DAgora: A Flexible, Scalable and Reliable Object-Oriented Groupware Platform

Jorge Paulo F. Simão*, Nuno Manuel R. Preguiça*,
Henrique João Domingos, and José Legatheaux Martins

Dept. of Computer Science
Faculty of Sciences and Technology, New University of Lisbon
2825 Monte Caparica - Portugal

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

*Work partially supported by PRAXIS XXI scholarships.

Abstract

In this paper we describe a flexible, scalable, and reliable "object-oriented groupware platform" specially tailored as a foundation to support synchronous, asynchronous, and multi-synchronous groupware applications. The platform relies on an hybrid replication approach where volatile objects are actively replicated to support synchronous interaction, and persistent objects are lazily replicated to meet scalability and availability requirements.

1. Introduction

A wide body of groupware applications, both synchronous and asynchronous, have been implemented up to this point. However, most of these applications continue to be implemented from scratch, relying only on traditional client/server approaches, without specific support for groups, and on other non scalable and unreliable technologies. This imposes a considerable burden on application programmers and distracts them from applications main issues.

Some attempts have been made to provide general groupware frameworks and toolkits to help programmers in the process of structuring and implementing groupware applications [1,2]. These frameworks, however, do not usually provide generic mechanisms and impose or suggest particular politics targeted to a narrow range of applications. Current *distributed object systems* although very useful in general settings, are also only of limited utility as platforms to support groupware. Often, they fail to take into account the specific requirements of groupware applications, namely, the requirement for low-latency in system/application operation [3], and the need for shared feedback awareness of group and user activities [4]. It seems that a gap is yet to be filled between the mechanisms of distributed object systems, currently available groupware frameworks and toolkits, and the wide variety of possible groupware applications.

In this paper we outline an ongoing research work to devise an object-oriented groupware platform specially tailored as a foundation to structure and implement groupware applications. The requirement to support synchronous, asynchronous (both in connected and disconnected mode of operation), and multi-synchronous groupware applications, and the requirements for flexibility, scalability, and reliability, has lead us to deploy an hybrid object replication approach based on the peer object-group design pattern and a loosely connected, replicated object store service.

The rest of this paper is organized as follows. In section 2 we describe the overall architecture underlying the platform and provide the rationale for it. In section 3 the peer object-group design pattern is introduced as a means to structure synchronous groupware applications. In section 4 we describe the object storage service, which allows both connected and disconnected modes of operation. Session 5 refers to the combined use of the provided services and abstractions. Session 6 comments on current experience, as well as on ongoing and future work. Finally, session 7 concludes the paper.

2. An Architecture to Support Cooperation

Groupware application can broadly be characterized as synchronous or asynchronous. In asynchronous groupware users work not necessarily in the same time-frame and interact for long periods of time (e.g. in the joint development of a software project). In synchronous groupware users work in a tightly-coupled manner

during relatively short common time-frame (e.g. during a distributed meeting). The synchronous and asynchronous cooperation paradigms are not alternatives, but rather complementary; real work is most often performed alternating asynchronous work with synchronized periods. Furthermore, the synchronous and asynchronous characterizations only represent the edges of a continuous spectrum representing different time-frames between which users see each others work and interact. Intermediate degrees of interaction are possible. Some applications may even support different levels of synchronization - multi-synchronous applications. This calls for a flexible platform encompassing the mechanisms required for each case.

In our platform we make a clear distinction between volatile, actively replicated objects, and persistent, lazily replicated objects. Volatile objects are manipulated in the context of synchronous sessions, and are actively replicated at the users' workstations using tightly-coupled group communication services (section 3). This enables low-latency on object manipulation and activity awareness functionalities. The lifetime of the shared workspace manipulated in a synchronous session is limited to the duration of the session. Mapping to persistent objects, if required, is performed by the application or other layers of the system (section 5).

Persistent objects are managed by a global, distributed and replicated object storage service. Objects are aggregated in volumes which constitute the unit for replication. Each volume is managed by a collection of servers which lazily replicate the volume and the objects contained in it using epidemic techniques. Clients cache objects and perform operations locally, possibly in a disconnected mode of operation. Latter, logged object updates are reintegrated in the system. Conflicting updates are handled accordingly to operations semantics (section 4). Figure 1 illustrates the overall architecture.

The rationale to introduce two kinds of objects - actively replicated, volatile objects, and lazily replicated, persistent objects, is because the approach promotes system flexibility, efficiency, scalability, and reliability. Flexibility and efficiency is improved mainly because the typical granularity of update operations involved in synchronous groupware is finer than for the asynchronous case (e.g. in a text editing application, a character or paragraph granularity may be used for synchronous edition, and for asynchronous edition the chapter or document level may be selected). For synchronous groupware, actively replicated objects provide the means for the required levels of shared feedback awareness and thightly-coupled cooperation. On the other hand, persistent objects are used to store durable parts of the shared workspace. The programmer uses in each case the replication model and persistence options that better suit application needs. In particular, some objects may not require persistent storage and some persistent objects may not require synchronous cooperative editing.

Scalability is promoted because persistent objects are lazily replicated and users may access objects at the closest available server. In synchronous sessions, the number of expected interacting users is smaller so strong consistency object replication and group protocols are a realistic and feasible approach.

Finally, reliability and high availability arises from the high degree of object replication; users workstations replicate all objects required for user operation. In the synchronous case, the volatile objects ensure that a user can continue to work even if other users workstations or storage servers fail or become unreachable. In the asynchronous case, cached persistent objects allow users to operate in a disconnected mode of operation, and object replication at servers ensure high availability in object access. In both models, low-latency is achieved using appropriate optimistic replication techniques and particularly tailored protocols.

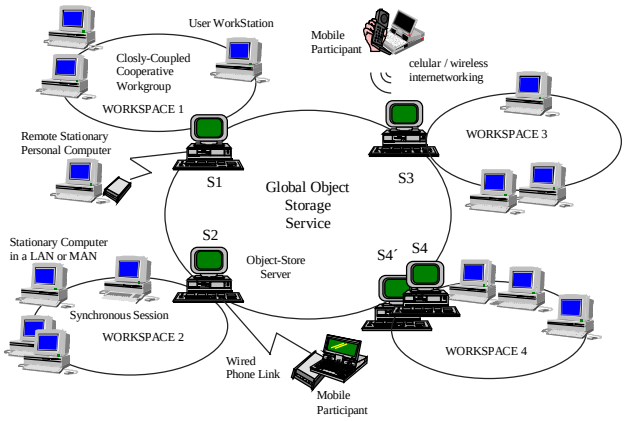

Figure 1 - The DAgora architecture.

3. Synchronous Groupware Support

We use the peer object-group design pattern as the main structuring abstraction to support synchronous groupware [5]. In the peer object-group approach, the shared workspace managed by applications is materialized as a collection of shared objects replicated amongst users workstations. Each workstation holds a replica for every object the associated user is currently accessing or working on. The set of replicas for a given object constitutes a peer object-group. Appropriate consistency criteria amongst the replicas is kept using group-communication services. Shared objects are mapped to

(object-)groups and operations on the objects are mapped to (reliable) multicast operations. Users gain access to objects by dynamically joining the corresponding object-groups - which may involve the transparent transfer of the object's current state to the local replica. When no longer interested in the objects, users leave the object-groups. Figure 2 schematically illustrates the model.

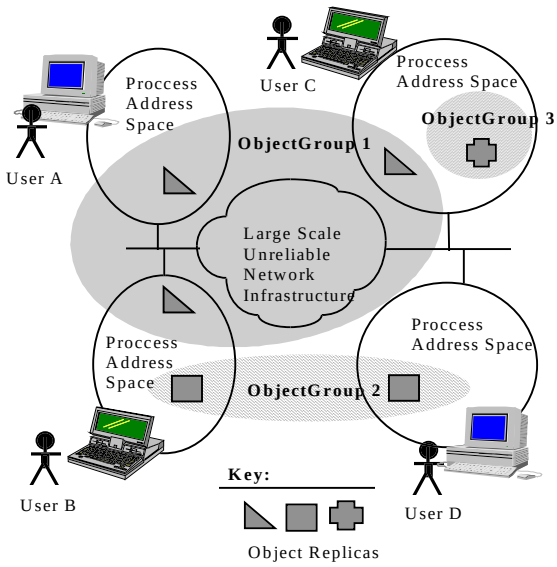

Figure 2 - The peer object-group design pattern.

Different shared-objects require different replication consistency requirements, meaning that group-protocols with different service semantics are required. To accommodate this diversity we have implemented a generic object-oriented framework for protocol implementation and composition as in [6]. Different protocol semantics are encapsulated in different classes, and concrete protocol layers are created as instance of those classes. Complete protocol structures (or stacks) are built attaching protocols objects together.

In the implementation of specific group-protocols we have taken in great consideration groupware specifics. Because users objects working-sets are expected to change often during the lifetime of a session and users should be able to enter and leave sessions dynamically, dynamic lightweight group membership services were used. In particular, we have specified a new membership and reliable multicast service semantics - *linear convergent synchrony*, which is weaker than *view synchrony* [7], the usually provided semantics by group communication toolkits, and can be implemented by protocols which incur less overhead for group membership management. Instead of requiring messages to be delivered in the same view by all members, we allow messages to be delivered in future views, but always before the view which removes the sending members [8]. The semantics and protocol are linear, i.e view and state merging is not allowed. This is motivated by the already available state reconciliation service provided by the external object store - case volatile objects are mapped to persistent ones. Moreover, it is expect that partitioning during the lifetime of a synchronous session is a rare event (provided that suitable failure suspectors are used). This approach also simplifies synchronous applications design. The protocol uses a specially tailored FIFO reliable multicast protocol, which does not incur high overhead for connection management.

To provide adequate levels in system response times, we have anticipated the need for optimistic ordering and concurrency control techniques. In particular, the Undo/Redo delivery paradigm is provided as an option to reduce update latency [9]. In this paradigm messages are delivered locally while asynchronously multicasted to the group. If ordering conflicts arise, some previously delivered messages/updates may have to be undone/redone, but operations semantics (e.g. the commutative property) are explored to reduce the probability of this event. Other ordering services, namely, total, causal and FIFO ordering of messages, and state transfer protocols for transparent and highly concurrent object state transfer are also provided.

Since the shared workspace of a synchronous session is constituted not by a single object-group but by a collection of them, services and abstractions are required for managing the workspace as a whole. This includes services for naming/binding to sessions, the creation and management of object-groups, and services to enable user activity awareness.

We have defined and implemented an extensible distributed object model which provides those services. In addition to allow programmers to create replicated objects supported by selected protocol structures, the model introduces the concept of a fully replicated Session object. A Session object is supported by a special bootstrap object-group, which all users must join to enter a session, that actively maintains information about the shared workspace, namely, information about created object-groups and users participating in the session. The binding information required to enter a session is fetched from an external service (e.g. the object store service). From an application programmer perspective, she/he can invoke the methods of a Session object to create, destroy, join or leave object-groups and to obtain information about users. A reactive programming style can also be used to act on session related events (e.g. a user entering

or leaving a session, or an object-group being created or destroyed). Conceptually, we abstract an application as a collection of shared object(-groups) and users organized around the fully replicated Session object. Figure 3 depicts an intuitive view of the distributed objects model.

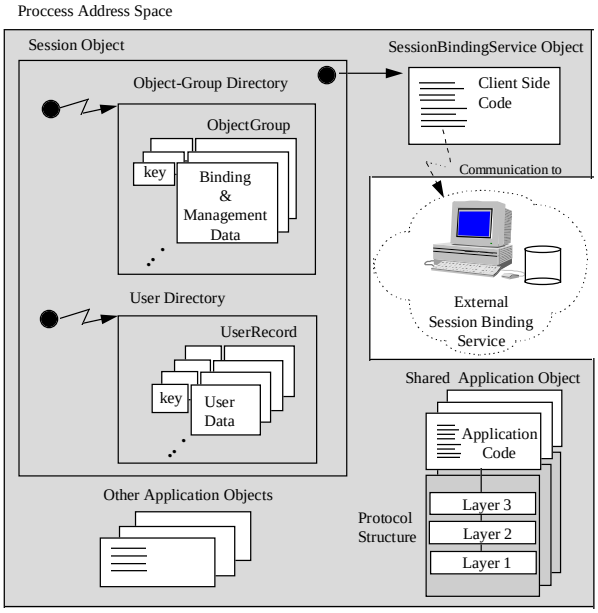

Figure 3 - Objects conceptual model.

4. Object Storage and Asynchronous Groupware Support

In this section we will describe the object storage service of our platform. As suggested before, to meet scalability and reliability requirements a system must employ intensive caching and replication techniques. Additionally, for effective support of asynchronous groupware a system should be able to: allow several users to modify the same data concurrently even if operating in a disconnect mode, not restricting their actions besides usual access control mechanisms and coordination rules' enforcement; conjugate all concurrent modifications in the resulting data; and enable type specific resolution of conflicting updates.

To gain access to the persistent objects managed by an object store a (client) user process creates a local copy of it; future operations are performed locally without requiring communication with the servers of the object store. Update methods invocations are logged by clients, until later re-integration with an object copy located at the object store. Updates performed concurrently by different users, are combined as logged updates are propagated to servers. Server procedures ensures that updates are ordered accordingly to consistency criteria selected on a per object class basis. If conflicts updates not amenable to be solved automatically by the system arise, users are asynchronously notified; task related coordination is expected to make this an unlikely event.

To provide a reasonable degree of autonomy, users local environments cache the objects needed for user activity. This enables them to continue work even when no server is accessible (due to communication failures or voluntary disconnection). Our current caching strategy makes decisions based only on recent access, but more complex and aggressive strategies based on pre-fetching of sets of related objects and statistical analysis of user activity can be considered [10,11].

Persistent objects are organized in volumes, which are sets of related objects. For user convenience, a volume is internally organized as an hierarchical space of objects identified by symbolic names (as in traditional file systems). We anticipated that different volumes be assigned to different collaborative work groups, allowing administrative boundaries and case-dependent access-control and security politics to be established.

Each volume is replicated by a (possibly) dynamically variable set of servers, using an epidemic communication infrastructure [12,13]. Defined pairs of servers communicate with each other, from time to time, to synchronize their current knowledge of objects state. Provided that the communicating pairs form a fully connected graph of replicating servers, object updates are eventually propagated to all servers. The concrete topological placement of server (pairs) should be conducted by the requirements of minimizing communication overhead (considering both server and client needs), distribution of server workload, and reducing probability of client isolation from all servers. Appropriate placement of servers, scheduling of server epidemic interaction, and structuring of objects in volumes is required to promote system scalability and availability.

When a pair of servers engages in a epidemic interaction, they must determine which updates must be propagated to the other server. This is be done by identifying update (sequences) with time-vectors [14], and comparing them with time-vectors reported by the peer. Update stability is checked using an acknowledge time-vector [13]. After server interaction, the newly received updates are logged for each object, and applied to the server's local copy accordingly to the ordering constraints selected. Class programmers may select one of the available orderings or (meta-)program new log orderings. Currently, we provide causal ordering, free ordering, (pessimistic) total ordering, optimistic total ordering with undo/redo, and optimistic total ordering.

When using optimistic total ordering, conflicting updates may cause the system to notify users, if unable to resolve them automatically (e.g. using operation semantic information). Figure 4 pictorially represents the object store architecture.

Because the set of servers managing each volume may vary with time, a state transfer mechanism exist to allow new servers to contact an existing server to obtain the volume's content and join volume's replicating server set. Membership information is managed by a special volume object which is propagated during epidemic interactions. Servers automatically resolve conflicting views of the membership before proceeding with normal operation.

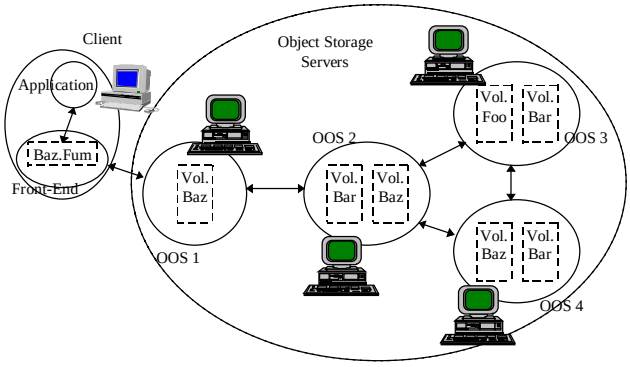

Figure 4 - Object Store architecture.

5. Combined use of Synchronous and Asynchronous Support

While we have made a clear distinction between volatile and persistent objects, in medium or long-term real cooperation scenarios the two abstractions must be used in combination. Persistent objects convey the durable part of the cooperative workspace, volatile objects provide the means for users to engage in tightly-coupled interactions. The synchronous and asynchronous interaction modes also benefit from each other. Information related with synchronous sessions (e.g. naming/binding, browsing/awareness and user information) can be persistently saved in the object store. On the other hand, conflicts updates to persistent objects detected by the object store and reported to users can be conveniently resolved using synchronous groupware merging tools. We are investigating issues related to the transparent mapping of the two kinds of objects for those cases in which that is required.

6. Experience, Ongoing and Future Work

Because we wanted to maximize flexibility, allow application and system components to be loaded on-demand, and support heterogeneity, we have chosen the Java language for implementing our system [15]. The integration with the Web was an additional motivation.

As an initial effort to test the suitability of the synchronous groupware support, we have implemented a demo white-board tool. It is a simple tool which manages a shared drawing canvas and requires only one object-group to be implemented. It was tested with only a small number of users in a local network. In this restricted setting, system response has revealed to be quite acceptable, i.e. system performance did not suffer significant degradation when operating on replicated shared objects. Additional experience and performance measures are required to analyse system behaviour in more general environments.

To test the suitability of the asynchronous groupware support we have defined a simple class of structured persistent objects. These objects are composed of containers, that contain leaves and/or other containers, and leaves, containing data with (possibly) multiple versions. We have also implemented a simple text editor that allows several users to asynchronously edit the same document. A document is supported by a persistent object and the document structure is mapped to the object's structure (e.g. a document is a container of chapters, a chapter a container of sections or a text leaf, and so on). When all users save their changes, the final document reflects all changes, and concurrent changes to the same leaves (chapters/sections text) are resolved by creating multiple versions of the conflicting components.

Many potential work directions were revealed during the course of our work. In the synchronous support, we plan to continue the process of specifying suitable group-communication semantics and implementing new protocols to support object-groups. In particular, we expect to develop layers for light-weighted groups, which in turn may call for the definition of multiple-group service semantics. Failure-detectors consistency should also be addressed. In the asynchronous support, we intend to develop a generic event notification service to provide users with shared feedback awareness of activities related with the persistent workspace. We also intend to implement the conflicts notification service with the generic notifications mechanisms. Common to both the synchronous and asynchronous support, we expect to tackle the always important issue of access control and security, and plan to develop additional tools and applications to help validating more clearly the usefulness of the abstractions outlined in the paper.

Finally, we intend to build appropriated linguistic support to simplify the task of applications programming.

7. Conclusions

Our work contribution is two-folded: identify the abstractions required to adequately support groupware and study the technical problems involved in the realization of them; devise a platform based on those abstraction to be used in the development of realistic groupware applications.

We believe that the provision of several kinds of objects by an object-oriented groupware platform promotes flexibility in application programming, because groupware applications are very broad in range. Programmers are free to make use of the abstractions that better suit their needs. In particular, an object store is suitable to support asynchronous cooperation and manage the persistent part of shared workspaces, and peer object-groups are a suitable base abstraction to structure and implement synchronous applications. Because modern cooperation scenarios may involve many entities, scattered world-wide, possibly using the Internet as the main cooperation infrastructure, scalability and reliability are important requirements. Still, further research is yet required to more clearly validate the usefulness of the identified abstractions and mechanisms. Incorporation in standard distributed object systems can also be a contribution in that direction.

References

[1] Mark Roseman, and Saul Greenberg, "GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications", Proc. ACM CSCW 92, November 1992.

[2] Tim Kindberg, George Couloris, Jean Dollimore and Jyrki Heikkien, "Sharing Objects over the Internet: the Mushroom Approach", IEEE Global Internet 96, London, November 1996.

[3] C.A. Ellis, S.J. Gibbs and G.L. Rein, "Groupware - Some issues and experience", Communication of the ACM, vol. 34, n.1, January 1991.

[4] Paul Dourish, and Victoria Belloti, "Awareness and Coordination in Shared Workspaces", Proc. CSCW 92, November 1992.

[5] Jorge Simao, J. Legatheaux Martins, Henrique Domingos, and Nuno Preguica, "Supporting Groupware with Peer Object-Groups", USENIX COOTS`97, "Reliable Distributed Objects Panel", Portland/Oregon, June 1997.

[6] Robbert van Renesse, Kenneth Birman, Roy Fridman, Mark Hayden, and David A. Karr, "A Framework for Protocol Composition in Horus", Proc. 14th IEEE International Conference on Distributed Computing Systems, 1994.

[7] Kenneth P. Birman, and Thomas A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", Department of Computer Science, Cornell University, 1987.

[8] Jorge Simao, "System Support for Distributed Synchronous Groupware Applications", MSc. Thesis, Dept. Computer Science, New University of Lisbon, July 1997.

[9] Alain Karsenty, and Michel Beaudouin-Lafon, "An Algorithm for Distributed Groupware Applications", Proc. 13th IEEE International Conference on Distributed Computing Systems, 1993.

[10] G. Kuenning, "The Design of the Seer Predictive Caching System", IEEE, 1994.

[11] J. Kistler, and M. Satyanarayanan, "Disconnected Operation in the Coda File System", Proc. 13th ACM SOSP, 1991.

[12] Rivka Ladin, et al., "Providing High Availability Using Lazy Replication", ACM Transactions on Computer Systems, 10(4):360-391, November 1992.

[13] Richard Golding, "Weak-consistency group communication and membership", Ph.D dissertation, University of California - Santa Cruz, December 1992.

[14] D. Parker, et al., "Detection of Mutual Inconsistency in Distributed Systems", IEEE Transactions on Software Engineering, vol. SE-9(3):240-247, May 1983.

[15] James Gosling, and Henry McGilton, "The Java(tm) Language Environment: A White Paper", Sun Microsystems, 1995.