Departamento de Informática
Faculdade de Ciências e Tecnologia
UNIVERSIDADE NOVA DE LISBOA
2825 Monte Caparica - Portugal

Technical Report

DI-UNL-02/97

Supporting Synchronous Groupware with Peer Object-Groups

Jorge Paulo Ferreira Simão (jsimao@di.fct.unl.pt)
José Augusto Legatheaux Martins (jalm@di.fct.unl.pt)
Henrique João Lopes Domingos (hj@di.fct.unl.pt)
Nuno M. Preguiça (nmp@di.fct.unl.pt)

- DÁgora Project Development Team -

DÁgora PROJECT
Secção de Arquitectura e Sistemas de Computadores
Grupo DÁgora

URL Ref: http://www-asc.di.fct.unl.pt/DAgora

Keywords: Object-groups, Reliable group-communication and protocols, CSCW/Groupware, Java

Note) This technical report relates with the on-going research developed in the DÀgora Project - DÁgora is an integrated and flexible platform and framework for large scale computer support of collaborative work applications and services.

Supporting Synchronous Groupware with Peer Object-Groups

Jorge Paulo Ferreira Simão
jsimao@di.fct.unl.pt

José A. Legatheaux Martins
jalm@di.fct.unl.pt

Henrique João L. Domingos
hj@di.fct.unl.pt

Nuno Manuel Preguiça
nmp@di.fct.unl.pt

- DÁgora Architecture Development Team -
(http://www-asc.di.fct.unl.pt/DAgora)

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

Abstract:

In this report we present the peer object-group design pattern. This pattern is a suitable system solution to support distributed *synchronous groupware applications* (SGA). In the report we argue that major benefits can be achieved if we drive system design by SGA specifics. We describe a flexible object-oriented subsystem implemented in JAVA, based on reliable group communication. The system provides an object-group programming environment specially tailored to support SGA. Simple implementation of realistic SGA is shown to be possible using our framework.

Keywords: Object-groups, Reliable group-communication and protocols, CSCW/Groupware, Java.

1 - Introduction

The message oriented paradigm of group-communication and the object-group design pattern have been identified as useful abstractions in the construction of reliable distributed applications. Group-communication systems provide a set of related services where groups of entities are addressed as a single unit, and messages are reliability disseminated accordingly to some service semantics [1]. Object-groups add a structuring model to group-communication, abstracting entity groups as groups of objects sharing a common state, which are kept consistent according to some criteria [2]. The object-

This work is partially supported by JNICT and the PRAXIS XXI scholarsip program - grants BM6969/94, BM6926/95.

group model maps quite well to group-communication, and has the merit of extending object-orientation concepts to fault-tolerant distributed settings.

Although group-communication and object-groups have a wide scope of applicability, we are primarily concerned with *Computer Support for Collaborative Work*, i.e. *Groupware*. In our work we have tried to characterize groupware applications as regards their system level desired services, since most of the current technology still puts considerable burden on application programmers. In particular, we have identified the desired system services to the two broad classes of groupware applications - "asynchronous" and "synchronous". The former benefits mainly from specially tailored high-available services (e.g. data storage and coordination services). For the latter we consider as essential the support of programming abstractions based on group-communication and object-groups.

In this paper we argue not only that object-groups and group-communication are very useful abstractions to build synchronous groupware application (SGA), but also that substantial benefit can be achieved if we take the approach of recasting these abstractions in the context of SGA. Instead of using/implementing generic group-communication subsystems (e.g. Isis [3], Horus [4], Totem [5], Transis [6], etc.), and generic object-group model adhering tools and environments (e.g. Electra [7]), it is our strong belief that reflecting SGA specifics in the realization of such abstractions will bring considerable improvements to SGA design, efficient implementation, and acceptance.

This paper is organized as follows: in section 2 we introduce the peer object-group design pattern. In section 3 we describe our Java framework for group-protocols implementation and composition. In section 4 we extend the framework with a distributed object model used to structure and implement SGA. In section 5 we describe particular group-protocol implementations. Section 6 comments on experience and future work, and section 7 on related work.Finally, section 8 concludes the paper.

2 - Characterizing and Supporting Groupware Applications

From a system-support level perspective, groupware applications can be most usefully classified in two classes - asynchronous, or different-time, and synchronous, or same-time. In asynchronous settings user interaction usually lasts for long periods of time with different users working not necessarily in the same time-frame (e.g. the joint development of a software project). On the other hand, with synchronous applications, users work and cooperate in a tightly-coupled manner during a common time-frame (e.g. a distributed meeting).

The two cooperation paradigms are not really alternatives, rather they should be taken as complementary. Often, real work is performed alternating between asynchronous work, and synchronized periods. Therefore, a generic architecture tailored to help in the development of groupware applications must encompass services and abstractions that accommodate both worlds.

Providing generic system support for asynchronous groupware applications calls mainly for global, high-available distributed services, namely: services for data/object storing, object version control and management, support for disconnected operations, and coordination of multi-participant activities. Scaling requirements are more demanding than for the synchronous case since the number of users is potentially greater, and the degree of coordination is looser. Object consistency criteria will typically be weaker, and lazy replication techniques will be the more realistic solutions to provide acceptable levels of service. Although we are also working in this area[1], this paper will only focus on system support for the synchronous case.

The common feature of synchronous applications is the provision of a shared workspace, which users use to communicate and cooperate. For acceptable productivity, users need to have an accurate notion of what the state of that 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.

While in synchronous sessions, users tend to divide and/or phase tasks into smaller sub-tasks. Moreover, several independent activities may be performed in parallel or in sequence. This means that SGA should not be seen as monolithic applications, but 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 behavior awareness, coordination, etc.). From a software-engineering perspective, it is also preferable to use a generic multi-tool approach, rather than to provide all the functionality from scratch in every application.

2.1 - Design Alternatives to Support SGA

One common, and expected, architectural approach to support distributed SGA is the client-server paradigm. A central server is used to manage the shared workspace and perform concurrency control on user accesses. While this is a very well understood paradigm, and is simple to support, it presents major drawbacks: fault-tolerance and scalability, since it is based on a central server, and low response-times. There is also the distributed-interface variation, where user interaction is multiplexed on a single centralized application. It presents the same problems, 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 remain 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 state of shared objects, extra mechanisms for event notification on object updates are required.

Preliminary experience on scalable, fault-tolerant 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, promotes, in our view, a much more natural approach - the peer object-group approach.

2.2 - The Peer Object-Group Design Pattern

The above characterization of SGA and implied requirements leads to the peer object-group approach. The shared workspace, common to virtually all SGA, can be seen as a collection of objects replicated amongst user processes. Each process holds a replica of every object the associated user is currently accessing or working with. The collection of replicas of each shared object constitutes an object-group. Consistency amongst the several replicas of the object will be kept through the use of 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.

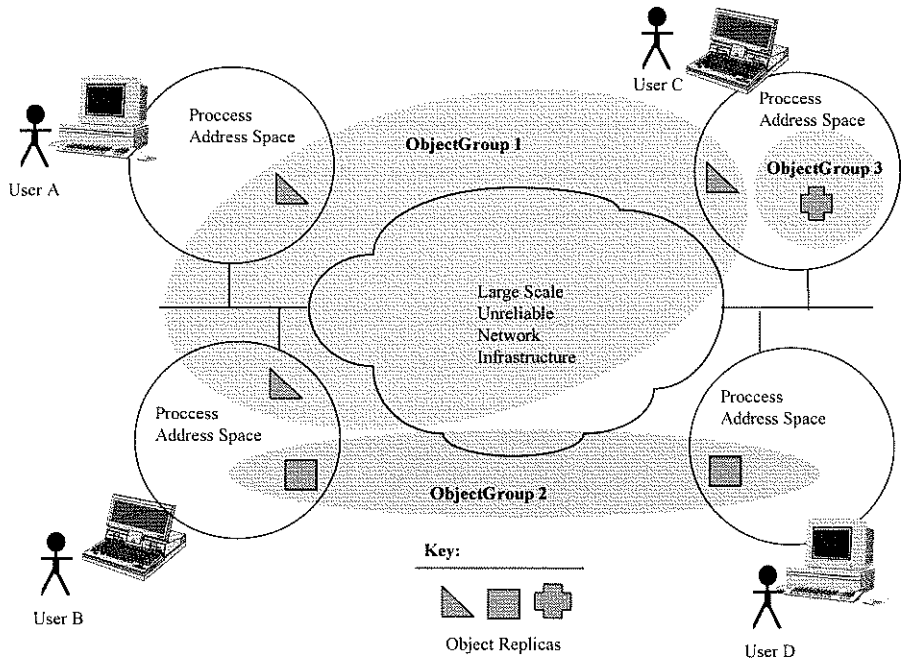

Figure 1. The peer object-group design pattern.

To gain access to an object users dynamically join the corresponding (object-)group - which involves the transparent transfer of the object's current state to the local replica. When a user no longer needs to

---

[1] This work is being done in a project called DÁgora: a generic platform and flexible framework to support large scale collaborative applications and services.

access the object her/him simply leaves the object-group, and stops receiving messages sent to it. User processes only need to maintain state information and handle updates to objects the user is interested in.

Different shared-objects may have different replication consistency requirements, meaning that the underlying group-communication subsystem should provide group-protocols with different service semantics. Some objects require strict consistency, while in others, message ordering, (and several other aspects of service semantics), can be loosen.

The selection of object-groups granularity must inevitably be tool driven. The programmer should be more or less aware of the operation cost associated with each protocol stack, and consider it when mapping shared application objects to object-groups. The more lightweight the protocols are, the finer can the granularity be. A typical and reasonable compromise is to map the shared space of each application tool into a different object-group.

The peer object-group design pattern allows users to keep accurate views of the shared state, since each state update is seen by all processes accessing the object-group. No provisions for additional notification mechanisms are required (as in the client-server or in the replicated-server approach). Latency in object manipulation is improved because no intermediate entities are present. Moreover, fault-tolerance and availability is maximized during the course of a session since no external services are required. As long as users keep connected (and the underlying group-protocols allow), they can continue to work. Even if user processes occasionally crash or became mutually unreachable, a K-degree of fault-tolerance is achieved has long as K+1 members keep copies of shared objects.

Object persistence is not addressed by the bare model. While it is desirable that some objects out-live sessions, we believe that this facility should be provided by the asynchronous groupware support. At application request time, data storage services should be contacted to save persistent objects on external, global, repositories.

In sections 3, 4, and 5, we describe the design and implementation of a system based on the peer object-group approach. Because we want to maximize flexibility, allow application tools and system functionality to be loaded on-demand, and allow easy support of heterogeneity, the Java language was a natural choice [8]. The integration with the Web was an additional motivation. Our stance is that implementing SGA using our system and the Java language is easy, and realistic applications can still be produced.

3. Object-Oriented Group-Protocols Implementation and Composition

The essential component to realize the peer object-group approach is the group-communication subsystem. This subsystem provides two related sets of group services - group membership services and message passing services. These services are implemented by group-protocols, which are accessed and provided through the combined use of a user-to-protocol service request interface, and a protocol-to-user event notification interface. The user-to-protocol interface includes methods for message sending, multicasting, and group management, and the protocol-to-user interface 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. Raw distributed systems are characterized by a high level of concurrency, unpredictable and independent component failures, and the lack of a global event ordering. Group-protocols make the system appear to behave more orderly, offering a system view where faults are perceived in a consistent manner, message delivery order is agreed by group members, and dynamic membership groups are addressed transparently.

This inherent complexity calls for a modular implementation of group protocols, and object-orientation can be quite useful in this context. Group-protocols can be built as a collection of modules each implementing a specific and simple service. A particular and very natural approach is to link modules in a multi-layer stackable architecture, where upper-layers use and enhance the service provided by lower-layers. Systems like Horus [9], pursue this vein for group-protocols, as well as others in a broader scope [10]. We have also taken this modular approach, and will discuss it below.

3.1 - The Object-Oriented Framework

We have implemented an object-oriented framework in Java specially designed to allow convenient implementation and composition of group-protocols. Protocol layers are implemented as objects of special classes, which implement group service related interfaces. Complete protocol structures (or stacks) are built attaching those objects together. Figure 2 depicts the defined classes and interfaces. The discussion of specific group-protocols to be used as a basic support for SGA will be deferred until section 5.

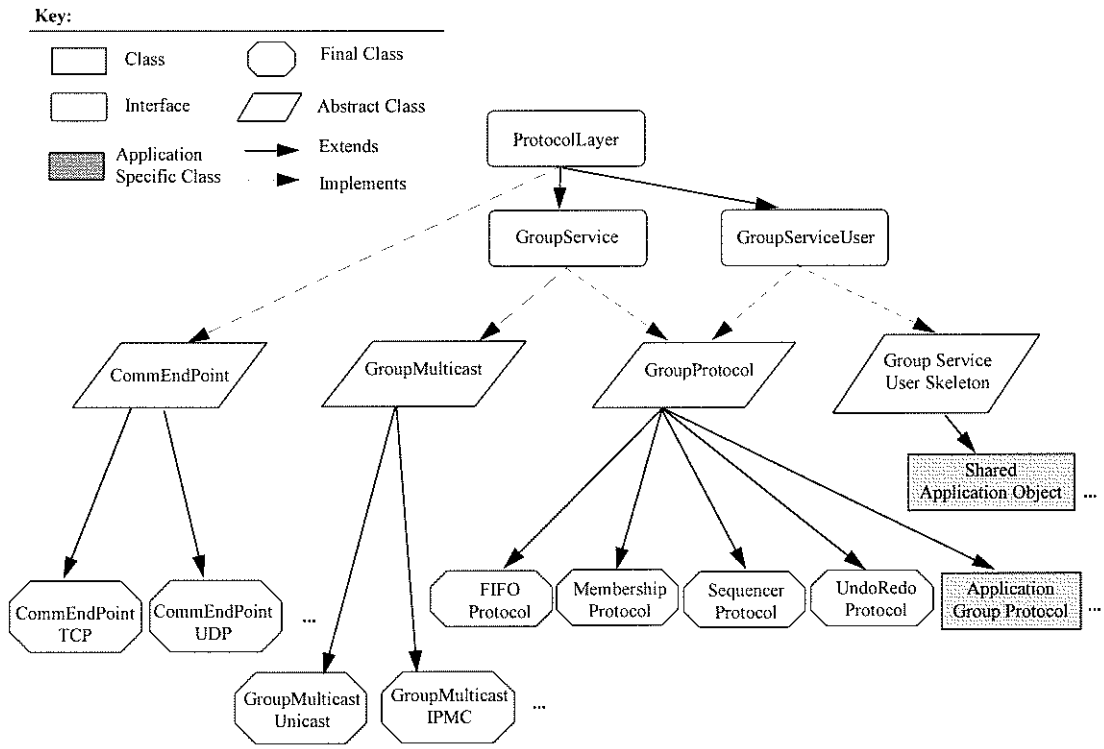

Figure 2. Framework classes and interfaces.

GroupService and *GroupServiceUser* represent two important interfaces. *GroupService* specifies the methods an object implementing a group service should provide. It includes methods for point-to-point message sending and multicasting, group joining, leaving, and creation, among others. They are usually dubed as down-calls. *GroupServiceUser* specifies the methods an object should implement to use a group service. It includes methods for message delivery, group membership changes notification, and state management. They are usually dubed as up-calls. Abstract classes *GroupProtocol*, *GroupMulticast*, and *CommEndPoint*, provide common functionality for objects implementing these interfaces.

GroupProtocol is the typical base class for objects implementing group protocols. Concrete group protocols derive from this class. *GroupMulticast* and *CommEndPoint* derived classes are responsible for providing hooks to native message passing services; message multicast services in the former case (e.g. IP multicast), and point-to-point message exchange in the latter (e.g. UDP).

A complete protocol structure will, typically, be built attaching a *CommEndPoint* derived class for point-to-point communication, a *GroupMulticast* derived class for multicast communication, and several *GroupProtocol* derived classes, each adding its own service semantics to the complete group service. The reference to the top layer object represents the service.

Application objects use a group service by extending the abstract class *GroupServiceUserSkeleton*. This class implements a dummy empty body for the event handling methods of the *GroupServiceUser* interface, and provides some "sugars" for accessing the underlying group service. Derived classes should override the event handling methods to catch relevant events. The group service to use is specified in the constructor method(s).

3.2 - Building Protocol Structures

While it is quite possible to build complete protocol structures "by hand", creating each protocol layer individually and attaching them together, a much more convenient mechanism is available - description strings and protocol structure generators. Protocol structure description strings convey information about which protocols should be used to build a particular protocol structure (together with its parameters), and the topological relationships between layers. Protocol structure generators parse description strings and generate correspondent protocol structure, by dynamically loading layer classes.

Syntacticly, description strings list the class names for the objects implementing specific protocols or layers. Constructors' arguments can be supplied using a familiar notation, and labels can be associated with layers for symbolic reference. Furthermore, aliases can be used to ease in the specification - an alias from a static pool can be used to represent a standard configuration of group protocols. From an application programmers' perspective, each string represents a specific group service semantics and a corresponding object-group consistency criteria. Below we show examples of such strings, and how protocol structures are generated.

```
String psds1=                                          //a typical protocol structure
  "Synchronizer:" +
  "UndoRedo:" +
  "Sequencer:" +
  "Membership_LCS(multicast='multicast', keep_alive_time=10000):" +
  "Fragmentation(MTU=1200):" +
  "FIFOReliable():" +
  "GroupMulticast_Unicast 'multicast'local:" +
  "CommEndPoint_UDP 'endpoint'global(priority=6)";

String psds2= "Synchronizer:alias(TOTAL):alias(COM)";  //the same as psds1, but using aliases
ProtoStructDescr psd= new ProtoStructDescr(psds1);          //create parser/generator
GroupService gs= (GroupService) psd.makeProtoStruct();  //make protocol structure
gs.dump(0, GroupService.RECURSIVE);      //dump to console the layer's textual representation
```

Two description strings are depicted in the example. They represent the same protocol structure, but one is shorten by using aliases. A *ProtoStructDescr* object is used to parse one of the strings and to generate an instance of a protocol structure. A reference to the top layer is returned.

Using description strings, well documented aliases, and supplying the application programmer with an appreciable battery of protocols, it is possible to isolate her/him from the details and complexity of group-communication and still obtain all its benefits.

3.3 - A Simple Example

Below, we show and discuss a simple example of how group services can be used in our Java framework to realize the peer object-group approach. Most code is independent of the particular object in hand or follows a common pattern, so it can be automatically generated by a stub-compiler.

```
import dagora.groups.*;

class MyObjectGroup extends GroupServiceUserSkeleton
{
 static final int METH_add= 1;              //method ID for: value+= arg
 static final int METH_multiply= 2;         //method ID for: value*= arg

 int value;                                 // the shared/replicated object state

 MyObjectGroup(GroupService gs, String name) { super(gs, name); } //constructor
//
// Methods for event handling (override base class methods).
//
 public void evt_getState() {                           //supply local state to new comer
  HeaderByValue hdr= new HeaderByValue();               //create header to hold state
  hdr.writeInt(value);                                  //write shared state
  try putState(hdr);                                    //supply local state
  catch (GroupServiceException e);
 }

 public void evt_putState(ReadableMessage state) {  //state initialization
  if (state==null) value= 0;                                //fresh state
  else value= state.readInt();                              //peer suplied state
 }

 public void evt_viewChange(int modifier, GroupView view, ReadableMessage rmsg) {
  super.evt_viewChange(modifier, view, rmsg);
  System.out.println("current view: " + getView());  //print current view
 }

 public void evt_mcastMessage(ReadableMessage rmsg, CommEndPointID from, boolean flag) {
  int meth_id= rmsg.readUnsignedShort();            //read and apply update
  switch (meth_id) {
   case METH_add:
    xpt_add(rmsg.readInt());
    break;
   case METH_multiply:
    xptd_multiply(rmsg.readInt());
    break;
  }
 }
//
// Exported methods code.
```

```
//
  xptd_add(int arg) { value+= arg; }                    //add argument to shared value

  xptd_multiply(int arg) { value*= arg; }               //multiply shared value by argument
//
// Method stubs and auxiliary update method.
//
  public add(int arg) { update(METH_add, arg); }              //stub for xptd_add()

  public multiply(int arg) { update(METH_multiply, arg); }    //stub for xptd_multiply()

  void update(int meth_id, int arg) throws GroupServiceException  {
    WritableMessage wmsg= new WritableMessage(); //create message
    HeaderByValue hdr= new HeaderByValue();                   //create header
    hdr.writeShort(meth_id);                                  //put data in header
    hdr.writeInt(arg);
    wmsg.addHeader(hdr);                                      //add header to message
    mcast(wmsg, GroupService.SYNC);                           //multicast and wait until applied
  }
//
// Test program.
//
  public static void main(String[] args) throws GroupServiceException, ... {
    GroupService gs= ...                                //protocol structure to use
    GroupID group_id= ...                               //group identifier
    boolean create= ...                                 //create flag
    MyObjectGroup obj= new MyObjectGroup(gs, null);  //create local replica

    if (create) obj.create(group_id);                   //create group
    else obj.join(group_id, null);                      //join group
    obj.add(5);                                         //perform first update
    obj.multiply(7);                                    //perform second update
    obj.leave();                                        //leave group
  }
}
```

In the example we are replicating a shared object that simply has an integer variable as its state, and only exports two methods - one for addition and another for multiplication. The code is in class *MyObjectGroup*.

Local replicas of the shared object extend the abstract class *GroupServiceUserSkeleton*, and override some relevant event handling methods, namely: *evt_mcastMessage()* - to handle multicasted messages, by unmarshaling method identifiers and arguments and applying the corresponding method; *evt_getState()* - to supply the local state for transmission to a newcomer member/replica; *evt_putState()* - to initialize local replica; and, *evt_viewChange()* - to act on membership changes.

Auxiliary method *update()* is used to submit updates to the object-groups. Basically, it creates a message and a message header, marshals the method identifier and argument into the header, inserts

the header in the message, and requests the underlying group service to multicast the message. The specified option is used to block the sender until the update has been locally applied.

Note that while received messages are streams like data types that implement the Java interface *java.io.DataInput*, from which data can be read (*ReadableMessage* class), outgoing messages are only place-holders for header objects (*WritabableMessage* class). Header objects themselves are used to convey data - they can be streams like data types implementing the Java interface *java.io.DataOutput*, to which data can be written (*HeaderByValue* class), or simply data references to byte buffers or some other type of objects (*HeaderByRef* class).

The test program is in *main()*. It creates a local *MyObjectGroup* object, specifying in the constructor the *GroupService* object to use. To make the local object a real replica, the object must join the corresponding (object-)group. This is done by invoking *create()* method - to create the group, or the *join()* method - to join the previously created group. Before any of this methods return, the state-transfer related methods are invoked to initialize the local replica state. Following in the code, two updates to the shared object are performed. Finally, the object leaves the group.

The group service is assumed to have been created using the mechanisms presented in section 3.2. In the next section we will discuss binding issues.

4. Object-Groups Management and Common Services

We have been discussing how the peer object-group design pattern can be used to implement SGA, but some issues remain to be addressed. There are also some specific services that seems to be so recurrent in applications that a common solution may well be justified.

We want to be able to designate sessions and object-groupsby symbolic names, i.e., human readable names. This implies that some binding mechanism must be provided to map symbolic names into the low-level naming and addressing data required by the communication subsystems. Additionally, we want human-users to have an accurate view of the objects currently available in a SGA shared workspace.

A common service required by SGA is that low-level communication endpoint identifiers be associated and mapped to entities representing real human-users. This may be used to display information about the human users currently in a session, or to provide user activity awareness (e.g. the sender endpoint identifier of a message carrying an object update can be mapped into a user differentiating visual mark).

Finally, although we want that (object-)groups membership vary as users working sets vary, we do not want that object-groups to "vanish into thin air" just because no user is currently working with it. Moreover, to make an object-group resilient to K process failures, at least K+1 processes must replicate the object. We need a minimal membership mechanism that precludes object-groups membership to decrease below some programmer specified threshold.

Below, we describe how the above issues are addressed in our system in an integrated manner. Concretely, we describe a distributed-object model that we use to structure and implement SGA.

4.1 - A Distributed-Object Model to Structure and Implement SGA

As discussed in session 2.2, we envision the SGA shared workspace as a collection of peer object-groups, whose state is replicated amongst the user processes accessing an object. We now introduce the notion of fully replicated *Session* object - an object which all user processes participating in a session must replicate. *Session* objects contain the data and implement the services required to solve the common issues identified above. Figure 3 depicts the complete object model.

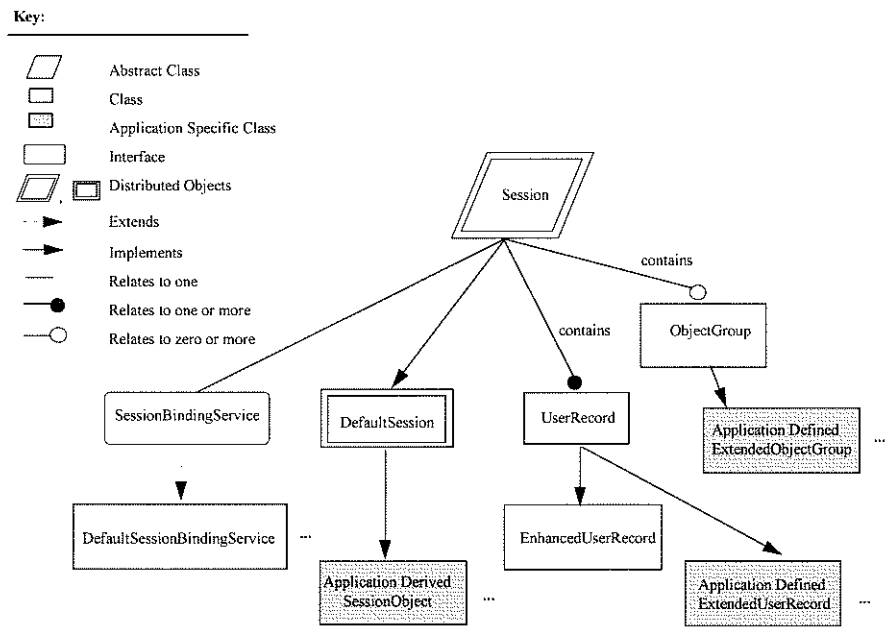

Figure 3. Distributed-objects model.

A *Session* object's main purpose is to store and manage directories (or mappings): an object-group directory, a user directory, and an endpoint-to-user mapping. The object-group directory is used to hold information about the shared-objects currently in the shared workspace. The user directory is used to hold information about the users currently participating in the session. The endpoint-to-user map is used to map endpoint identifiers to users. Both object-groups and users are identified by keys - typically, symbolic names, while endpoint identifiers are low-level globally unique identifiers.

Object-groups are represented by the *ObjectGroup* class. *ObjectGroup* objects contain, in addition to the respective key, relevant binding and management data. The binding data includes the low-level group identifier for the supporting group and its protocol structure description string. The management data includes a minimal membership value for the object-group, some replication options, and some "sugar" fields to be used solely by the application programmer (e.g. object-group creator, class or type of the attached application object, etc.). Application derived classes may store additional information (e.g. access control or coordination information). The object-group directory of the session object is used to map the object-groups keys to the corresponding *ObjectGroup* objects.

Human users are represented by the *UserRecord* class. *UserRecord* objects are expected to contain relevant human readable information about users. This class only contains the user identification key (e.g. the login name), but derived classes may store additional information (e.g. user full name, user photography, e-mail address, the Web homepage, etc.). The user directory of the session object maps the user keys into the information contained in these *UserRecord* objects.

Note that *ObjectGroup* and *UserRecord* objects are non distributed objects, in the sense that they are not replicated; they are immutable local objects, referenced by replicas of Session objects, and passed by value as arguments of several (hidden) methods of Session objects.

The process of binding to sessions is performed with the use of external session binding services. The *SessionBindingService* Java interface defines the methods an object must implement to allow others to access these services. Typically, the objects implementing the interface will be proxies to remote objects, or, in a different perspective, clients to remote services. The particular session binding service to use is specified in the constructor of a *Session* object, although a default service exists. Currently, it is a simple, non-replicated, service that maps symbolic group/session names to group binding data.

When a user wants to join a session, the specified external session binding service is used to get the required binding information, and afterwards the user process joins to the group supporting the respective session object. The *UserRecord* for the local user is automatically registered in the user directory, unless a duplicate key error occurs and the user is forced to leave the session.

When object-groups are created they are registered in the session object. More concretely, a *Session* object can be requested to create an object-group, using the data stored in an *ObjectGroup* object. This will cause the session object to create the supporting group service, to automatically attach the local application object replica to the underlying protocol structure, to create the (object-)group, and, finally, to register the *key-to-ObjectGroup* mapping in the object-group directory. A top-layer wrapper is inserted between the main protocol structure and the application object to capture sensitive *GroupService* interface methods (see section 3.1). In particular, group leaving and terminating methods are intercepted to automatically unregister object-groups and endpoint-to-user mappings, and to

perform minimal membership control, i.e., to check and forbid that the object-group membership decreases below the the specified threshold.

Using object-groups after they have been created is simple. The programmer only has to specify the identification key, and the local object replica to use. The session object will locally fetch binding information, create the appropriate protocol structure, attach the local replica to it, and join the object-group.

It is now clear that we have employed a two-level approach to solve the binding issue. Binding to session objects is performed using external binding services, whereas binding to other shared-objects is performed using the fully replicated directory contained in session objects. This two-level approach is, in our perspective, much more adequate to SGA specificity than one using only external binding services. Conceptually, we should abstract the application as a collection of shared-objects organized around the fully replicated *Session* object. Figure 4 illustrates the idea.

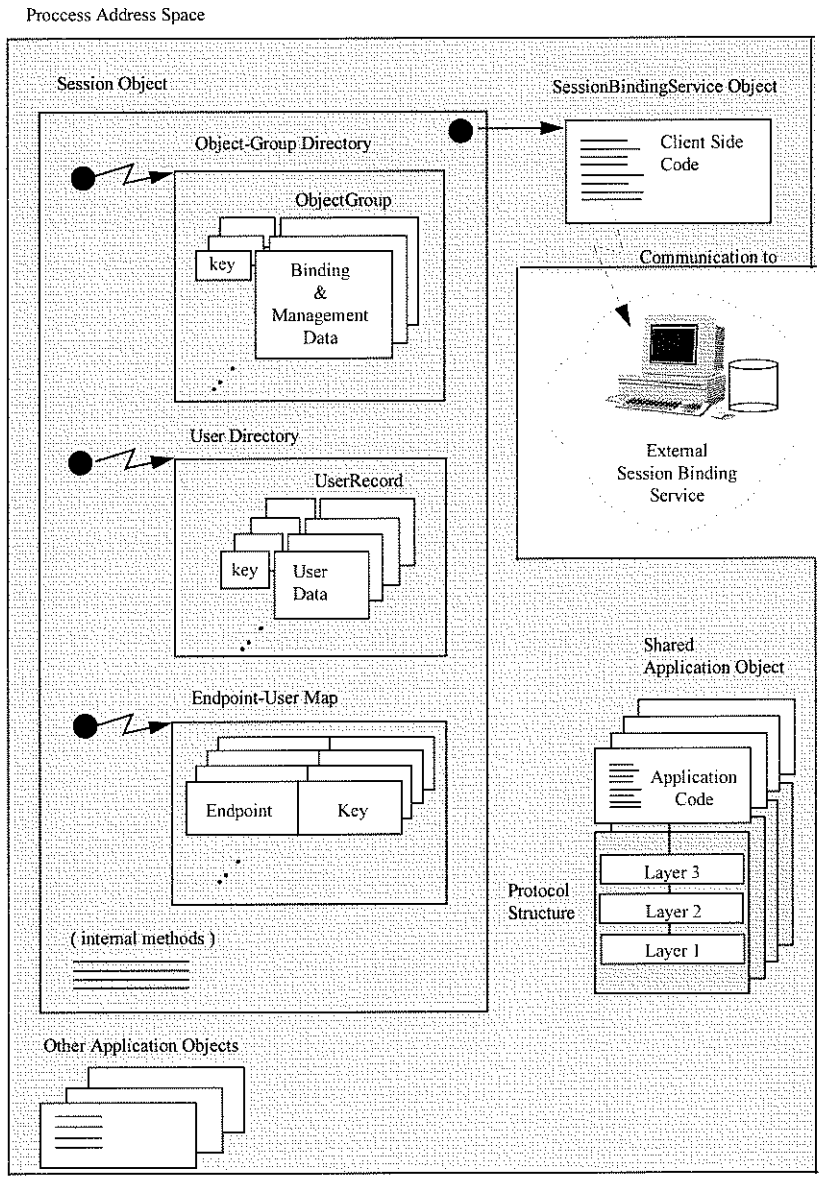

Figure 4. Objects conceptual model.

Finally, a reactive programming style is used to act on session related events. *Session* objects can be derived and event handling methods overriden. There are event handlers which are triggered when a user enters a session, when a user leaves a session, when an object-group is created, destroyed, etc. This can be used as a basic awareness mechanism.

There can be many ways of implementing a *Session* object-like interface; that will depend mainly on the semantics of the underlying group-protocol. We anticipate several variants, but provide a default one - a *DefaultSession* object, currently based on a total order protocol, and a membership and reliable multicast protocol.

4.2 - An Example

Below we sketch an example of how our object model can be used to structure and implement distributed SGA. A *Session* derived object is used to enter and leave a session, to store user records, and to manage the shared-objects, i.e. the object-groups. User records are registered and unregistered automatically as corresponding *Session* objects replicas enter and leave a session. *Object-groups*, on the other hand, are registered/created, used, and destroyed, on application request.

```
import dagora.groups.*;
import dagora.groups.services.*;
import dagora.rmi.*;
import dagora.ogm.*;

class MySession extends DefaultSession
{
 public MySession(UserRecord user, SessionBindingService binder) //constructor
 { super(user, binder); }
//
// Methods for event handling (override base class methods).
//
 void evt_newUser(UserRecord user)                    //event handler for new users
 { System.out.println("A new user in town: " + user); }

 void evt_userLeft(Object user_key)                   //event handler for leaving users
 { System.out.println("User is leaving: " + User(user_key)); }

 void evt_userFailed(Object user_key)                 //event handler for an unreachable user
 { System.out.println("User become unreachable: " + getUser(user_key)); }

 void evt_newObjectGroup(ObjectGroup obj_grp)    //event handler for new object-group
 { System.out.println("Object-group created: " + obj_grp); }

 void evt_objectGroupDestroyed(Object obj_key) //event handler for object-group destroyed
 { System.out.println("Object-group destroyed: " + getObjectGroup(obj_key)); }



//
// Test program - enter a session.
//
 public static void main(String[] args) throws ServiceException, ClassNotFoundException,
 RemoteException,  InvalidArgumentException,  GroupServiceException,  InvalidKeyException,
MalformedURLException
 {
  Object session_name= ...                                    //select session name
  Object user_name= ...                                       //select user name: "Johnny" or "John"?
  SessionBindingService bind_service= new DefaultSessionBindingService(); //binding service to use
  UserRecord user= new EnhancedUserRecord(user_name);  //create user record
  Session session= new MySession(user, bind_service);         //create local session object replica
  session.enterSession(session_name);                         //enter session
 }
}
//
```

```
// Sample class for shared-object (whiteboard).
//
class SharedWhiteBoard extends GroupServiceUserSkeleton
{
  ...                                          //shared object specific code
}
//
// Sample code to handle user request to create shared-object (whiteboard).
//
{
  String obj_key= "Sketching Canvas-1";                    //human-readable object key
  SharedWhiteBoard swb= new SharedWhiteBoard();            //create local replica
  String psds= "Synchronizer:alias(TOTAL):alias(COM)";     //protocol structure to use
  ObjectGroup og= new ObjectGroup(obj1_key, psds);         //create object-group record
  session.registerObjectGroup(og, swb);                    //register object-group
  ...                                                 //save SharedWhiteBoard for latter reference
}
//
// Sample code to handle user request to use shared-object (whiteboard).
//
{
  String obj_key= "Sketching Canvas-1";                    //human-readable object key
  SharedWhiteBoard swb= new SharedWhiteBoard();            //create local replica
  ObjectGroup og= session.useObjectGroup(obj_key, swb);    //use object-group
  ...                                          //save SharedWhiteBoard for latter reference
}
//
// Sample code to handle user request to stop using shared-object (whiteboard).
//
{
   SharedWhiteBoard swb= ...                               //get reference to SharedWhiteBoard
   try swb.leave()                                         //leave object-group
   catch (MinimalMembershipException)
   { ... }                                                 //can't leave
}
//
// Sample code to handle user request to destroy shared-object (whiteboard).
//
{
   SharedWhiteBoard swb= ...                               //get reference to SharedWhiteBoard
   swb.terminate()                                         //terminate object-group
}
//
// Sample code to handle user request to leave session.
//
{ session.leaveSession(); }                                //leave session
```

The example code contains three main code fragments. In the first code fragment, the class *MySession* is defined, and extends the default implementation of session objects. This is used exclusively to catch session related events: users entering or leaving a session, and object-group creation and destruction. Typically, this code will notify the human-user of the respective event. It shows also the code required to enter a session. First, a user identification key (e.g. name), and the external identification of the session to join to (e.g. in URN format), is obtained by some means (e.g. user supplied). Next, a session

binding service is selected - in this case the default was selected, so it could have been omitted, and a *UserRecord* object was created. A library *EnhancedUserRecord* object was used - it reads additional user information from a local file. Last, a session object replica is created (MySession class), and the session is entered (unless an error occurs).

In the second code fragment, the *SharedWhiteBoard* class is defined. It is used to implement a particular shared-object - a shared drawing canvas. It will hold code specific to the shared-object, (in this case, the code for shape drawing and user interaction events handling), and the code required to make the object distributed - as illustrated in the example of session 3.3.

Finally, the last fragment of code contains sample code for how user requests should be processed. To create and register an object-group, an *ObjectGroup* object must be created. The object key and the protocol structure supporting the object-group is the minimal data which must be specified; default values are used for the remaining data. When the object-group is registered, the local object replica is automatically attached to the underlying group service and the group is created.

Using an object-group is simpler. Only the object key and the local object replica must be specified. The required binding information is automatically fetched, the local replica attached to the group service and joined to the corresponding group.

To stop using an object-group is also simple, but care must be taken in order not to violate the minimal membership requirement. Destroying an object-group or leaving a session is trivial.

5 . Group-Protocols to Support SGA

As we have mentioned in the beginning, although we could use generic group-protocols, we believe that reflecting SGA specifics in protocol design brings major benefits.

SGA are interactive applications by nature, so it is required that the system responds and evolves accordingly to users expectations [11]. Users desire short or immediate response times; by preference, similar to single user application. It is not acceptable for a user to wait a considerable time to perform an update on a shared object, or to wait a long time to join or leave the group supporting that object. Users desire short notification times, i.e., they desire to see each others actions as soon as possible, otherwise, user cooperation effectiveness may be injured. Users objects working-sets are expected to change often during the lifetime of a SGA session, and users should be able to enter and leave sessions whenever they please (unless restricted by coordination policy decisions). This calls for group services which support dynamic lightweight group membership changes. Failing to satisfy these requirements, group-protocols may revel to be unsuitable to support SGA.

We are highly engaged in the process of defining new group services semantics and implementing new group-protocols specially tailored for SGA. In the following, we will present some of the protocols we have already implemented. In each case, we will highlight some design decisions closely related to the specifics of SGA.

To deal with network unreliability we have implemented a protocol providing FIFO reliable channels between pairs of communication endpoints. The protocol ensures that point-to-point and multicasted messages delivery order is consistent with the sending order. Furthermore, it implements efficient connection management strategies in order to reduce group joining and leaving latency times. Forcing endpoint identifiers to be unique and non reusable, and properly managing connection records, the protocol is able to perform duplicate free connection establishments, without incurring in the performance penalty of classical techniques (e.g. the three-way handshake connection establishment technique [12]).

To consistently manage group state efficiently, we have devised alternative membership and reliable multicast service semantics [13]. In particular, we have defined a new semantics which is weaker than the "standard" "view synchrony" semantics [14], and can be implemented by protocols which incur in less overhead for group membership management, (although to recover from failures the cost is the same). Still, it is powerful enough to allow other useful protocols to be implemented (e.g. ordering protocols). The semantics precludes group views from re-merging, which avoids object's state to be re-merged after processes become partitioned. Mechanisms to perform state merging are assumed to be provided by external data storage services, as discussed in section 2.

For total message ordering and object state transfer we have implemented a sequencer based protocol [15]. This approach was used because at low service load - the expected scenario in SGA, it is the one with which better latency measures can be obtained [16]. Only a point-to-point and a multicast message transmission and processing time are required to apply an update.

We have also experienced with the undo/redo actions paradigm [17]. We have implemented a protocol, in which the commutative and masking properties of object methods are explored to diminish update latency times, by using an optimistic ordering technique. When the optimistic ordering is discovered not to be semantically equivalent to the underlying total order, updates are undelivered and delivered again in the proper order.

Other miscellaneous protocols/layers were implemented for additional services (e.g. one to get the local reply of a shared method invocation, one to fragment messages, etc.). Figure 5 illustrates examples of how they can be stacked on each other to materialize the support for object-groups.

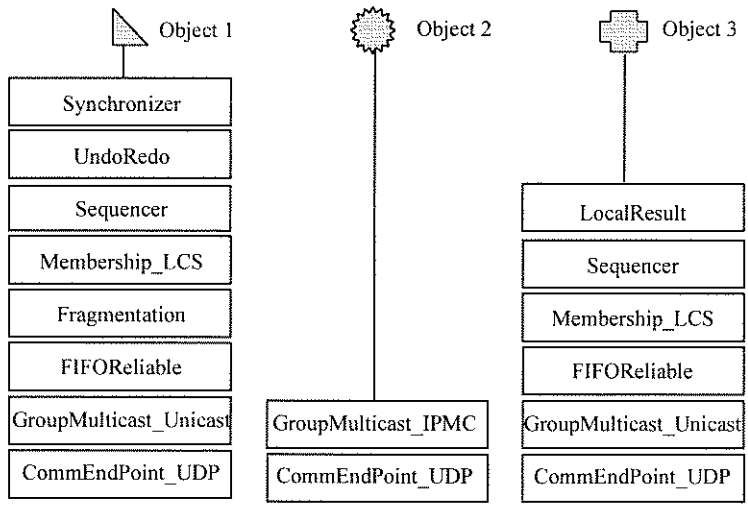

Figure 5. Stacking protocol layers to support object-group.

6 . Experience and Future Work

As has been described, we have developed, up to this point, the protocol composition framework, the object-groups management framework, and a set of stackable group-protocols. We have also tested the suitability of our ideas implementing a demo white-board tool. It is a simple tool which manages a shared drawing canvas. It requires only one object-group to be implemented.

The drawing tool 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. Still, additional experience is required to analyze system behavior in more general environments.

Many work directions where 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 a similar view 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, and hope to validate more clearly the usefulness of the abstractions presented in the paper. We will consider enhancing our object model with additional structuring abstractions and common services as more experience is gained. Still, we

think that the model is so simple and flexible that the provision of most additional functionality can be obtained with the use of reusable generic tools.

Finally, we intend to build a stub-compiler to simplify the task of shared objects programming.

7. Related Work

Our approach to support SGA is based on a fully-replicated architecture - data and application code replicated, and is very broad in scope extending several closely related areas.

Horus [4], and the x-Kernel [10], are two well known frameworks for protocol implementation and composition. While x-Kernel is intend for generic protocols, Horus was specially designed to support group-protocols. Our framework borrows many ideas from these systems, but we recast all the required abstractions to a fully object-oriented setting.

Many generic group-protocols have been proposed in the literature, but only a few were specially designed for SGA. In [17], and in [18], the authors describe specific group-protocols based on optimistic event ordering techniques to reduce updates latencies. In [19], the author describes multi-group semantics and sequencer based protocols to support groupware applications. In all these cases, fault-tolerance is not addressed. Our approach is much more flexible, in the sense that several protocol functionalities are provided, and can be selected, composed, and used as and only when necessary. This allows the system to incur in the less minimal possible overhead for a given consistency criteria. Moreover, in the layers in which the issue is relevant, fault-tolerance is addressed.

Several object-group model adhering systems have been devised - each taking a somewhat different view of what an object-group is. In [20], the author surveys a few - Electra, SOS, Orca, ANSA, and RDO. The adoption of these systems and object-group oriented design patterns seems to be very promising in the establishment of efficient programming environments and frameworks for collaborative applications development. However, there is not yet relevant experience about it. Our work is a contribution in this direction.

8. Conclusions

In this paper we argue that the peer object-group design pattern is a suitable architectural system solution to structure and implement SGA. It offers a natural programming model, has the potential to be flexible and scalable, and can render better performance measures than the client-server or the replicate-server approaches. Although its realization can be complex, the provision of an appropriated group-communication subsystem can hide the application programmer from the low-level details of distributed systems programming. Certain services and functionality is so recurrent in SGA that common solutions are well justified. Providing a common structuring and programming model with

carefully chosen abstractions can be a significant help in the development process. Finally, we conclude that many system design options should be recast to the specific context of SGA. In particular, group-protocols should be specially redesigned to explore and satisfy the specifics of SGA.

Bibliographic References

[1] Kenneth P. Birman, "The Process Group approach to reliable Distributed Computing", Communication of the ACM, Vol. 36, n. 12, December 1993.

[2] Silvano Maffeis, "The Object Group Design Pattern", Olsen&Associates, Zurich, Switzerland

[3] Kenneth P. Birman, and Thomas A. Joseph, "Reliable Communication in the presence of failures", ACM Transaction on Computer System 5(1):47-76, February 1987.

[4] van Renesse, Robbert, Birman, Kenneth P., Fridman, 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.

[5] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, P. Ciarfella, "The Totem Single-Ring Ordering and Membersip Protocol", University of California, Santa Barbara

[6] Silvano Maffeis "Run-time Support for Object-Oriented Distributed Programing", Ph.D. Thesis, February 1995

[7] Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki, "Transis: A Communication Subsystem for High Availability", The Hebrew University of Jerusalem, Israel.

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

[9] van Renesse, Robbert, "The Horus System Specification", Cornell University, March 3, 1995.

[10] Hutchinson, C. Norman, and, Peterson, Larry L., "The x-Kernel: An Architecture for implementing Network Protocols", IEEE Transaction on Software Engineering, Jan. 1991.

[11] 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.

[12] C.A. Sunshine, and Y.K. Dalal, "Connection management in transport protocols", Computer Networks, vol. 2, 454-473, 1978.

[13] Hiltunen, Matti A., and Schlichting, Richard D., "Understanding Membership", TR 95-07, Department of Computer Science, the University of Arizona, July 19, 1995.

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

[15] Kaashoek, M.Frans, and, Tanenbaum, S. Andrew, "Efficient Reliable Group Communication for Distributed Systems" Dept. of Math and Computer Science, Vrije Universiteit, 1993.

[16] Flaviu Cristian, Richard de Beijer and Shivakant Mishra, "Comparing How Well Asynchronous Atomic Broadcast Protocols Perform", Distributed Systems Engineering Journal, Vol. 1, No. 4, 1994, pp 177-201.

[17] 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.

[18] C.A. Ellis, S.J. Gibbs, "Concurrency control in Groupware Systems", in proceedings of the ACM SIGMOD'89 Conference on Management of Data, May 1989.

[19] Tim Kindberg, "A Sequencing Service for Group Communication", Dept. of Computer Science, Queen Mary & Westfield College, University of London, Septmenber 1995.

[20] Richard Achmatowicz, "Object Groups For Groupware Appliction: Application Requirements and Design Issues", Dept. of Computer Science, Queen Mary & Westfield College, University of London, September 1994.