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

Technical Report

DI-UNL-01/97

An Object-Oriented Framework for Efficient Protocol Composition

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**: Communication Protocols, Distributed Systems, Object-Oriented Programming and Protocol Composition, Fault-Tolerance, Group-Oriented Reliable Communication*

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.

An Object-Oriented Framework for Efficient Protocol Composition

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)

Abstract:

In this report we describe a technique for efficient, modular, protocol composition in which event passing between protocol layers is performed using synchronous procedure invocation, and thread creation is minimal. We also present an object-oriented framework for group-protocol composition which relies on this technique, and comment on our early experience in using it to implement group-protocols to support synchronous groupware applications.

1. Introduction

Modular protocol writing and composition has been identified as an essential methodological technique to deal with the increasing complexity of distributed systems and the required protocols. This is particularly true when we address issues of fault-tolerance, multiple participants (group protocols), and/or large-scale networks settings, since semantics options to be provided by the protocols can vary over a broad spectrum. Moreover, it's highly desirable to allow for application programmers to compose in user space, by the extensive use of micro-protocols layering, protocols with the semantics that better suits their needs. Monolithic implementations where a small, fixed number of protocols are available, like in a traditional Unix kernels, or systems like Amoeba, where a single group protocol is available [1], are not satisfying since it's not possible or realistic to have a single program module providing all the required services.

In this vein, several systems have implemented generic frameworks for protocol composition which strive to simplify and promote the task of modular writing of communication protocols for distributed

systems. In [2] the x-Kernel framework for generic protocol composition is described, and in [3] the authors describe the Horus framework for composition of group protocols in user space.

Unfortunately, flexibility and easy reconfiguration in the writing and use of communication protocols can be in opposition with efficiency. New techniques must be devised to fully realize this new trend, while still retaining the efficiency of more traditional approaches. In particular, efficient techniques for event scheduling between protocol layers, and message editing, must be provided. Traditional schemes for event delivery like event queues, or dynamic thread creation, and extensive use of message data copies between layers, are no longer appropriated. On the other hand, the goal of modularity and extensibility should not be compromised.

In this paper, after giving an overview of our object-oriented approach to protocol composition, we describe a technique for efficient protocol composition in which inter-layer event scheduling overhead is minimal, allowing for the creation of (micro) protocol structures with many layers, without compromising the overall efficiency, and still retain the desired modularity. We then briefly present an object-oriented framework for group-protocols composition, which we have implemented in Java, and comment on our experience in using it to write fault-tolerant group-protocols to support a distributed object model specially tailored for synchronous groupware applications. Finally, we conclude the paper pointing directions for future work.

2.Object-Oriented Protocol Composition

A complete protocol service is implemented through the composition of the services provided by a linked set of (micro) protocol objects or layers. Each one of those (micro) protocol objects implements a simple protocol and relies on the services provided by the layer(s) below. It encapsulates the data which describes the protocol and service current state, and provides two standard interfaces. A standard upper interface for service provision to an upper layer, and a standard lower interface describing the events which lower service providers deliver. To those sets of linked (micro) protocol objects we call protocol structures. An application object attaches itself to the top of a protocol structure.

Different protocol structures may implement the same or different protocol service semantics, and map, typically, to different application objects. Most times, protocol structures have an almost stack

like topology minimizing service multiplexing. The enumeration of the layers that compose a protocol structure, together with its parameterization, and the overall topology, constitutes the protocol structure description. This description is included in the binding data of application objects and works like a template that allows processes to make a local instance of the protocol structure. Figure 1 sketch the idea.

This kind of model has been pursued by several systems, being the most notorious example the Horus system [4], from which we have borrowed many ideas.

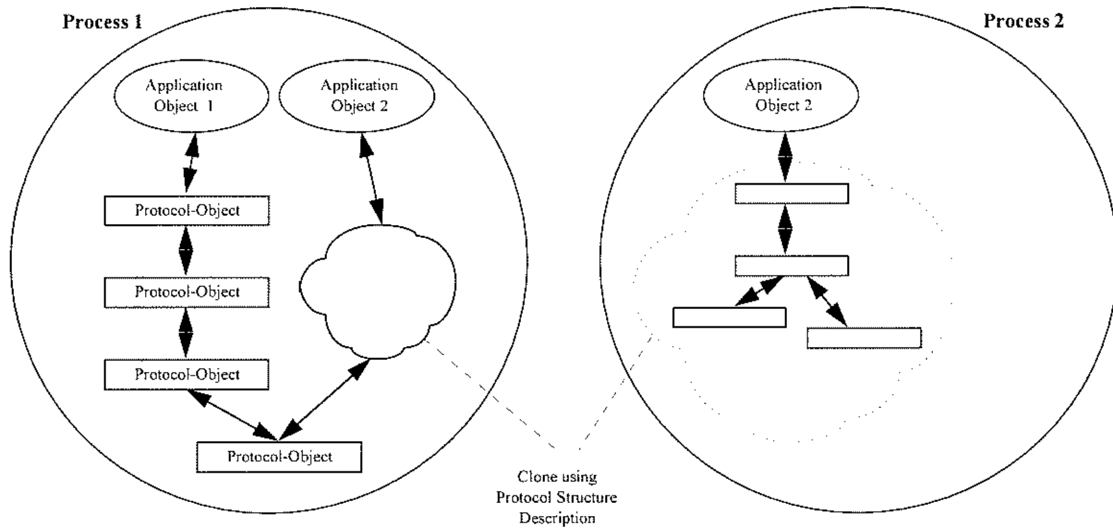


Figure 1 - Protocol structures and application objects.

3. Efficient Protocol Composition

As we have noticed in the beginning, efficient techniques must be used to fully realize the above model. In this section we tackle the event scheduling issue.

Our technique for event scheduling was greatly influenced by the early suggestions and experiences described in [5]. The author proposes that protocol layers and execution contexts (thread) be made independent, and event delivery between layers be made using normal synchronous (local) procedure invocation. Threads work like vertical stripes which cross the horizontal protocol layers (called by the author "Multi-task Modules"). Yet, many design options were left open. We proceed detailing a simple scheme in which efficient options are taken without compromising modularity or clearness.

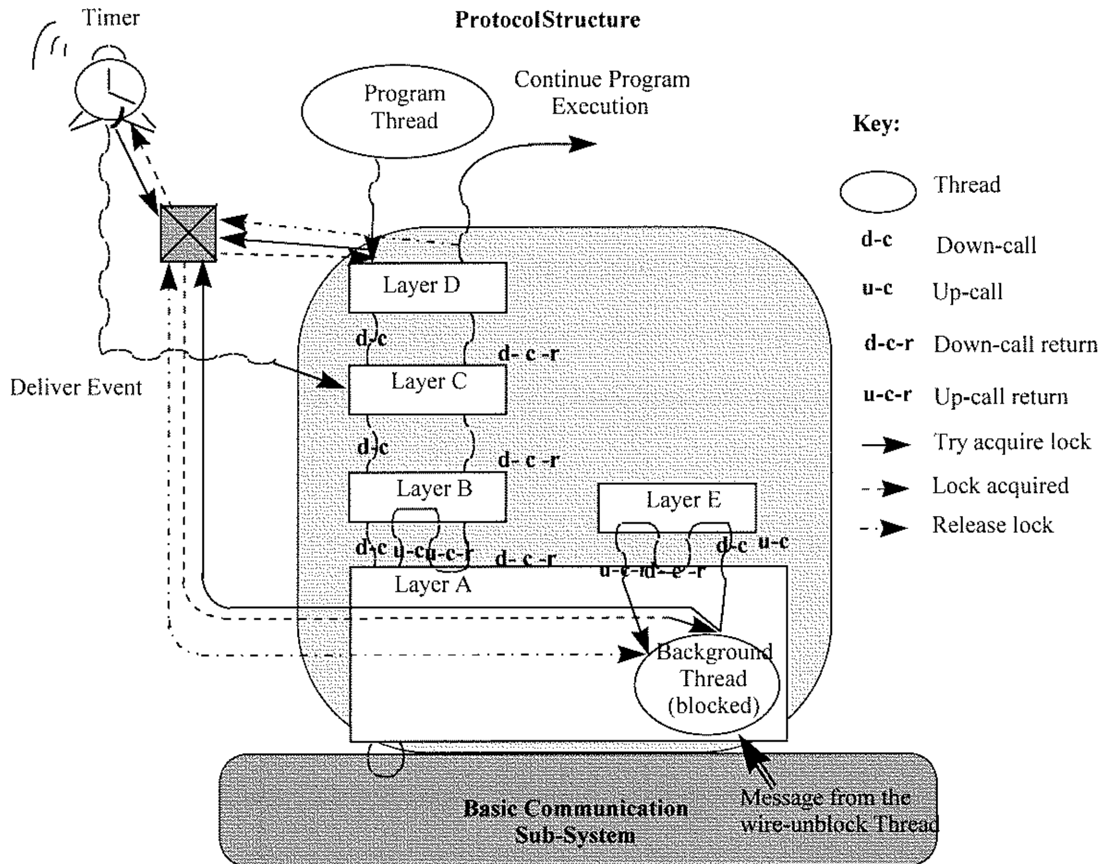
In our scheme we take Clark's suggestion to the extreme, with all event delivery and service requests being performed using synchronous procedure/method invocation - no exceptions, so that a single thread of control may cross many or all the layers. We assume, or enforce, in the service specification, that all down-calls and up-calls are not blocking, meaning that the caller will "quickly" return control without the risk of being suspended because a upper or lower layer is waiting for some external event. This is required to avoid layers from becoming blocked after a first up-call/down-call to another layer, while some work remains to be done. When services can't be implemented this way, we associate (at least) two events with the service provision - a service request event (typically a down-call), and a service completion event (typically an up-call) (e.g. when a service implementation requires several message exchanges between peers).

Using this scheme we require minimal dynamic thread creation. To see this, we can conceptually think that events arise from three sources: upper user layers, lower service provider layers, and internal events, i.e., time triggered events. Upper user layers events and lower layer events are themselves consequence of other events, meaning that in the end case there are only three events generators - requests for service from the user program, message deliveries from the lower communication layer, and timers. Only one thread for each of those ultimate event generation sources needs to exist in the process space.

Still, two major issues need to be addressed to put the scheme to work: deadlock avoidance, and layer consistency. Deadlock may occur, unless something is done, when two threads try to enter two different layers from "different directions", and each other has previously acquired the layer synchronization lock that the other needs. The layer consistency problem is a single thread issue, and is raised when a layer after making an up-call or down-call is called back, and ends up in an inconsistent state when processing of the first event resumes. We will describe next the techniques we use to solve those problems.

To solve the deadlock problem we associated with each protocol structure a synchronization object which is common to all the layers. Any thread that wishes to enter the protocol structure for service request or event delivery must first lock the synchronization object, and only after may proceed with his work. This synchronization policy enforces the serialization of all accesses to the modules

belonging to the protocol structure and making thread deadlocks impossible. Figure 2 ilustrates the idea.



Although resembling the highly optimized schemes for event delivery in some monolithic implementations, we retain the desired modularity in layer writing, since no coupling between any modules have to exist.

Figure 2 - Event scheduling and synchronization scheme.

To solve the consistency problem two techniques are used. First, notice that the problem does not occur so often. Only when the module state is changed in such a way, before a initial up-call/down-call returns, that the remaining processing of the event is erroneously affected. This reminded us of a simple and practical solution to the problem - when in peril of the state being erroneously affected, make the up-call/down-call be the last relevant instruction in event processing code. We dub this technique tail event delivery, and, as can easily be seen, it solves the problem. When several up-calls/down-calls must be performed to complete the processing of an initial event, simple tail event delivery is not, obviously, enough. For those cases, we can use a local variable (i.e. one which lives on a thread's execution stack), which is a queue where pending up-call/down-call are saved (call-queue).

Tail event delivery now applies to all the calls saved in the call-queue, which are delivered in (apparent) succession.

As a final issue, we notice that many times application programmers tend to prefer synchronous and easy to use interfaces to communication protocols. As such, we anticipate the need to use a top layer, or front-end to the protocol structure, which converts the asynchronous interface required by our scheme into a synchronous one. The typical options are available to implement this layer, and for the scheme used by it to deliver events to the application's layer.

To make this somewhat dense description easier to understand, we depict in Figure 3 a simple example of a request-reply protocol which uses our event scheduling scheme. The main program thread in the sender side acquires the synchronization object before entering the protocol structure, and then crosses all layers until it requests the system for message delivery. Only then control returns to the application code. On the receiver side, the background thread that is blocked most of time waiting for messages to arrive from the network, unblocks, and tries to acquire the synchronization object for its protocol structure. After the lock being acquired, the thread raises trough the protocol structure until it delivers message to application code, which them replies using the same thread. Notice the need for the synchronization object due to potential threads coming from above. Later, the sender receives the reply with a similar scheme being used. When there's a need for message retransmission, the timer must also acquire the synchronization object for the all protocol structure at the sender side.

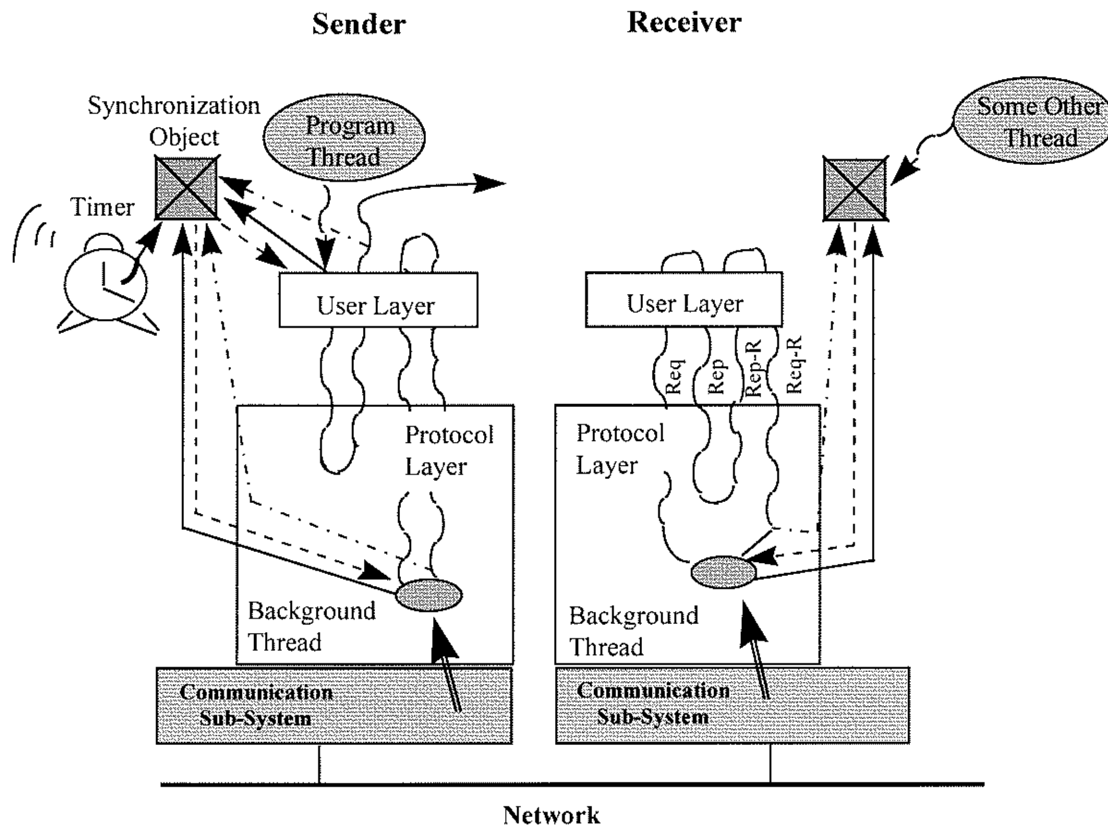


Figure 3 - A simple example with a request-reply protocol.

Although resembling the highly optimized schemes for event delivery in some monolithic implementations, we retain the desired modularity in layer writing, since no coupling between any modules have to exist.

4. An Object-Oriented Framework for Group Protocol Composition

We now briefly describe an object-oriented framework that we have implemented for group-protocol composition, which uses the techniques presented before. Our motivation in implementing such a framework was to allow for the implementation of group-protocols to support a distributed objects model specially tailored to simplify the writing of groupware applications. The language of choice was Java [6], since being an elegant, object-oriented, multi-threaded, portable, dynamic language, with standard APIs, was the one which better suited the requirements for our work. Yet, due to the potential performance problems we were forced to take special precautions, and, in fact, that was the driving force that lead us to develop our techniques for protocol composition.

A few base Java classes and interfaces[1] in the framework allow for the implementation of complete (micro) protocol structures with complex group semantics. Figure 4 show a diagram depicting the relationships between those classes and interfaces.

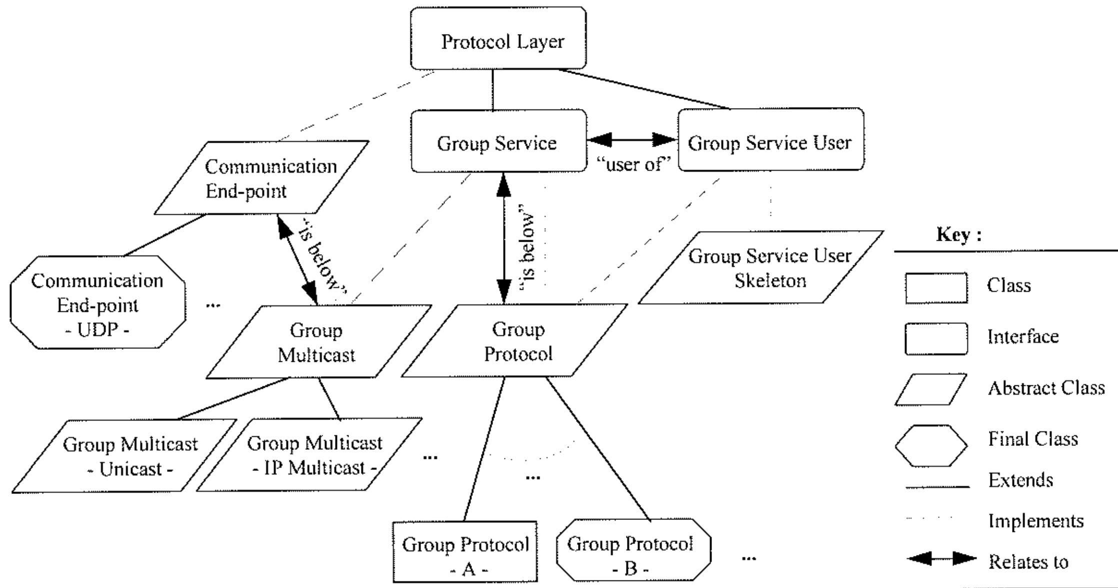


Figure 4 - Framework Classes and Interfaces.

A "Protocol Layer" interface must be implemented by all types of protocol layers, and defines methods pertaining to the synchronization strategy describe in session 3. One of these methods (setSyncObj), is used at bootstrap time to inform the top protocol layer of a protocol structure which synchronization object to use. The top protocol layer will on his hand, propagate that information to the layers attached below him, and so on - spilling the synchronization object identity to the all protocol structure. From then on, all threads entering the protocol structure should (try) lock the synchronization object, and after his work is done they should release the lock. Even timer objects are made aware of those synchronization requirements, and lock/unlock the synchronization object given to them at creation time when delivering the time expiration event.

[1] A Java interface is a data type which defines a set of methods, and their signatures, without providing a body for any of them, i.e. something like an abstract class with all methods abstracted and no data fields.

In our framework we identify two types of protocol services: connection-less point-to-point protocols, and group-protocol services. An abstract class for "Communication End-Points" defines the interface to the former, and an abstract class for "Group Services" defines the interface to the later. We will particularly focus on the later, since implementations of the former map directly to widely available protocols (e.g. UDP/IP). Still, notice that a layer providing a simple front-end to a connection-less point-to-point protocol may, and probably should, perform multiplexing since it substantially reduces the overhead of interacting with the underlying communication system. Moreover, a single background thread may be shared by all upper protocol structures, reducing resource usage. We have taken this in consideration.

A "Group Service" defines the common interface to be provided by a layer implementing a group protocol, and a "Group Service User" defines the common interface to be provided by a layer which uses a "Group Service". These two standard interfaces greatly simplify protocol composition from the linguistic point of view. The semantics of the services provided by different implementations of "Group Services" may, of course, vary.

"Group Service" methods can be roughly divided in five sets (Table 1): attachment related methods, group management, message sending, low level membership and reconfiguration management, and miscellaneous. Attachment related methods allow for (micro) protocol objects to attach themselves to each other, and consolidate the desired protocol structure. A full protocol description mechanism exists also, which allows for simple instantiation of a complete protocol structure given is description. Group management methods pertain to the activities of creating, joining, leaving, or terminating a group. Binding issues are deferred to the application, since is the place where it can better be solved, i.e., it's the application that must provide the mapping between some form of high-level names, and the low-level unique identifiers (UIDs) required by group management services. Message sending services allow for group multicasting, and point-to-point communication between group members. Ordering guarantees depend, of course, of concrete implementations. Low level membership and reconfiguration management methods define methods for view installation, and other, required by some protocols. Finally, the last set of methods allows for group members role setting and management and proving hints to layers about service usage and message traffic patterns.

Attachment related methods	
attach	attach to GroupService
detach	detach from GroupService
getCommEndPointID	get my end-point ID
Group management	
join	join to group
getGroupID	get group UID
leave	leave group
terminate	terminate group
exclude	exclude member from group
Message sending	
mcast	multicast message to group
send	send message to member
Low level membership and reconfiguration management	
installView	set current membership
getView	get current membership
Miscellaneous	
locus	hint about message traffic pattern
unlocus	hint about message traffic pattern
setRole	set member role
dump	dump debug information

Table 1- Methods of the GroupService interface (down-calls).

The “Group Service User” interface defines the events which must be handled by users of “Group Services”, and can also be roughly divided in five sets (Table 2): message delivery, view changes, state management, operation completion notices, and problem reporting. Message delivery events are used to deliver messages multicasted to the group, or sent to this particular member. View change events report changes in the membership of the group, together with the reason for such change. State management events are used when layers implement some object-group consistency protocol. Operation completion events arise from our restriction of not allowing a layer to became blocked because some other layer is waiting for an external event to occur (e.g. completion of join and leave operations are reported this way). Finally, problem reporting events are used to report to upper layers problems that lower layers can’t solve.

Message delivery	
evt_mcastMessage	deliver multicasted message
evt_sentMessage	deliver point-to-point message
View changes	
evt_viewChange	a change on group membership has occurred
State management	
evt_getState	get object state for transfer
evt_putState	put new object state
Operation completion notices	
evt_operationCompleted	operation completed
Problem reporting.	
evt_problem	report some problem

Table 2- Methods of the GroupServiceUser interface (up-calls).

Concrete group protocol implementations are expected to be supplied in foreign Java packages[2]. Still, some effort was done to simplify the task of writing protocols, by providing some structuring abstract classes. An abstract class "Group Multicast" is provided, together with concrete implementations, which implement or offer a front-end for message diffusion protocols. For the typical case where a protocol structure has a stack like topology, we provided a "Group Protocol" abstract class which offers a "Group Service" to a single user and relies also, on the service provided by another single "Group Service". Concrete group protocols will extend "Group Protocol" as a class or a final class[3]. Finally, for the case where a "Group Service User" wants to catch only some events, we provide a "Group Service User Skeleton" which provides null or dummy code for event handling. Concrete users can then extend this class, and catch only the relevant or desired events.

As expected, efficient message editing operands for minimal data copies has deserved our attention, and are supported in the framework. Two types of messages are supplied: writable messages, and readable messages. Writable messages are structured objects in which headers can efficiently be pushed. Header objects is where data is written. Writable messages are linearized only when message

[2] A Java package it's a scope unit which aggregates several public or private classes and/or interfaces.

[3] A Java final class is a class which can not be extended any more.

sending is requested to the underlying basic communication system and the message goes to the wire. Readable messages, on the other hand, are to its interface users raw objects. They provide methods for data reading only, although they can be created from writable message without data copies, making the original message read-only. Notice the asymmetry: for message editing headers are the objects in which data is written; for message consumption is the message object itself which is read for data items.

5. Experience in Using the Framework

As mentioned, our prime motivation to build a framework for group protocol composition was to support a distributed objects model with the purpose of simplifying the writing of multi-user synchronous, or "same-time", groupware applications. This fact has deeply influenced the design of the protocols.

In our distributed objects model, we identify two essential object's types: session objects, and application objects. Session objects are fully replicated objects, i.e. all participants in a work session have a local replica of the object state. On the other hand, application objects are selectively replicated, i.e. only participants actually working with the object, or interested on others work, have a local replica. In this sense, session objects define the maximal membership of the entities manipulating an object. Session object's main purpose is to provide a directory service to application objects. Application objects are all other objects required by the application, which must be made accessible to several participants. Both kinds of objects, use the services provided by some group protocol structure, in order to make its state became shared in a replicated, consistent, manner. In addition to this two types of objects, two more exist: session binding protocol objects and storing protocol objects. The former are used to bootstrap a session, and allow processes to obtain the required binding information. The later are used to store and fetch object's persistent state. Figure 5 depicts those objects and relationships. Our object model and overall architectural approach to collaborative applications structuring is better described elsewhere [7]. We proceed here, enumerating identified requirements for group protocols, and briefly describing the protocols we have written.

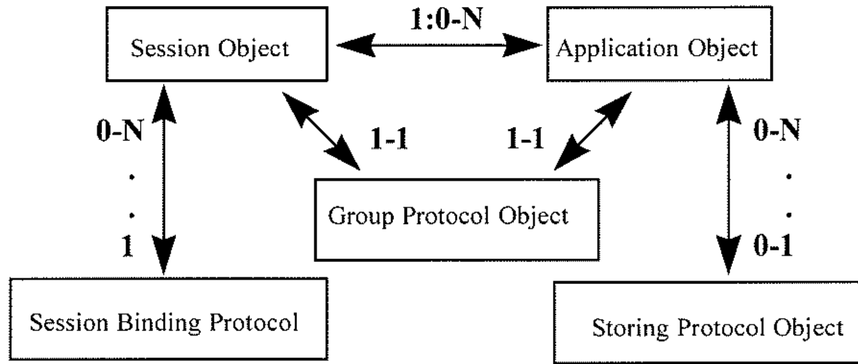


Figure 5- Object Model.

Synchronous groupware applications are interactive application by nature, and this poses special requirements to group communication and object consistency protocols not necessarily found in other settings. End users tend to desire short or immediate response times when performing operations on objects, even if objects are shared, distributed, and replicated. Users desire short notification times, and expect to see each others actions as soon as possible. Users objects working sets are expected to change during the lifetime of a synchronous groupware session, this imply that efficient group membership view management protocols must be used. Failing to satisfy those requirements, the group protocols which support application's objects may revel to be unsuitable. In particular, protocols which incur in high latency for normal operation, or require complex and time consuming message exchanges when in reconfiguration states are to be avoided.

Taking those requirements and caveats in consideration, we have implemented several group protocols each providing a different service semantics. When relevant, we have isolated the dynamic nature of groups from the fault-tolerance requirements, allowing different (sub-)protocols to be used to deal in different situations (e.g. in managing group views we have clearly separate the fault-free scenario where process only join, and leave the group, from the fault-present scenario where process fail or become unreachable). Most protocols we have built are stacked and attached together, although some may be, and in fact are, used stand-alone.

We have implemented a protocol to manage a local view of FIFO reliable channels. The issue of network buffering was addressed, since we hope to be able to use our protocols in large Internetwork environments. Efficient connection establishment and graceful connection release was used. A

protocol for view management, with view agreement and adequate total order [8], which relies on the service provided by the FIFO reliable protocol, was also implemented. Group view and state merging was purposely neglected, since it highly increases application's complexity, and it's avoidable given our overall architecture where synchronous and asynchronous collaborative interaction is integrated [7]. A protocol for fast message total ordering was also implemented. It's a sequencer style protocol, but where the sequencer can change on application's request. This protocol relies on the view agreement and order protocol. A stand-alone protocol for FIFO unreliable channel was also implemented, and is expected that other lightweight protocols came to mind. Finally, a layer to provide a more synchronous interface to the protocol services was implemented.

6. Conclusion and Future Work

It's our believe that we have in fact built a valid, elegant, and useful framework for implementation and composition of group protocols. Although, we have not yet performance measures, we expect values to be appealing, in spite of biased by the interpreted characteristic of the Java language. In practice, although not heavily tested, the system tend to exhibit an acceptable behavior, given that the enumerated requirements for synchronous groupware application are met.

We plan to implement more group protocols in a short term, namely, protocols to implement appropriated distributed lock models, and experiment with a message delivery paradigm where message delivery can be undone and later redone, for ordering purposes [9]. This paradigm will be provided with the purpose of reducing updates latency.

7. References

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

[2] Hutchinson, C. Norman, and, Peterson, Larry L., "The x-Kernel: An Architecture for implementing Network Protocols".

[3] 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.

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

[5] Clark, David D., "The Structuring of Systems Using Upcalls", in proceedings of *10th Symposion on Operating Systems Principles*, 1985.

[6] Gosling, James, McGilton, Henry, "The Java™ Language Environment: A White Paper", Sun Microsystems, 1995.

[7] Domingos, Henrique J., Martins, J. Legatheaux, Simão, Jorge P., "A Flexible Object-Group-Oriented Framework to support Large Scale Collaborative Applications", to apper in proceedings of *30th Hawaii International Conference on System Science.*

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

[9] 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.