Supporting Synchronous Groupware with Peer Object-Groups

Jorge Paulo F. Simao, Jose A. Legatheaux Martins,
Henrique Joao L. Domingos, and Nuno Manuel R. Preguica
Dept. of Computer Science, Faculty of Sciences and Technology,
New University of Lisbon

Supporting Synchronous Groupware with Peer Object-Groups

Jorge Paulo F. Simão*, José A. Legatheaux Martins,
Henrique João L. Domingos, and Nuno Manuel R. Preguiça*
Dept. of Computer Science,
Faculty of Sciences and Technology, New University of Lisbon
2825 Monte Caparica - Portugal
{jsimao, jalm, hj, nmp}@di.fct.unl.pt
*Work partially supported by PRAXIS XXI scholarships.

Abstract

We propose the peer object-group design pattern as a suitable architectural solution to structure and implement synchronous groupware applications. We discuss a reliable group-communication subsystem and a distributed objects model, implemented in Java, used to realize the approach.

1. Characterizing Groupware

"Computer Supported Cooperative Work" - CSCW, deals with the use of computer systems by people to cooperatively work on common tasks. Groupware is software specially built to allow people to work cooperatively.

Groupware and user interaction can be roughly classified in two broad classes - asynchronous or different-time, and synchronous or same-time. When using 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). When using synchronous groupware users work in a tightly-coupled manner during relatively short common time-frames (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.

We are in the process of investigating generic system-level services to support and allow simple development of robust groupware applications. In this text we will focus on system support for synchronous groupware. In particular, we will discuss structuring and programming abstractions based on group-communication and object-groups specially devised to help in the development of synchronous groupware applications (SGA).

A virtually common feature to all SGA is the provision of a shared workspace which users use to communicate and cooperate during a synchronous session. For acceptable productivity, users need to have an accurate notion of what the state of the shared workspace is. In particular, users should have mutually consistent views of the state of the workspace and should see each others actions as soon as possible. SGA are also interactive applications by nature, so it is required that the system responds and evolves accordingly to users expectations [1]. Users desire short or immediate response times; preferably, similar to that found in single user applications. Users do not find acceptable to wait a considerable amount of time to perform some operation (e.g. to update a shared object). Because users tend to divide/phase tasks into smaller sub-tasks and user communication and cooperation has multiple facets, SGA should be seen not as monolithic applications but rather as a collection of tools aggregated in the context of a single session (e.g. including a tool for shared drawing, a tool for message exchanging, a tool for text or document editing, tools providing audio and video channels, user activity awareness, coordination, etc.). From a software-engineering perspective, it is also preferable to use a generic multi-tool approach than to provide all the functionality from scratch in every application.

2. Design Alternatives to Support Distributed Synchronous Groupware

A commonly used architectural approach to support distributed SGA is the client-server paradigm. A central server is used to manage the shared workspace, to perform concurrency control on user accesses, and to provide other session related services (e.g. user activity awareness). User processes use the server to operate on shared resources and to disseminate information to other users. While this is a very well understood paradigm, and it is simple to realize, it presents major drawbacks: fault-tolerance and scalability, since it is based on a central server. Moreover, performance can be somewhat injured by this architectural approach, although clients may replicate/cache parts of shared workspace in order to mitigate the problem. A variation is the centralized application-distributed interface approach, where a single application multiplexes user interaction and disseminates output across several user interfaces. It presents the same problems as the client-server approach, and is in general less flexible.

An alternative is the replicated-server, or object-group, approach. A group of servers actively replicates objects and/or service state. Even if a subset of servers crashes or becomes unreachable, the service will be available as long as some of them remain reachable (one or the majority - depending on consistency criteria). This approach is very suitable for many distributed fault-tolerant services, but still presents some drawbacks in the context of SGA. Because users want to have accurate views of the shared workspace, and because users actions are largely driven by other users actions, extra mechanisms for event notifications are required.

Preliminary experience on scalable, fault-tolerance, distributed systems has suggested that migrating complex system functionality from servers to clients may be a suitable design option. This argument, the need for flexibility in tool building, and the low-latency requirements of SGA, suggests, in our view, a much more natural approach - the peer object-group approach.

3. The Peer Object-Group Design Pattern

In the peer object-group approach the shared workspace managed by SGA is materialized as a collection of objects replicated amongst users local environments. Each local environment 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. Consistency amongst the replicas is kept by a group-communication subsystem implementing appropriate consistency criteria. Shared objects are mapped to object-groups and operations on the objects are mapped to (reliable) multicast operations. Figure 1 schematically illustrates the model.

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.

Users keep accurate views of the shared state since updates are received by all object-group members; no provisions for additional notification mechanisms is required. Latency in object manipulation is improved because no intermediate entities are present. Fault-tolerance and availability is also improved; a K-degree of fault-tolerance is achieved as long as K+1 members keep copies of shared objects.

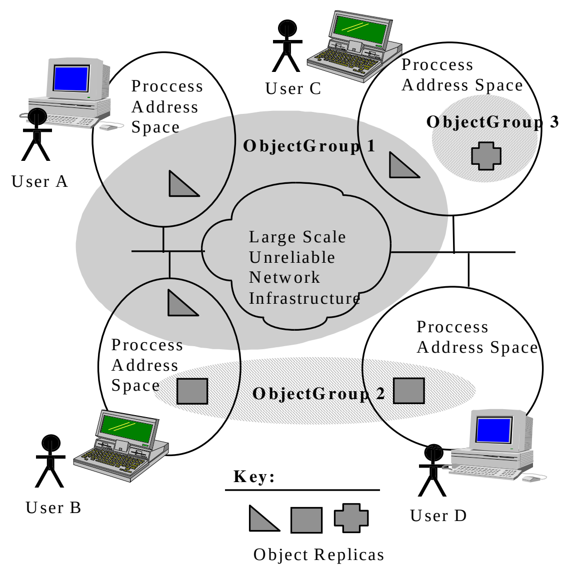

Figure 1 - The peer object-group design pattern.

Different shared-objects may have different replication consistency requirements, meaning that the underlying group-communication sub-system should provide group-protocols with different service semantics. The selection of object-groups granularity must inevitably be tool and protocol driven; the lighter-weighted the protocols are, the finer the granularity can be.

Object persistence is not addressed by this bare model. While it is desirable that some objects outlive sessions, that facility is provided by the asynchronous groupware support, and will not be discussed in this paper.

4. Object-Oriented Group-Protocol Implementation and Composition

The essential component to realize the peer object-group approach is the group-communication subsystem, which provides group membership and message passing services. Those services are implemented by group-protocols and accessed through the combined use of a user-to-protocol service request interface and a protocol-to-user event notification interface. The former includes methods for message sending and multicasting as well as group management. The later includes methods for message delivery and group membership view change notifications.

Implementing group-protocols is a complex task, mainly because they must convert an unfriendly system environment into a friendly one. A good engineering option is to use a modular approach in designing and implementing group protocols. Multiple (micro-)protocol layers, each implementing some specific service, are stacked to built complex protocol services [2].

To realize the peer object-group we have implemented an object-oriented framework to allow the convenient implementation and composition of group-protocols. Because we want to maximize flexibility, allow application and system components to be loaded on-demand, and support heterogeneity, the Java language was a natural choice [3]. The integration with the Web was an additional motivation.

In our framework protocol layers are implemented as objects of special classes, which implement group services related programming interfaces. Complete protocol structures (or stacks) are built attaching protocols objects together. To allow simple construction of protocol structures, protocol structures description strings and generators are used. Description strings convey information about which protocols should be used to build a particular protocol structure and the topological relationships between the layers. Protocol structure generators parse strings and generate the correspondent protocol structures, by dynamically loading the layer classes and creating the layer objects.

In implementing specific group-protocols we have considered SGA 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 the "standard" *view synchrony* [4], but can be implemented by protocols which incur in less overhead for group membership management. The semantics and implemented protocol are linear, in the sense that no view merging is allowed, because we assume that state reconciliation due to network partitions is performed using the external data storage services. The protocol uses a specially tailored FIFO reliable multicast protocol.

We have also experienced with optimistic ordering techniques to reduce system response-time. In particular, the Undo/Redo delivery paradigm was used to reduce update latency [5]. In this paradigm messages are delivered locally while asynchronously multicasted to the group. If ordering conflicts arise, some previously delivered messages/updates are undone. Object operations semantics (e.g. the commutative property), is explored to reduce the probability of conflicting updates. The protocol sits on top of a sequencer based total ordering and state transfer protocol, which achieves high levels of concurrency even during process joins.

5. Object-Groups Management and Session Services

In addition to a group-communication subsystem SGA programming can benefit from the provision of other more specific services. Mechanisms and services are required for the naming and binding to sessions, for the management of the object-groups in the shared workspace, and to enable user activity awareness.

We have defined and implemented an extensible distributed object model to structure and implement SGA, which tackle the above issues in an integrated manner. In addition to the collection of peer object-groups which constitutes the shared workspace, we have introduced the notion of fully replicated Session object. A Session object is supported by a special bootstrap object-group, which all user processes must join to enter a session. Binding information required to enter a session is fetched from an external binding service.

A Session object's main purpose is to store and manage directories which hold information about created object-groups and users participating in the session. Object-groups information includes binding and management data (e.g. protocol structures description and replication options). User information includes human readable data about human users (e.g. user full name, user photography, e-mail address, the

Web home-page, etc.). Both object-groups and users are identified by names, and are represented as Java classes which can be application derived to convey additional information. Conceptually, we abstract an application as a collection of shared object(-groups) and users organized around the fully replicated Session object. Figure 2 depicts an intuitive view of the distributed objects model.

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).

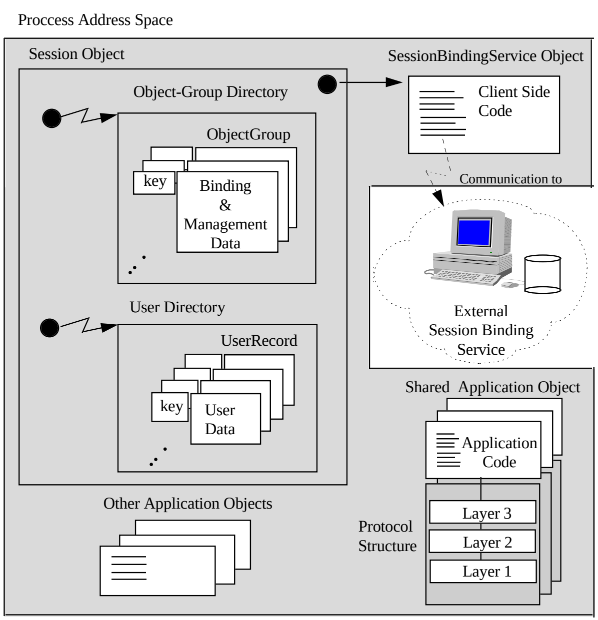

Figure 2 - Objects conceptual model.

6. Experience and Future Work

As described, we have developed up to this point the framework for protocol composition, a set of stackable group-protocols and the distributed objects model. We have tested the suitability of our ideas implementing 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 reveled 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 analyze system behavior in more general environments.

Many potential work directions were revealed during the course of our work. We plan to continue the process of specifying suitable group-communication semantics and implementing new protocols. In particular, we expect to develop layers for light-weighted groups - multiplexing many groups into a small number of groups, in order to increase system performance when many fine granularity object-groups are used. This may call for the definition of multiple-group service semantics. The issue of failure-detectors consistency will also be addressed, i.e. ensure that the membership layers of several protocol stacks have similar views of what the connectivity state is. Also, we expect to tackle the always important issue of security and access control.

We also plan to develop additional tools and applications, to help validating more clearly the usefulness of the abstractions discussed in this paper. We will consider enhancing our object model with additional structuring abstractions and common services as more experience is gained. Finally, we intend to build a stub-compiler to simplify the task of shared objects programming.

References

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

[2] van Renesse, Robbert, Birman, Kenneth P., Friedman, Roy, Hayden, Mark, and Karr, David A., *A Framework for Protocol Composition in Horus*, in proceedings of the 14th IEEE International Conference on Distributed Computing Systems, 1994.

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

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

[5] Karsenty, Alain, and Beaudouin-Lafon, Michel, *An Algorithm for Distributed Groupware Applications*, in proceedings of the 13th IEEE International Conference on Distributed Computing Systems, 1993.