as the file is the unit of storage, either it is possible to access the whole file or nothing at all (in a *cache miss*). However, when files are very large this approach may be a problem for resource-poor mobile devices. In object stores and object databases, as objects are the unit of manipulation, caching is usually done at the level of objects or set of clustered objects (e.g. [5]). However, an application usually manipulates a complex graph of objects. To support disconnected operation, it is necessary to cache all objects that will be used. This approach enables to cache only a subset of the data, but the determination of which objects must be cached is very complex because objects are usually very small and there are many connections among them.

In this new version of DOORS, we have adopted a strategy that can be seen as a tradeoff between those two. Programmers should define a coobject as a set of interconnected sub-objects. Each sub-object can still be a complex object and be implemented as a composition of many small objects. However, a sub-object is a unit by itself for caching purposes and it is possible to invoke operations on each sub-object independently – for example, a section of a shared document can be a sub-object. This approach enables the partial caching of a coobject, while allowing users to continue to work in the cached parts – a user can cache only the sections she wants to modify and edit them while disconnected. Obviously, the efficiency of our solution relies on the way programmers partition coobjects, but from our experience, this does not pose any major problem – for example, documents and calendars could be easily modeled. If a coobject only contains a sub-object or if every object is a sub-object, we have the previously described file-system and object database scenarios, but with a small programmers' effort it is possible to provide a better solution for resource-poor devices.

In several object-based systems (e.g. Globe [1], PerDiS[4]), related objects have been clustered together according to different policies and motivations – sharing a distribution strategy or minimizing the necessary binding and I/O operations are just a few of those. In DOORS, coobjects cluster together sets of sub-objects that share common (coobject) policies – concurrency control, awareness, etc. Each sub-object can yet be seen as a cluster of (common) objects that acts as a unit to the system – the whole sub-object can be cached independently of other sub-objects. These options are highly related with the goal of the system – the support of cooperation in a mobile and disconnected environment.

3.4 Blind operation invocation

In distributed storage systems, cache misses usually prevent users to continue using data. However, sometimes it is still possible to execute useful work despite the unavailability of some data. For example, in a shared calendar a user may request the scheduling of new appointments even when he cannot access the calendar. These requests must be validated in the servers, as they would need to be validated in a normal situation. In a structured document where each section may have multiple versions, it is also possible that a user wants to create a new version despite he cannot access the current versions.

To allow users to continue their work in these situations, thus minimizing the problems with cache misses, we have introduced the following two mechanisms. First, the operations invoked in the proxy of a sub-object can be always logged and later executed in the servers, even when the sub-object is not cached (in the client). In the previous examples, this mechanism can be used to request appointments in a shared calendar. Second, a sub-object not locally available can be instantiated in "replacement" mode, whenever there is a reference for it. Therefore, besides being able to invoke operations, it is also possible to check the expected results of the executed operations. Obviously, these operations are later executed in the servers, as usual, using the current state of sub-objects. This mechanism can be used in the previous shared document example. Besides the fact that each sub-object specifies which mechanisms can be used, the user can still choose to use them or not when he *"opens"* the coobject.

The proposed mechanisms can also be used in other ways. For example, it is possible to define a special sub-object that implements all possible operations in the coobject – e.g. in a shared calendar it would be easy to define all update operations that can be executed using additional parameters (e.g. the date) to identify the actual sub-objects where they should be executed. As a sub-object operation can invoke operations in other sub-objects, it would be possible to invoke all actions even without having, in the client, references to all needed

sub-objects. As the special sub-object would not require any state, it could be easily instantiated in "replacement" mode whenever necessary. As far as we know, no other system presents a similar approach.

3.5 Adaptation component

In mobile computing, it has been identified the need for adaptation of applications to the variation in the network conditions [11]. This approach seems particularly interesting in situations where it is possible to modify the data quality, such as in multimedia streams.

In DOORS, the adaptation component has been primarily thought to allow the immediate execution of operations in a server, whenever that is possible and interesting. To this end, it uses a remote method invocation mechanism provided by the system. In the previous version of DOORS, this remote invocation mechanism had already been used in sub-objects that act as surrogates of RDBMS – with the adaptation component it is integrated in a more clean way.

As the adaptation component processes all executed operations (queries and updates) it can also be used for other purposes. For example, we are currently using the adaptation component to synchronously maintain multiple copies of a shared document in different clients - these copies are part of a synchronous editing session. Although the multi-synchronous document editor is still under development, the adaptation component revealed itself as very important in this scenario. Another use that seems possible, but that it has not been yet pursued, is for the integration of sub-objects acting as surrogates of multimedia streams, such as video or audio streams.

3.6 Support for programmers

In the previous subsections we have presented some of the mechanisms introduced in DOORS to support disconnected operation. It is important to notice that different applications will define different type-specific solutions that use the available mechanisms in different ways. Taking this into consideration, the DOORS design choices have reduced the system core to the essential and have moved most of the operational "aspects" of data sharing to the coobjects, thus providing high flexibility to accommodate different solutions.

To fulfill the objective of providing support for the development of new applications, this flexibility is not sufficient – application programmers should be assisted in the implementation of their specific data types. To this end, we have created a data management object framework that decomposes each coobject in several components, each one responsible for a different aspect of the object "operation". We have also implemented a set of pre-defined components that execute different policies. Using the DOORS open object framework, application programmers may create new data types composing these pre-defined components with regular object classes (to implement sub-objects) – see examples in [13]. If necessary, programmers may create new components or extend any pre-defined one.

We are currently investigating the use of a new component model, associated language (ComponentJ) and tools to provide a better development environment to programmers. We expect to produce a simple and clean model to specify the composition of coobjects and, if necessary, to specify a component as the composition of even more basic components. Currently, coobjects are defined and composed as part of the coobject code, which is pre-processed to transparently log updates. The decomposition of objects in different components had already been used in other distributed systems projects – e.g. [2,1]. Although some of the identified components have similar functions, the different motivations lead to different frameworks and to different ways to use these frameworks (with different basic components).

4 Final remarks

To support disconnected and mobile computing, DOORS allows users to access and modify shared information even while disconnected. In this paper, we have focused in the characteristics that support disconnected operation. Some of these characteristics have been recently implemented in the DOORS prototype, as the result of previous experience. Although the system shares goals and approaches with several other systems, the important principle of allowing the support of different strategies distinguishes it from others that only support the customization of a single strategy – to this end, most of the functions that are usually part of the system have been moved to the objects. This approach is complemented with a strong support for the development of new data types, based on the DOORS object framework. This framework allows programmers to compose their type-specific solution using pre-defined components for several data sharing aspects.

The system also presents some the following mechanisms to support disconnected operation: objects are defined as a cluster of sub-objects to allow the partial caching of large objects; the blind operation invocation enables operation despite some cache misses; and the systems presents integrated awareness support. The concurrency control and adaptation problems have been addressed through components that allow different approaches. Some problems are under investigation: enhanced linguistic support; the possibility of providing partial caching using different object implementations; and a new event-dissemination architecture that supports the propagation of awareness information.

5 References

1. A. Bakker, M. Steen, A. Tanenbaum. From Remote Objects to Physically Distributed Objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Dec. 1999.

2. G. Brun-Cottan, M. Makpangou. Adaptable Replicated Objects in Distributed Environments. *Technical report INIRIA n° 2593*, May 1995.
3. P. Dourish, V. Bellori. Awareness and Coordination in Shared Workspaces. In *Proc. of CSCW'92*, 1992.
4. P. Ferreira, M. Shapiro, et al. PerDiS: design, implementation, and use of a PERsistent Distributed Store. In *Recent Advances in Distributed Systems*, LNCS 1752, Feb. 2000.
5. R. Gruber, F. Kaashoek, B. Liskov, L. Shrira. Disconnected Operation in the Thor Object-Oriented Database System. In *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, Dec.1994.
6. T. Horstmann, R. Bentley. Distributed Authoring on the Web with the BSCW Shared Workspace System. *ACM Standards View*, Mar. 1997.
7. A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. of 15th SOSP*, Dec. 1995.
8. J. Kistler, M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, Feb. 1992.
9. A. LaMarca, W. Edwards, P. Dourish, J. Lamping, I. Smith, J. Thornton. Taking the Work out of Workflow: Machanisms for Document-Centered Collaboration. In *Proc. of ECSCW'99*, 1999.
10. Y. Lee, K. Leung, M. Satyanarayanan. Operation-based Update Propagation in a Mobile File System. In *Proc. USENIX Annual Technical Conference*,1999.
11. B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of 16th SOSP*,1997.
12. N. Preguiça, J. Legatheaux Martins, H. Domingos, J. Simão. System Support for Large-Scale Collaborative Applications. Tech. Report 01-98 DI-FCT-UNL, 1998.
13. N. Preguiça, J. Legatheaux Martins, H. Domingos, S. Duarte. Data management support for asynchronous groupware. In *Proc.of CSCW'2000*, Dec. 2000.
14. M. Shapiro, A. Rowstron, A. Kermarrec. Application-independent reconciliation for nomadic applications. In *Proc. of SIGOPS 2000 European Workshop*, Sep. 2000
15. C. Sun, C. Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proc. of CSCW'98*, 1998.
16. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. of 15th SOSP*, Dec. 1995.