17

An object group model and its implementation to support cooperative applications on CORBA

Fábio Moreira Costa and Edmundo Roberto Mauro Madeira
Department of Computer Science
Universidade Estadual de Campinas - UNICAMP
Cidade Universitaria, Campinas, Sao Paulo, Brazil
e-mail: {fmc,edmundo}@dcc.unicamp.br

Abstract

Cooperative applications are emerging as a promising new field for the use of distributed systems technology. This new type of application also presents new requirements, challenging research and development of distributed systems. One of these requirements refers to the group nature of the interactions in these new environments, which has not found suitable support with traditional distributed system mechanisms. In this paper we propose a model of object groups to support this kind of applications. An implementation model based on distributed objects is also proposed and a prototype constructed on top of a CORBA (Common Object Request Broker Architecture) platform is described.

1 INTRODUCTION

Group communication is a key factor in computer supported cooperative work (CSCW) applications. The patterns of interaction found in this kind of application frequently involve several users taking part in some common task that is to be accomplished through cooperation among them. These users are often considered members of a cooperative group, sharing some common resources. In this context, users' actions need to be propagated to the rest of the group, so that all of its members can see each other's actions in a consistent way. An action could be thought as any form of updating of the resources shared by the group. Member actions can be communicated to the group by means of multipoint interactions between the source member and the rest of the group.

Cooperative applications can be further characterized by the inherent distribution of the users across a communication network. In this distributed environment, applications are structured as a number of autonomous cooperating components, each one running on behalf of its respective cooperative user. Therefore, some kind of distribution support must be provided so that application components can effectively and reliably interact with its peers in the system.

In this paper we propose an *object group* model as an effective abstraction to deal with the complex interaction patterns of CSCW applications. In this context, one can structure a group of objects, each object representing a group member, to support the cooperative application. These objects act as intermediaries between their respective users and the group, performing

A. Schill et al. (eds.), *Distributed Platforms*
© Springer Science+Business Media Dordrecht 1996

tasks that are specific to group structuring and interaction. The proposed model is composed of object structures, services to perform group structuring and communication, as well as policies to control services execution. An implementation of a Group Support service based on this object model is described. The prototype was implemented on top of an ORB (Object Request Broker) platform [24].

The work is part of the Multiware Platform, that is being developed at the University of Campinas, Brazil [5, 22]. This platform offers ODP services to the applications, mainly for cooperative applications. The Middleware layer of Multiware is composed of an ORB (ORBeline [1] in the current version, and Orbix [2] in the next one) and some services objects, like the Trading Object [21], the Transaction Support Object and the Group Support service we are proposing [10]. Over the Middleware layer, there is the Groupware Layer, that deals with the different kinds of CSCW applications. This layer would be probablely the main user of the Group Support service (although this service can also be used directly by the applications). Figure 1 illustrates the architecture of the Multiware platform.

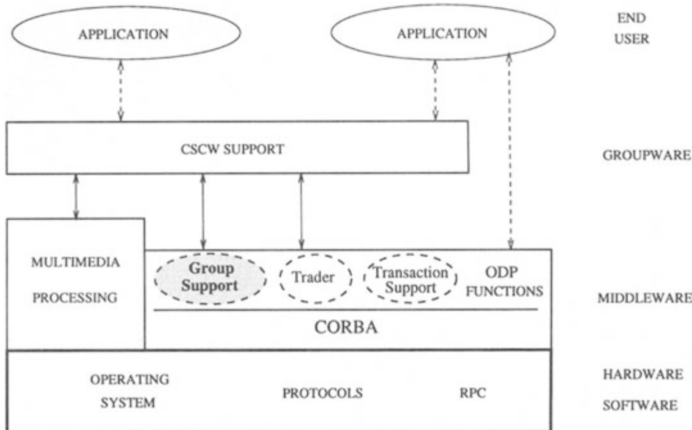

Figure 1: Architecture of the Multiware Platform

The next section presents a characterization of the fundamental requirements of CSCW applications found in the development of distribution support platforms. Section 3 describes in detail the proposed model for object groups, while in section 4 we propose a distributed objects infrastructure to support these groups. Section 5 describes the implementation of the Group Support prototype on top of the CORBA and section 6 describes how the Group Support service can be used. Finally, in section 7, some concluding remarks are presented concerning the experience of implementing object groups using CORBA as well as the applicability of the proposed model.

[1]ORBeline is a mostly complete implementation of the CORBA 1.1 specification, being a trademark of Post-Modern Computing, Inc.

[2]Orbix is a Registered Trademark of IONA Technologies Ltd.

2 SUPORTING COOPERATIVE APPLICATIONS

CSCW systems are becoming one of the main applications of distributed systems technology. However, most of the traditional distributed platforms do not provide adequate support for this new class of applications. This has forced CSCW developers to deal with somewhat "low-level" issues related to distribution that could be better provided as uniform mechanisms by the underlying platform [9].

One of the major sources of problems is related to the new styles of interaction found in CSCW applications [27]. The most common form of interaction involves more than merely two parties, which exchange or share information as part of the cooperation process. Therefore, it would be helpful if the distributed platform could provide the developer with mechanisms for multicast of interactions among the possible many cooperative users, maintaining the consistence and reliability of these interactions, as well as the consistence of the group's shared state. It is also important to make group communication as similar as possible to one-to-one interactions. In this sense, equivalent group interactions, such as the multiple responses generated by a group invocation, must be *collated* together to be delivered to the users as a unique interaction coming from the group. Another important facility that would be of great benefit, if provided at the platform level, refers to the capacity to control the structure of a group in terms of its constituent members and to allow changes on this structure to address the dynamic nature of cooperative groups [4].

3 THE OBJECT GROUP MODEL

In this section we present the proposed model of object groups to be used as a structuring and communication abstraction to support distributed cooperative applications. The model is composed of two main parts, one related to group composition and functionality and the other to the services used to provide these functionality. We derived the concepts presented in this model from previous work related to process or object groups in distributed systems [8, 2, 18, 14, 11]. However, these early systems aimed to provide groups for fault tolerance and for replication management, whereas our model aims to provide groups specifically to support cooperative applications. The model was also strongly influenced by the Reference Model for Open Distributed Processing (RM-ODP) [15], specially by one of its coordination functions, called the *Group Function*, which aims the prescription of group-based services to support the development of open distributed cooperative applications [17].

A group of cooperative users is represented as a set of interrelated objects, each object corresponding to a user and performing application-independent functions to allow group interaction between the users. A cooperative user, in this context, is normally represented by an application object, such as a cooperative text editing tool, that performs application level actions on behalf of the real user. In this way, an application object interacts with its group object instead of interacting directly with the other group members. This group object is therefore responsible for receiving and interpreting group interactions from its user and to forward them to the group. It is also responsible for the reception of group interactions originated by other group members, giving them the appropriate treatment and delivering them to the user. Group interactions are usually materialized in terms of messages sent from one member to other members of the group.

When a group is created, an initial set of members is assigned to it and they can therefore interact in a group fashion by means of the exchange of group messages. This group structure can be changed when new members join it or when existing ones leave it. The only users that

can interact with the group are just its members. This feature characterizes our object groups as *closed* with respect to group interaction [6, 3]. If a user outside the group wants to interact with it, this user can either join the group or send a message to one of its members who then forwards the message to the rest of the group. In addition, our group model does not provide explicit support for inter-group interactions (inter-group communication could be addressed by the applications by means of overlapping groups).

Another important feature of the model refers to the fact that a user may send a group interaction to only a subset of the group members. To do so, the user can refer directly to member names or, alternatively, it can use a concept known as member *roles* so that the interaction is sent only to the members associated with the specified roles. A *role* can be related to some function which a subset of the members performs in the group at the application level; a given member can have several roles, which can also be used for authorization purposes (to verify if a given member is authorized to perform a given group service). There is a distinguished member role, the *coordinator role*, that is assigned by default to the user that created the group.

3.1 Event Ordering

In object group environments, event ordering is important to assure consistent group interaction, so that all members always have the same view of the shared environment. Event ordering can be achieved using basically two types of mechanisms and its variations. They are known as *total ordering* and *causal ordering*, respectively. The first type consists of sequencing all group events (messages) sent by all group members, so that all of them receive these events in the same order. Total ordering is used in the Amoeba system to provide ordering of distributed group events [19]. On the other hand, causal ordering mechanisms only order those events which have a potential cause and effect relationship, leaving concurrent (independent) events unordered.[3] Causality is determined using the rules proposed by Lamport in his classical paper [20] and was firstly used in the context of groups by the Isis system [1]. Another, radically different approach, followed by the V system [8], consists of having no ordering at all, leaving the task of ordering to the applications. This approach is based on the assumption that causal dependencies among events can only be effectively determined if one has knowledge of the semantic information contained in the events [7].

Our group model takes advantage of these two basic approaches, providing ordered as well as non-ordered delivery of group communication events. It is expected that applications using the group support services will choose event ordering to be applied to critical events (those that can change the shared environment) and no ordering to the others. Applications can also choose the second approach in cases where the overhead of determining the order of events is not tolerable.

The mechanism for ordered delivery adopted in our implementation uses a protocol similar to the first one presented in [19] for the Amoeba system. It is based on a centralized event sequencer, where all events to be distributed to the group are sent to the sequencer, which assigns order information to the event and then effectively distributes the event to the destination group members. At the destinations, the order information is examined to determine if the event has been received in order. If so, it is promptly delivered to the application. On the other hand, the event cannot be delivered until all the previous events in the order have been received and delivered. A future version of the prototype could provide a causal ordering mechanism, but we found that the current version has performed considerably well for the types of applications and the moderate sizes of groups we aim to support.

[3] The term *event* is used here to refer to both application messages (interactions) and to group support control messages.

3.2 Group Support Services

A set of object group support services is provided so that applications have a means to structure groups and to interact with them. The access to the services is done through a set of primitives available at the interface of the group support system. In this way, a given service is requested by calling the respective primitive with parameters that specify and control its execution. To give a level of flexibility to the group support services a set of *policies* can be assigned to them, prescribing alternative paths to its execution, as suggested by the RM-ODP's Group Function and applied in [11]. This is based on a consensus found in the CSCW community concerning the need for a clear separation between policies and mechanisms [26] to provide more flexible services in a coherent and extensible fashion. Our group support system provides a set of basic policies for each group service. These policies are meant to be common for a wide range of applications (including collaborative text editing, voting systems and teleconferencing) and can be specified without knowledge of the application-level semantics. It is expected that more complex policies, that depend on application knowledge, can be implemented using the services for group communication provided by the group support system. In the Multiware environment, such complex policies will be implemented by the Groupware layer.

The set of group support services is based on the Group Function of the RM-ODP. This function is defined in terms of four basic functionalities: group *interaction*, interaction *collation*, event *ordering* and group *membership* control. In our model, we designed a set of services for membership control, called *group structuring services*, and another set of services for interaction (*group communication services*). The other two functionalities of the Group Function are addressed by the internal mechanisms used for reply collection and event ordering. These services and functionalities were proposed to support cooperative applications at the distributed systems level. Application-level functionalities are provided by higher layer entities, such as the Groupware layer of the Multiware platform or even by the applications themselves.

In the following, we provide a description of the support services for group structuring with its respective parameters, options and policies.

- **Group Creation**: this service allows applications to specify the constitution and configuration of a new group. The functions performed by this service include: membership confirmation of each proposed member; the establishment and initialization of the infrastructure (described in the next section) to support the group; and to provide information about the group to each of its members. The embedded policies for group creation are essentially concerned with the confirmation of membership for each user that was proposed as a member of the new group:

  - *creation with queries*: determines the need to query each proposed group member to know if it is willing to participate in the group; with a negative answer the member is considered out of the group.
  - *unrestricted creation*: determines the group creation with "mandatory" participation of each proposed member; this policy should also be used in cases where the application implements an external independent policy that is applied before the beginning of the group creation process.

The other services for group structuring are used to change the configuration of existing groups. This services are subject to the following set of common built-in policies:

- service execution after the group (by means of queries to the group members);

- service execution after approval from the group coordinator;
- service execution without the need of aproval (queries).

In addition, the execution of these services is subject to the event ordering mechanism, because their execution changes the group's shared environment as viewed by the group members. In this way, all the group members see the execution of these services at the same point in the sequence of ordered events in the group (including communication events). These group structuring services are listed bellow:

- **Group Termination:** this service allows explicit termination of existing groups, according to the group policy and to the authorizations for its execution.
- **Group Join:** this service allows users outside a group to engage in the cooperation process, becoming group members. The new member is provided with group support infrastructure and with information about the group. After that, the other members of the group are informed about the new member.
- **Member Leave:** members are allowed to quit a group by using this service. The mechanisms for group leaving involve, after authorization checking and application of the policy, the notification to all remaining group members informing that the old member has quit, followed by the deletion of all references and information about that member.
- **Policy Change:** allows the replacement of the policy assigned to a given service. This service involves a notification to be sent to all group members informing the change.
- **Authorization Change:** allows someone (that has authorization) to change the set of authorized roles of a service.
- **Member Role Change:** allows a new role to be assigned to a member, replacing existing ones or simple being added to them; a given role may also be removed.

Group communication services are accessed using a separate set of primitives for message distribution, which have to support two types of group interaction, similarly to the types proposed in the RM-ODP [16], named:

- *interrogations*, two-way interactions (messages) that could represent, at application level, the invocation of group services or even simple questions to group members (such as in a voting system). Interrogations require responses (called terminations in the ODP terminology) from their recipients and these responses need to be collated so that the user application receives a single and equivalent group response.
- *announcements*, one-way messages.

There is a distinguished primitive to send group interrogations, another to send group announcements and yet another to reply to interrogations. The primitives take as parameters information concerning the message to be distributed (such as group name, list of destinations, sender of the message, necessity of ordering and collation mode) as well as the message contents. They are described bellow:

- **Group Interrogation:** to distribute messages corresponding to interrogations. Another required parameter refers to the form in which replies have to be collected; this parameter represents the collation policy suggested by the Group Function definition, but in a "per interrogation" basis. Three collecting modes are provided:

- effect the collation of all replies, delivering a single logical reply that consists of their concatenation;
- wait only for the first n replies and deliver their collated version to the application; the other replies are discarded;
- no collation, where the replies are delivered independently as they are received.

A record of the interrogation messages is created to allow the correct collection and collation of their replies. The invocation of this primitive returns an identifier that permits the application correctly to associate delivered replies with its original interrogation

- **Group Announcement:** to distribute one-way messages
- **Group Reply:** to send responses to group interrogations; the application needs to identify only the received invocation that generated the reply.

Message distribution may or may not be subject to event ordering, and this is specified on a "per message" basis, with the exception of responses, which are always communicated without ordering treatment. It is up to the application to decide which messages will be distributed with ordering and which will not. Cooperative applications can choose ordered message distribution for messages that modify the shared environment of a group, so that all members receiving the message will do it in the same logical instant (at the same point in the group's sequence of ordered events, including those events that represent the execution of a group structuring service). In addition, the delivery of messages to all the non-failed group members is assured by the group support service.

4 THE IMPLEMENTATION MODEL

The group support model is implemented through a distributed object configuration where each object performs well-defined complementary tasks. As long as these objects have no shared memory, they must rely upon some strong type of information exchange, such as reliable message communication to cooperate and properly maintain coherence among the distributed, perhaps replicated, information. This communication support is currently based on CORBA and will be described in the next section.

The object configuration is composed of four classes of objects, each one performing a distinguished role in the task of supporting cooperative groups. These classes and their functions are summarized bellow:

- **SGstruct**: constitutes the core of the group support system with respect to structuring services, named group creation and termination, member joins and leavings, and policy, role and authorization changes. These services are accessible at the interface of this object. The SGstruct object is responsible for policy application and authorization checking, concerning the execution of these services. As long as all these services have group-wide effects, there is a requirement for cooperation between the SGstruct and the other objects of the group support configuration. In addition, most of these services depend on event ordering, demanding cooperation with the centralized element of ordering. The SGstruct object is unique in a given cooperation domain, where many users and various cooperative application groups can exist. For fault tolerance, this object can be replicated using passive replication techniques, and a further development could provide replication of the SGstruct

for performance. Despite this, as long as structuring services are expected to be much less frequent than communication services, this centralized approach is reasonable.

- **SGcoordinator**: this is the core object in the event ordering mechanism. For each group, there is an object from this class, which is responsible for the addition of sequence numbers that assign order to group events. All events to be ordered are sent to the SGcoordinator that forwards (multicasts) them to all the destinations. Event distribution can be done either by the use of native multicast facilities of the network or by simulation on top of point-to-point communication facilities. This object can also have a passive replica to address fault tolerance.

- **SGlocal**: for each group member, there exists an object from this class that provides applications with the interface to the message distribution services. This object is what really constitutes the group objects of the model. For messages requiring ordering, the SGlocal forwards them to the SGcoordinator, which does the effective distribution. On the other hand, non-ordered messages are directly distributed (multicast) by the SGlocal. On the receiving side, the SGlocals are the objects responsible for message delivering to the applications, imposing the order determined by the SGcoordinator, when applicable. This object is also responsible for reply handling and collation. This SGlocal object is unique to each group member, and is responsible for interfacing with all groups in which the member participates. Note that between the SGlocal and the real user there may be application objects that carry out the application-defined tasks outside the scope of the group support system, such as a collaborative text editor.

- **Failure Handler**: this is the object responsible for failure confirmation and treatment and its final goal is the maintenance of group consistency. Failures are detected by one of the three objects described above and are reported to the failure handler which tries to determine its cause as well as the involved object or objects. The recovery actions taken by the failure handler depend on the type of the involved object. The failure handler is unique within a domain, but can be made redundant by means of a passive replica to tolerate its own failures.

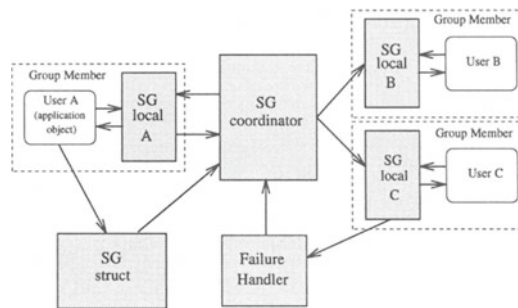

Figure 2: Possible interactions between group support objects

Figure 2 shows the possible interactions between the objects involved in a given group. Group support users (applications), for example, can directly interact only with its respective SGlocal

(for group communication) and with the SGstruct object (for group structuring services); in this way, the objects SGcoordinator and Failure Handler can only be reached by the other objects of the group support system (not by the users). The figure also shows that a group member can be regarded as the configuration constituted by a user and its SGlocal object.

The interactions between these objects to accomplish real group support services are illustrated by two diagrams in figures 3 and 4. These diagrams show the temporal sequencing of the interactions, which are represented by arrows between the entities involved in the interaction. Group users are represented by vertical lines, while group support objects are represented by vertical bars (each user is represented near its respective SGlocal object).

In figure 3, the execution of a group structuring service (the join of a new user) is presented. The member which is willing to join the group sends a request to the SGstruct object, which, after application of the policy (consulting the group members), creates the new SGlocal (only if it doesn't yet exist), providing it with information about the group. In what follows, the SGstruct calls the SGcoordinator, which provides ordering information through a notification event that is sent to all group members (including the new one). After this notification, which is delivered in the same logical instant to all group users, the new member is effectively considered a member of the group.

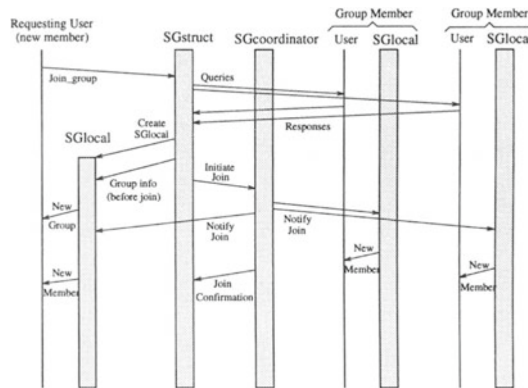

Figure 3: Typical interactions between group support objects for the *join* of a new member to the group

Figure 4 shows the sequence of interactions between group objects for the ordered message distribution service. The process is initiated by a member that has a message to be distributed to the group. It specifies the message contents and requirements and requests the distribution to the associated SGlocal. The SGlocal verifies that the message requires ordering, adds to it some control information and then forwards it to the SGcoordinator, which, after the addition of a global sequence number, multicasts the message to all of its destination members. After receiving the message, each SGlocal object can only deliver it to the application after the assurance that all of its previous messages or events were already delivered.

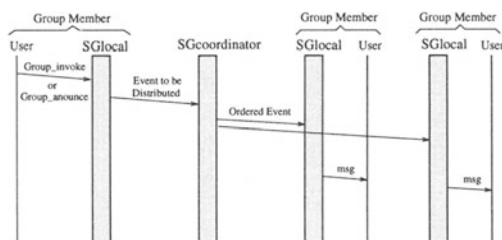

Figure 4: Typical interactions between group objects for group communication with event ordering

5 PROTOTYPE IMPLEMENTATION ON CORBA

CORBA [24] provides a suitable support environment to structure systems as distributed object configurations, so that the system's functionality is distributed among several object components. In addition, the provision of a standardized language for interface definition (the IDL) contributes to a large degree of interoperability between different object systems and to a large level of portability across different platforms.

In most CORBA implementations, such as ORBeline [25] (which we have used for this implementation), there are two major ways to create objects: one static, by simple instantiation of the corresponding class, and another dynamic, where the object is registered at runtime with the Object Adaptor (in ORBeline, the Basic Object Adaptor) and activated when it is firstly invoked.

In this way, the objects of the group support system are defined as CORBA object implementations in a dynamically structured environment. Only two objects, the SGstruct and the Failure Handler, are statically created. The other group support objects are created at run time when they are needed: a new SGcoordinator is started whenever a new group is created, whereas the SGlocal objects are created whenever a user becomes a member of any group for the first time. The SGlocals are created on the same host where its user is running, while the SGcoordinator is normally created on the host of the group creator. In ORBeline, as in most CORBA implementations, there is no means to create objects at remote hosts, so that we have statically to install a small starter object on each host where some member can be found [4].

In this CORBA environment, each object performs the client and server roles at different times, acting as a client to request services from other objects and as a server to provide its own services to other objects and to the application. For information exchange, this style of interaction is also used, with the information source acting as a client that makes a request (containing the data as a parameter) to a server object which will receive the information [5]. All the interactions between the objects of the group support system, and between them and the applications, use this common mechanism. The Object Request Broker (ORB) is responsible for the transparent delivery of these requests to the appropriate objects, so that communication with remote objects becomes closely similar to method invocation in a shared address space environment (such as the C++ method invocations). Figure 5 illustrates the role of the ORB as a communications platform between the group support objects.

[4] We expect this need to be overcome as implementations of the *Life Cycle* object service become largely available

[5] It is argued that message-oriented middleware (MOM) would be more suitable for this kind of interaction

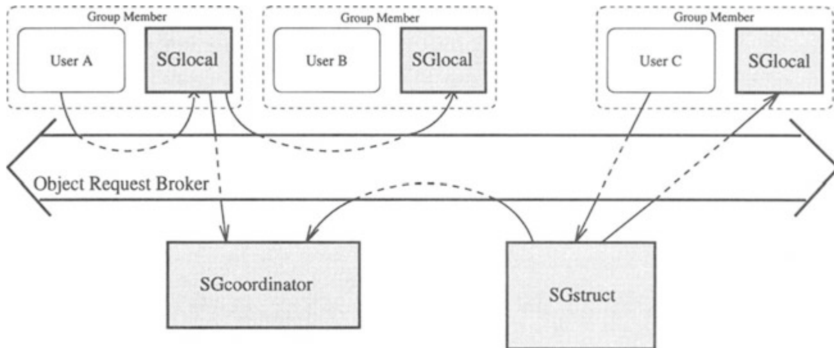

Figure 5: The relationship between the group support objects and the ORB

The interfaces to all these objects are defined in IDL, so that they can be exported to clients and used by them to request services. These interfaces are presented in Appendix A. These exported interfaces contain operations for the invocation of each of the group support services described in section 3.2. On the other hand, the application objects that interact with the group support system must also export an interface with predefined operations that allow the group support objects to call them in order to deliver messages and to indicate other occurrencies (such as the join of a new member to the group). The object implementations may be done using any language that has a mapping (and a compiler) from IDL, such as C or C++.

In a distributed environment there is a natural potential for concurrent operation among the various system components, and this potential must be explored to enhance the overall system's performance. Therefore, it's not desirable that client objects stay blocked until the requested service is being executed by the server, because these objects could also be serving other requests. This is specially true in non-multi-threaded implementations of CORBA, as the one we have used [6]. In our prototype, this question is addressed by the extensible use of *one-way* object invocations, an asynchronous communication mechanism provided by CORBA. In this way, a client can be sure of the delivery of a request through the ORB's *exception handling* mechanism. In addition, invocation results are returned via one-way invocations in the reverse direction (identifiers are used when needed to properly associate the returned results with their respective invocations). Other advantage of using this non-blocking mechanism is the minimization of possible deadlock conditions, although all object interactions were carefully designed to avoid these conditions.

Another important point to note about the prototype refers to the way group communication was implemented. As CORBA currently does not provide a multicast invocation facility at its static interface (through stubs), we had to simulate this type of communication using point-to-point (one-way) invocations. The atomicity of this simulated multicasts is then assured through unique identifiers associated with them.

The prototype was implemented on top of ORBeline for SunOS, which provides nearly all the concepts and facilities found in the CORBA 1.1 specification, plus some extensions that approach the 1.2 version of the CORBA, including a comprehensive mapping from IDL to C++ (the language of implementation). The ORBeline mapping from IDL to C++, however, has some

[6]Although ORBeline is also available in a multi-threaded version

minor differences from the official one adopted by OMG [23]. Apart from this, we choose not to use the extended features of ORBeline, so that portability to other CORBA implementations is made easier. A future version of the prototype will be migrated to Orbix [13], to integrate the larger context of the Multiware platform.

6 USING THE GROUP SUPPORT SYSTEM

Cooperative applications to be build on top of the group support system have to express the interactions among its components in terms of invocations of the group support services. A typical cooperative application undergoes three main phases, when it uses different functionalities of the Group Support service:

- *establishment of the cooperation environment (session)*, when a corresponding group is created through the group creation service;
- *cooperation phase*, when group members collaborate to perform the cooperative work, which is made through the communication services available at the SGlocal interface. During this phase the group structure can also be altered, for example, to accommodate new members or to change service authorizations or group policies;
- *session termination*, when the corresponding group must be dissolved in a consistent way, through the use of the group termination service.

For structuring services, such as group creation or member quit, the correspondence between application functions and group services is obvious. On the other hand, to use communication services, the application needs to decide on the granularity of the interactions (how much information should be conveyed by each group message). A collaborative text editor, for example, whose data is distributed across the multiple cooperative editing components, may alternatively choose to propagate each new or modified character, line or paragraph, depending on its synchronization and requirements and overhead tolerance. These units of information are then encapsulated in a message and distributed to the other cooperative editors through one of the message distribution primitives (these primitives are described in appendix A). Message distribution should be ordered so that the document being edited is always the same to all group members. In addition, the application could also use the message distribution services to express other types of interactions not directly related to the informations being manipulated, as in the case of application-level control interactions (to implement protocols to control concurrent access to the shared environment, for example).

We argue that these group support services can be used by any kind of distributed application that involves cooperation and coordination of its components. However, the services was specifically designed to support cooperative applications as part of the Multiware architecture. In this context, application-level common functions are supported by the Groupware layer, whereas support at the distributed systems level is provided within the Middleware layer by this Group Support Service. Examples of applications that will use this service would be collaborative text editing and teleconferencing systems.

7 CONCLUDING REMARKS

We proposed a model of object groups with features and functionalities that are considered appropriate for cooperative applications. CORBA was chosen as the underlying distributed platform for reasons such as the suitability of its object model to represent distributed group structures in a transparent environment, as well as for interoperability of the prototype with the other components of the Multiware platform, which may be developed on top of different hardware platforms. In addition, as long as CORBA is becoming a "de facto" standard for distributed client/server systems, we expect a great portability level of our work to different platforms.

We addressed most of the requirements presented by CSCW applications, such as the need to structure applications as groups of components capable of cooperation and the need for a powerful communication abstraction to support the style of group interaction. Our approach is derived, although slightly different, from previous work in the distributed systems community that proposed process or object groups as a tool for the design of reliable and efficient group-like applications. The differences are based on the facility to distribute group interactions to subsets of the group members and the combination of the two alternative approaches for group communication in the form of ordered and non-ordered message distribution. In addition, we deal with questions that arise at the application level, but are common for a wide range of applications, such as the control of policies and member roles, group membership management and the treatment of the two basic styles of interaction, namely interrogations and announcements. We hope that the provision of such a support at the platform level will substantially simplify the design and construction of distributed cooperative applications.

However, we didn't directly address another important requirement of emerging CSCW applications that relies upon multimedia interaction, such as teleconferencing or telepresence, namely the real time support for interactions. We chose not to deal with real time support because we believe that such a support must be provided by the underlying communications platform separately from the group management concerns. Performance of such applications may also be compromised by the simulated multicast communication mechanisms used in the prototype. We chose this way to perform group interactions, in contrast with the use of real network multicast, for portability reasons, because group members can be spread on several communication networks, not all of them having buit-in multicast transport capabilities. For CSCW applications that do not use multimedia information exchange, such as collaborative text edition, voting systems and decision support systems, the current version of the prototype can offer appropriate support. We argue that any kind of distributed application which have group-like interaction patterns between its components could take advantage of such kind of group support system. In addition, the proposed model for group support (and, in a larger extent, the implemented prototype) is independent from the underlying communications platform. If, for example, future versions of the CORBA specification provide some type of support for multi-peer object communication with the performance required by multimedia applications, only minimal changes to the object group model will be necessary to adapt it to the new environment.

ACKNOWLEDGEMENTS

The authors wish to thank FAPESP, CNPq (project PROTEM GEOTEC) and CAPES, that have partially funded this work.

References

[1] Birman, K. and Cooper, R. (1990) The ISIS project: Real experience with a fault tolerant programming system. Technical Report TR90-1138, Department of Computer Science, Cornell University.

[2] Birman, K., Schiper, A., and Sephenson, P. (1991) Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314.

[3] Birman, K. (1993) The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53.

[4] Blair, G. and Rodden, T. (1991) CSCW and distributed systems: The problem of control. Technical Report CSCW.1.91, Lancaster University, UK.

[5] Cardozo, E., Loyolla, W.P.D.C, Madeira, E.R.M., Magalhães, M.F. and Mendes, M.J. (1994) Multiware platform: An open distributed environment for multimedia cooperative applications. *IEEE COMPSAC'94, Taipei, Taiwan.*

[6] Chanson, S.T., Liang and Neufeld, G.W. (1990) Process groupss and group communications: Classifications and requirements. *IEEE Computer*, 23(2):57–66.

[7] Cheriton, D.R. and Skeen, D. (1993) Understanding the limitations of causally and totally ordered communication. In *Proc. of the 4th Symposium on Operating Systems Principles*, pages 44–57. ACM.

[8] Cheriton, D.R. and Zwaenepoel, W. (1985) Distributed process groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107.

[9] Chevalier, P.-Y, Riveill, M. and Saunier, F. (1994) Towards a generic system support for cooperative applications. Technical Report 94-wet-ice, Université de Savoie - LGIS - Campus Scientifique.

[10] Costa, F.M. and Madeira, E.R.M. (1995) Cooperative groups support in the Multiware platform. In *PANEL'95 - XXI Latin American Conference on Informatics*, Canela, RS, Brazil. Brazilian Computer Society.

[11] Edwards, N. and Oskievicz, E. (1993) A model for interface groups. ANSA Architecture Report AR002, Architecture Projects Management Limited.

[12] Ellis, C.A. Gibbs, S.J. and Rein, G.L. (1991) Groupware: Some issues and experiences. *Communications of the ACM*, 34(1).

[13] IONA Technologies Ltd (1995), Dublin, Ireland. *Orbix Programmer's Guide - Release 1.3.1.*

[14] Isis Distributed Systems Inc. (1993) *Object Groups: A response to the ORB 2.0 RFI.* OMG Doc. 93-4-11.

[15] ISO / ITU-T (1994) *ITU-T X.901 — ISO/IEC CD 10746-1. ODP Reference Model - Part 1. Overview.*

[16] ISO / ITU-T (1994) *ITU-T X.902 — ISO/IEC CD 10746-2. ODP Reference Model - Part 2: Descriptive Model.*

[17] ISO / ITU-T (1994) *ITU-T X.903 — ISO/IEC CD 10746-3. ODP Reference Model - Part 3: Prescriptive Model.*

[18] Kaashoek, M.F., Tanembaum, A.S. and Verstoep, K. (1993) Group communication in Amoeba and its applications. *Distributed Systems Engineering Journal*, 1:48–58.

[19] Kaashoek, M.F. and Tanenbaum, A.S. (1992) Efficient reliable group communication for distributed systems. Technical Report IR-295, Vrije University, Amsterdam.

[20] Lamport, L. (1978) Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565.

[21] Lima, L.A.P. and Madeira, E.R.M. (1995) A model for a Federative Trader. In *Proceedings of the International Conference on Open Distributed Processing*, pages 155–166, Brisbane, Australia.

[22] Madeira, E.R.M. (1995) Multiware platform: Some issues about the middleware layer. 7th. IASTED - International Conference on Parallel and Distributed Computing and Systems, pp. 162-166, Washington, USA.

[23] Object Management Group (1994) *Revised C++ Mapping Submission.* OMG TC Doc. 94-9-14 (Final submission to the Object Request Broker 2.0 C++ mapping RFP).

[24] Object Managemet Group (1991) *The Common Object Request Broker: Architecture and Specification - Rev. 1.1.*

[25] PostModern Computing Technologies, Inc. (1994), Mountain View, California, USA. *ORBeline User's Guide*

[26] Rodden, T. (1992) Supporting cooperative applications. Technical Report CSCW.11.92, Computing Department, Lancaster University, UK.

[27] Rodden, T. and Schmidt, K. (1992) Putting it all together: Requirements for a CSCW platform. Technical Report TR-CSCW-9-92, Computing Department, Lancaster University.

8 BIOGRAPHY

Fábio Moreira Costa is a professor at the Department of Statistics and Informatics of the Universidade Federal de Goiás (Brazil). He received his B.S. degree in Computer Science from the Universidade Federal de Goiás in 1989 and his MSc degree in Computer Science from the Universidade Estadual de Campinas - UNICAMP (Brazil) in 1995. Current Address: Departamento de Estatística e Informática, Universidade Federal de Goiás, Cx. Postal (PO Box) 131, CEP (Zipcode) 74001-970, Goiânia, Goiás, Brazil.

Edmundo Roberto Mauro Madeira is a professor at the Department of Computer Science of the Universidade Estadual de Campinas - UNICAMP. He received his MSc in Computer Science in 1985 and his PhD in Electrical Engeneering in 1989 from UNICAMP. Currently, he is a coordinator member of the Multiware Project at UNICAMP.

A INTERFACE DEFINITIONS

The interfaces to the group support objects, as they are seen by the applications, are presented here (operations used for group support internal control are not showed). The interfaces are described according to the Interface Definition Language (IDL) syntax proposed in the CORBA specification [24].

A.1 Interface to the SGstruct

```
module GroupStructuring{

  typedef sequence<string> RoleList;

  struct MemberList{
      string member_name;
      RoleList member_roles;
  };

  enum PolicyTypes{ ... };

  typedef sequence<PolicyTypes> PolicyList;

  // INTERFACE OF THE SGSTRUCT OBJECT
  interface SGstruct{

     // FOR GROUP CREATION:
     oneway void create_group (in string creator_id,
                               in string group_name,
                               in MemberList members,
                               in PolicyList policies);
     // FOR GROUP JOIN:
     oneway void join_group (in string requester_id,
                             in string group_name,
                             in Member new_member);
     // FOR GROUP LEAVE:
     oneway void leave_group (in string requester_id,
                              in string group_name,
                              in Member member);
     // FOR GROUP TERMINATION:
     oneway void finish_group (in string requester_id,
                               in string group_name);
     // FOR POLICY CHANGE:
     oneway void change_policy (in string requester_id,
                                in string group_name,
                                in string service_name,
                                in Policy new_policy);
     // FOR AUTHORIZATION CHANGE:
     oneway void change_authorization (in string requester_id,
                                       in string group_name,
                                       in string service_name,
                                       in ChangeType change_type,
                                       in string authorized_role);
     // FOR ROLE CHANGE:
     oneway void change_role (in string requester_id,
```

```
                              in string group_name,
                              in string member_name,
                              in Change_Type change_type,
                              in string new_role);

  }; // end interface
}; // end module
```

A.2 Interface to the SGlocal

```
module GroupCommunication{

  enum DistributionMode {WHOLE_GROUP, BY_ROLE, BY_MEMBER};

  typedef sequence<string> DestinationList;

  enum CollationMode {ALL_REPLIES, FIRST_REPLY, NO_COLLATION};

  struct MessageStructure{
      string group_name;
      string message_sender;
      DistributionMode distr_mode;
      DestinationList destinations;
      CollationMode collation;
      boolean order_required;
      string message_contents;
  };

  interface SGlocal{

     // TO DISTRIBUTE GROUP INTERROGATION MESSAGES:
     // (returns the invocation identifier to allow the application
     // correctly to associate the received replies;
     // a return value of zero means that the message was not
     // accepted for distribution, because it was not properly specified)
     //
     unsigned long group_invoke (in MessageStructure message);

     // TO DISTRIBUTE GROUP ANNOUNCEMENT MESSAGES:
     // (returns TRUE if the message was accepted for distribution)
     //
     boolean group_announce (in MessageStructure message);

     // TO SEND RESPONSES TO GROUP INTERROGATIONS:
     // (returns TRUE if the message was accepted for distribution)
     //
     boolean group_reply (in MessageStructure message,
                          in unsigned long interrogation_id);

  }; // end interface
}; // end module
```