Light-Weight Groups : an implementation in Ensemble*

Alexandre Pinto
Universidade de Lisboa
apinto@di.fc.ul.pt

Hugo Miranda
Universidade de Lisboa
hmiranda@di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Abstract

This paper reports a practical experience of implementing a group communication service using a protocol composition framework. Specifically, the paper describes the implementation of a *Light-Weight Groups* (LWG) service on the *Ensemble* system. This is an interesting case-study because the LWG service is very demanding in terms of protocol composition since its implementation requires the use of forked shaped protocol stacks that are reconfigured at run-time. The insights that we have gained with the exercise are being used in the development of a new composition framework called *Appia*.

1 Introduction

The *Group Communication* paradigm is a very useful tool for the development of distributed applications [2, 3]. Implementations of the paradigm use a combination of different protocols such as membership protocols, reliable dissemination protocols, causal and total ordering protocols, message stability protocols, etc. To simplify the development of these protocols a number of protocol composition frameworks have been developed, such as x-kernel [6], Horus [10], Coyote [1] or Ensemble [5], to name a few.

This paper describes the implementation of a specific layer of a group communication service, the *Light-Weight Groups* service [4] using a concrete protocol composition framework, the Ensemble [5] system. A first version of this protocol has been implemented in the Horus system. In this second implementation we extended the code to cope with network partitions, a feature that was not considered in the original design [4] but that was later added to the protocol [9]. Therefore the purpose of the experience was twofold:

- On one side, we wanted to evaluate the difficulties in implementing the partitioned operation extensions to the LWG protocol.
- On the other hand, we wanted to evaluate the flexibility of the Ensemble framework, given that the LWG protocol is extremely demanding in terms of the flexibility supported by the composition and execution kernel, due to its unusual stack and dynamic reconfiguration requirements.

The paper discusses the several lessons that we have learned with this experimental work and that we are now applying in the development of a new protocol kernel called *Appia* [8].

The remaining of the paper is structured as follows. In Section 2 we describe briefly the Light-Weight Groups service. Then in Section 3 we describe the Ensemble system and in Section 4 we describe the actual implementation, and the decisions made, to put the two together. Section 5 discusses the insights gained from the implementation and Section 6 concludes the paper.

2 Light-Weight Groups

This section provides a brief introduction to the Light-Weight Groups service. For more complete description the reader is referred to [4] and [9].

The goal of the Light-Weight Groups service is to improve the efficiency and performance of group communication platforms in scenarios where applications are required to use many dynamic groups that may overlap. The LWG service dynamically keeps track of the membership of the groups used by the application and, when these exhibit a significant overlap, attempts are made to promote resource sharing by using common resources, such a

*This work was supporrted by Paxis/C/EEI/12202/1998, TOPCOM

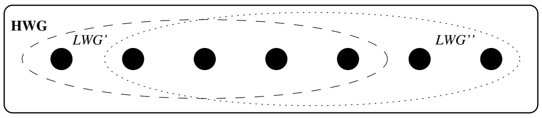

Figure 1: Two LWGs mapped on a HWG

failure detectors or IP multicast addresses to support many groups. The service can be implemented as a layer that maps multiple user groups, or *Light-Weight Groups* (LWGs), to a single low level group, or *Heavy-Weight Group* (HWG) (Figure 1).

One possible example is an on-line auction service, where to each item being auctioned corresponds a group. Because usually buyers bid for several items, they belong to several groups. Therefore in our implementation each item would have its LWG, while the set of all buyers is the HWG.

On the other hand, when two unrelated groups share a common pool of resources, interference between two otherwise independent activities may occur. Thus, the challenge of building an efficient LWG service is in managing the conflicting goals of maximizing resource sharing while, at the same time, minimizing interference. In order to do so, the service may be forced to re-configure at run-time, in response to changes in the membership of the user groups (LWGs).

Ideally, the use of a Light-Weight Groups should be transparent to the application code, thus the LWG service exports precisely the same application programmers interface as the underlying basic group communication service.

There are three main tasks that a LWG service must perform:

- preserve the virtual synchronous interface of the HWGs to the LWGs users;
- define, evaluate and decide on the appropriateness of the mappings between LWGs and HWGs in a way that satisfies the requirements stated;
- execute, when required, a *switch protocol* that changes the mappings, by commuting a LWG from an original HWG to a more suitable HWG.

Mappings between LWGs and HWGs should be stored in a way that can be accessed by every process. This is achieved by an external *Name Service* that can be accessed to store changes in mappings and later to obtain the current mapping of some given LWG.

The implementation of the LWG service in partitionable networks requires additional tasks. To ensure that concurrent group views are merged when connectivity is reestablished, the system must support some sort of "peer-discovery" mechanisms and execute partition healing protocols. This poses additional problems for the LWG service because in the presence of a partition is impossible to ensure coordination of the mapping policies. Thus, in order to heal the partition the LWG service must also conciliate (potentially) different mapping decisions. The following strategy is used[9]:

1. The LWG service relies on a partitionable HWG to offer us failure detection, "peer-discovery" at the HWG level, merging of HWG views, and virtually synchronous communication. We therefore avoid "re-implementing" at the LWG level the complex protocols that are supported by the HWG. These mechanisms are therefore transparent to the LWG.
2. The LWG service also relies on an external naming service to provide up-to-date mappings between LWG views and HWG views. The naming service must be implemented using distributed cooperative servers in order to keep working correctly in the presence of network partitions.
3. With the two previous mechanisms, once a HWG partition heals, the LWG must merge, following the steps:

   Step 1 Concurrent LWG views mapped on different HWGs become aware of each other.

   Step 2 All the LWG views that satisfy the previous step must re-map in order to become mapped on the same HWG.

   Step 3 After all concurrent LWG views share the same HWG, they become aware of each other using "peer-discovery" mechanisms local to the HWG.

   Step 4 Concurrent views of a LWG mapped on a single HWG, merge into a single LWG view.

The LWG service has been designed considering group communication systems that support the *Virtual Synchrony* model[3]. Informally, virtual synchrony delivers to the participants membership information in the form of *views*. Processes that remain active and mutually reachable deliver the same sequence of views. Additionally, if two processes deliver two consecutive views, v^1 and v^2 they deliver exactly the same set of group messages between these two views (these messages are said to be delivered in view v^1). To ensure the ordering of messages with regard to views, before a new view is delivered a protocol is executed to ensure agreement on the messages that must be delivered on the previous view. This protocol is often called a *flush* protocol. To ensure liveness, communication is usually temporarily stopped in the group during the execution of the flush protocol.

3 The Ensemble System

Ensemble [5] is a protocol composition framework that has been designed to support the development and execution of group communication protocols. Ensemble is mainly focused in vertical composition. A protocol stack is a vertical composition of layers with a private session state. Therefore, it is impossible to create two stacks that share one or more layers. As will be seen, this makes it difficult to implement a layer that dynamically establishes mappings between "partial" stacks.

The interaction between adjacent layers is supported by the exchange of events. A layer is therefore designed to receive, process and possibly generate events. A predefined set of approximately forty events is defined. These events are recognized by most layers and some of them have a specialized processing by the protocol kernel itself. There are no explicit mechanisms to allow the protocol designer to create new events.

A particularity of Ensemble is that the application layer is not located on top of the stack, like in the OSI model. Instead the application, more correctly the Ensemble application interface, resides on the side of the stack, communicating with a layer in the middle of the stack. The purpose of this architecture is to optimize the data path, avoiding that data messages have to cross the complete stack before being delivered to the application.

The application interface consists of a set of call-back functions that can be configured by the application. The call-back functions have a direct relation with some of Ensemble events, such as data messages, although the events themselves are transparent to the application. Some call-back functions are allowed to return a set of operations that the application wishes to execute, like sending a message or prompting for a new view. Because this is the only mechanisms the application has to request the execution of operations, one of the call-back functions, "heartbeat", is called periodically. Its main purpose is to allow the propagation of events that are not created in response to external events.

One important characteristic of Ensemble is the language in which it was developed, the *Objective Caml* (OCAML). This is a functional language, derived from the ML language. An important characteristic of OCAML is the use of garbage collection; this makes the code safer but raises several performance concerns. Notably the OCAML Garbage Collector is not efficient in the management of certain data structures, including variables of large size such as the payload of messages, which forces the protocol programmer to explicitly manage these structures. Ensemble is thus equipped with tools and mechanisms to explicitly manage a data structure that encapsulates these large chunks of data, the "Iovecl". The programmer also has to be careful when using data structures that, due to their nature, tend to generate large quantities of garbage collectible items. It is therefore advisable to use only the imperative features of OCAML, somewhat limiting the advantages of its functional paradigm. Ensemble follows these rules and achieves good performance results.

4 Light-Weight Groups in Ensemble

This section describes the design and implementation decision that had to be taken in order to implement the Light-Weight Groups in the Ensemble system. In particular, we give emphasis to the changes to the original design that had to be made due to the specific features of the platform.

4.1 Architecture

In its first implementation for the Horus system, Light-Weight Groups were implemented as a protocol layer residing on top of the virtual synchrony layer (more precisely, on top of the collection of protocols implementing virtually synchrony). In the Horus implementation, the LWG layer acted as multiplexing layer, connecting LWG "half-stacks" above with the underlying HWG "half-stacks". Naturally, the state of this multiplexing layers has to be shared by all channels using the service, such that appropriate mapping decisions can be made based on the relative membership of different groups

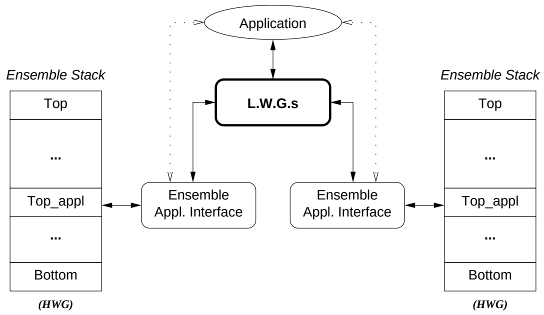

Figure 2: LWG positioning in Ensemble

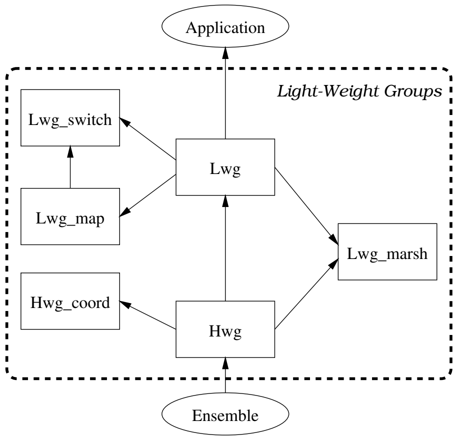

Figure 3: Main LWG Blocks

Unfortunately, the same type of architecture cannot be used in Ensemble, since sharing is not allowed. This is a fundamental limitation of the Ensemble architecture. The only way to circumvent this limitation was to implement the LWG group as an extension to the application interface as illustrated in Figure 2. This LWG interface wraps the original Ensemble API and multiplexes the user calls to different Ensemble stacks.

However, not being a layer in the protocols stack has a number of limitations: *i)* It can only be placed in a predefined fixed position in the stack, limiting the configuration alternatives; *ii)* All LWGs mapped to the same HWG must use exactly the same stack, and its configuration; *iii)* The LWG layer doesn't have access to some events that flow through the stack that are not delivered to the application.

4.2 Building Blocks

The Light-Weight Groups service implementation has been decomposed in the building blocks depicted in Figure 3. This section provides a brief description of each of these blocks.

The "Hwg" component implements the interface with the basic Ensemble stacks. Its main responsibility is to register itself with Ensemble, like an application, and forward to the other components the events it receives from Ensemble, and vice-versa. It is responsible for maintaining an HWG pool, and for storing local mappings between LWGs and HWGs.

Similarly, the component responsible for the application interface is the "Lwg". To ensure that the service is transparent to the application, this module exports exactly the same interface offered by Ensemble. The "Lwg" component also implements all the LWG internal functionality, such as the removal from the LWG view of members that are no longer in the HWG view, or the installation of a merged LWG view.

Most of the algorithms that support the Light-Weight Groups service are coordinator based. That is, a specific member of each view, typically the oldest member of the view, assumes the role of the coordinator to perform a number of management activities such as installing new views, initiating view merges, triggering new mappings, etc. This functionality is encapsulated in the "Hwg_coord" component. It basically keeps the LWG views mapped on the HWG, checks to see if two concurrent views exist, and if so it orders a merge. It also provides the new merged view by sending an ordered set of all current concurrent LWG views. It then requests a view change of the HWG, to act as the synchronization point. In fact, because the coordinator is the only member allowed to request a new view, when some other member wishes a view change, for instance when it wants to leave, it must pass the request to the coordinator. Because a LWG view change implies a HWG view change, it may cause large interference. For this reason the coordinator tries to gather the greatest number of LWG views changes in a single HWG view change, by delaying some of the changes.

The coordinator also does all communication with the *Name Server* (NS), to minimize messages. When a new view is installed all mappings are sent, in a single message to the NS.

The protocols required to perform a switch of the mapping of a LWG, from one HWG to another HWG are encapsulated by the "Lwg_switch" block. A switch may have to be performed when two partitions reconnect or when the current mappings are no longer appropriate due to changes in the membership of LWGs. The module responsible for keeping track of the appropriateness of the current mappings is the "Lwg_map" component. It basically runs a periodic test to see if the mapping is appropriate, according to some predefined heuristics, and if the mapping remains unsatisfactory it triggers a switch.

Finally, the LWG service depends upon the availability of an external Name Server. Naturally, to tolerate network partitions and failures, the name service is also replicated.

4.3 Algorithms

As noted in the introduction, the implementation of the Light-Weight Groups service in Ensemble was the first to include network partition support. The algorithms described in [9] were slightly adapted to minimize the amount of control traffic, by changing some of the fully decentralized approaches for a more coordinator-based approach. The following algorithms had to be implemented:

- Algorithms to merge two concurrent views.
- Algorithms to switch mappings.
- Algorithm to merge concurrent views mapped on different HWGs (basically a combination of the two steps above).

These algorithms must ensure that the virtually synchronous properties of the system are preserved. Therefore, they require the execution of a flush protocol on the affected LWGs. There are two main approaches to implement the flush procedure at the Light-Weight Groups layer:

- To rely on the flush mechanisms of the underlying HWG. If a flush of the HWG is forced, all LWGs mapped on it will implicitly be flushed. This simplifies the implementation but does not minimize the interference (since some LWGs are disturbed by the view changes of other LWGs).
- To implement at the Light-Weight Groups layer a flush protocol, that would run independently for each LWG. This solves the interference problem but introduces some redundancy with regard to the mechanisms already implemented at the HWG layer.

Despite the redundancy of code, for better performance we were willing to implemented the second alternative. Unfortunately, it turns out that the Ensemble mechanisms that ensure the coordination among layers during view changes preclude the possibility of a LWG flush without flushing the complete stack. In fact, the coordination mechanisms assumes that the complete stack is blocked during the view change and that a view unblocks the stack. Therefore we were forced to implement the first alternative.

5 Lessons Learned

The experience with Ensemble highlighted both positive and negative aspects of the architecture. As a positive aspect, we must emphasize the event model used to support protocol composition, which is highly intuitive and efficient. The complete and powerful set of protocols already implemented in the architecture also simplifies the development of additional services. On the other hand the composition model was very restrictive, forcing the Light-Weight Groups service to be implemented as a wrapper to the Ensemble interface. This is a rather awkward solution since, not being a full fledged layer, prevented the reception of some useful events. Another significant limitation is the use of a closed event model: all events that flow in the stack are predefined and it is not trivial for a set of cooperating layers to add their own events or to refine previously defined events.

In our current work, we are developing a framework for protocol composition, named *Appia* [7], that offers a more flexible composition model and allows programmers to create their own set of events.

In terms of programming model, the strong virtually synchronous model proved to be very useful in the design of the Light-Weight Groups merge and switch protocols. However, the coordination mechanisms implemented in Ensemble are also too restrictive. In particular, the impossibility of delivering a view without unblocking the upper layers makes it very difficult for other layers to control view delivery. In our case, this prevented the development of a merge protocol that was local to a single LWG without flushing the complete HWG. This is a source of interference among LWGs that could be eliminated.

The way joins are handled in the Ensemble virtually synchronous layers is also not the most convenient for the implementation of our service. In Ensemble, when a node wants to join a group, it first creates a singleton view which is later merged with the view of the current members. Although Ensemble tries to optimize the case where several nodes join at once, by inserting an artificial delay before the merge, the upper layers have no control on the sequence of joins. When the Light-Weight Groups layer reconfigures the mappings, it may require that a set of nodes join a different HWG. Performance could be strongly improved if all these joins were performed in a single operation at the lower levels but, with the current architecture, there is no way to enforce this behavior and, in the worst case, each member may join independently.

Another more practical lesson, related with the implementation using the OCAML language, is that the programmer must be very careful with the structures it uses to avoid delays associated with garbage collection. Since we have opted with the use of Java in the *Appia* system, as this language has many advantages in other dimensions, we are faced with the same sort of concerns.

6 Conclusions

In this paper we have briefly described the implementation of the Light-Weight Groups in the Ensemble platform. We have implemented and extracted a number of lessons from this experience. Some of these insights have motivated us to experiment with alternative protocol composition patterns that we are developing in the context of the *Appia* system. One of the concerns in the *Appia* design was to support complex stacks, for instance fork shaped stacks like the ones required by the LWG service [1]. Layer sharing between stacks is an important feature of the system. It therefore supports the Light-Weight Groups concept in a much more convenient way than Ensemble.

Early tests with the Ensemble implementation show a minor overhead in latency (less than 1%) but a significant overhead in throughput (30%) for zero length messages. A careful analysis of the sources of overhead needs to be performed, but, for the zero byte messages, the overhead is due to the increased size in message header (this overhead becomes negligible for larger messages). For future work we are interested in implementing the same service in the new *Appia* platform and to do extensive performance tests on both architectures. The comparative analysis of these two implementations, along with the implementation that was previously performed on Horus, will allows us to obtain a better understanding of the positive and negative features of each of these protocol composition frameworks.

[1] Other forked shaped examples are given in [8]

References

[1] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[2] K. Birman and T. Joseph. Exploiting replication in distributed systems. In Sape Mullender, editor, *Distributed Systems*, pages 319–366. ACM Press Frontier Series, 1989.

[3] K. Birman and R. van Renesse, editors. *Reliable Distributed Computing With the ISIS Toolkit*. Number ISBN 0-8186-5342-6. IEEE CS Press, March 1994.

[4] K. Guo and L. Rodrigues. Dynamic light-weight groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 33–42, Balitmore, Maryland, USA, May 1997. IEEE.

[5] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[6] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[7] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, page to appear, Phoenix, Arizona, April 2001. IEEE. http://appia.di.fc.ul.pt.

[8] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cacún, México, September 1999. IEEE.

[9] L. Rodrigues and K. Guo. Partitionable Light-Weight Groups. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'20)*, pages 38–45, Taipe, Taiwan, April 2000. IEEE.

[10] R. van Renesse, Ken Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.