Non-hierarchical Dynamic Protocol Composition in Jgroup/ARM

Hein Meling
hein.meling@uis.no

Department of Electrical and Computer Engineering
University of Stavanger, 4036 Stavanger, Norway

Abstract

Protocol composition is a common approach to structure generic protocols used by networked applications, and typically a vertically layered approach is taken. This paper presents an alternative approach, where the protocol composition is a weakly-coupled set of protocol modules organized in a non-hierarchical structure. Protocol modules are dynamically constructed at runtime. The approach is designed for systems that involves multiple communicating entities and multicast style interactions are supported, making the approach suitable for building reliable network applications. The main advantage of the approach is that modules in the same composition communicate by direct interaction, whereas other frameworks typically use a vertically layered protocol stack, forcing all messages/events to pass through all intermediate layers introducing unnecessary delays.

1 Introduction

Networked computer systems are prevalent in most aspects of modern society, and we have become dependent on such computer systems to perform many critical tasks. Moreover, making such systems *dependable* is an important goal. Yet, dependability issues are often neglected when building systems due to the complexities of the techniques involved. Modularization is a well-known principle for simplifying complex systems. Furthermore, a common technique used to improve the dependability characteristics of systems is to replicate critical system components whereby the functions they perform are repeated by multiple replicas. Replicas are often distributed geographically and connected through a network as a means to render the failure of one replica independent of the others.

This paper presents the design and implementation of a protocol composition framework for the Jgroup/ARM middleware platform [8, 11]. Jgroup/ARM is a middleware framework for developing and operating dependable distributed applications based on Java. Jgroup [11] integrates the distributed object model of Java remote method invocations (RMI) with the *object group communication* paradigm, enabling the construction of groups of replicated server objects that provide dependable services to clients. The *Autonomous Replication Management* (ARM) framework [8] provides automated mechanisms for distributing replicas to host processors and recovering from replica failures.

This paper was presented at the NIK-2006 conference; see http://www.nik.no/.

The Jgroup/ARM middleware platform is aimed at simplifying the development of dependable network information services. One part of this simplification is accomplished through modularization of generic protocol modules and composing them into complete protocol stack. Protocols in Jgroup/ARM usually involves multiple communicating entities, i.e. all members of a replicated object group, and specialized multicast interactions are supported for interactions with peers in the same group. Protocol modules in the same composition communicate by direct interaction. Using this framework, a dependable service can easily construct and configure its own protocol composition dynamically at deployment time. Each protocol module in the composition can be parametrized according to the dependability requirements of the service. Adding new protocols to the system is also very easy.

The paper is organized as follows: Section 2 discuss previous works on protocol architectures and relate these to the approach taken by Jgroup/ARM. Section 3 briefly presents the Jgroup/ARM middleware. Section 4 introduces the concepts on which the protocol composition framework is based, and illustrate a sample protocol stack. Section 5 details the various ways in which protocol modules can communicate, both internally and externally. Finally, in Section 6 the dynamic composition of protocol modules is discussed and Section 7 concludes the paper.

2 Introduction to Protocol Architectures

Protocol composition is traditionally based on layered protocol stacks. However, in the last decade, micro-protocols have become increasingly popular, as they enable a more flexible approach to protocol composition. To accomplish this, micro-protocol frameworks restrict their protocol layers to follow a specific model, rather than building protocols in an ad hoc manner. These restrictions include: the protocol layers have to communicate using events that travel up or down the protocol stack, and that the layers cannot share any state. This way protocols become more maintainable and configurable as new protocols can easily be added to the system. The cost however, is reduced performance.

Micro-protocols were first introduced in the *x*-kernel [7], and have since been used in a variety of systems, including group communication systems such as Ensemble [5], Horus [16], JavaGroups [3], Cactus [6] and Appia [10]. Ensemble, Horus, JavaGroups and Appia [5, 16, 3, 10] follow a strictly vertical stack composition, where events must pass through all layers in the stack. In the Horus system, a protocol accelerator [15] implements optimizations that reduce the effects of protocol layering. The limitation of these optimization techniques is that the set of protocols to be bypassed must be well-defined, and the optimizations were hand-coded into the protocol stack. Thus, it reduces the configurability of the micro-protocol framework. Similar optimizations are also feasible with the Ensemble system [5]. Both Appia [10] and JavaGroups [3] are also based on micro-protocols in its purest form, since none of the optimizations implemented in Horus and Ensemble are available. That means that every event has to pass through all intermediate layers, even though the event is not being processed by all of the layers. The Cactus [6] micro-protocol framework is conceptually similar to the protocol composition framework discussed in this paper. Each layer has to register its interest in the events of other layers, and protocols can be constructed according to formal rules, such as a dependency graph. Thus, such a protocol stack does not follow a strict vertical composition. An advantage of the Jgroup protocol framework over JavaGroups, Appia and the Cactus system is type-safety. Events are passed by means of method calls on a set of well-defined interfaces for the various modules (layers), whereas other systems have to implement a common handler method in each layer which takes care of demultiplexing the received events based on the type of the events. In Jgroup, events are passed directly

to the appropriate event handler. Another advantage of Jgroup over the Cactus system is the possibility to specify interception rules, enabling a module to delay and/or modify events from another module. The SAMOA [12] framework is also conceptually similar to the approach in this paper. The main differences are that SAMOA supports concurrency and asynchronous stack internal interactions, whereas Jgroup uses synchronous interactions and leaves concurrency a non-framework issue. Synchronous interaction is a simpler approach, and makes it easier for developers to write protocol modules. Both the Neko [14] protocol prototyping framework and Jgroup/ARM uses an approach based on *dependency injection* [4].

3 The Jgroup/ARM Middleware

Jgroup/ARM [8, 2, 11] integrates the Java RMI *distributed object models* with the *group communication* paradigm and *autonomous fault treatment*. Jgroup provides three core services aimed at simplifying coordination among replicas: a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS) [11].

The task of the PGMS is to provide servers with a consistent view of the group's current membership, to enable server coordination. Reliable communication between clients and the object group take the form of *group method invocations* (GMI), that result in methods being executed by the servers in the group. To clients, GMI interactions are indistinguishable from standard Java RMI: clients interact with the group through a *group proxy* that acts as a representative object for the group, hiding its composition. The group proxy maintains information about the servers composing the group, and handle invocations on behalf of clients by establishing communication with one or more servers and returning the result to the invoker. This form of GMI is called External GMI (EGMI). On the server side, the GMIS enforce reliable communication among replicas within the group and are called Internal GMI (IGMI). Finally, the task of SMS is to support developers in re-establishing a global shared state when two or more partitions merge after a network partition.

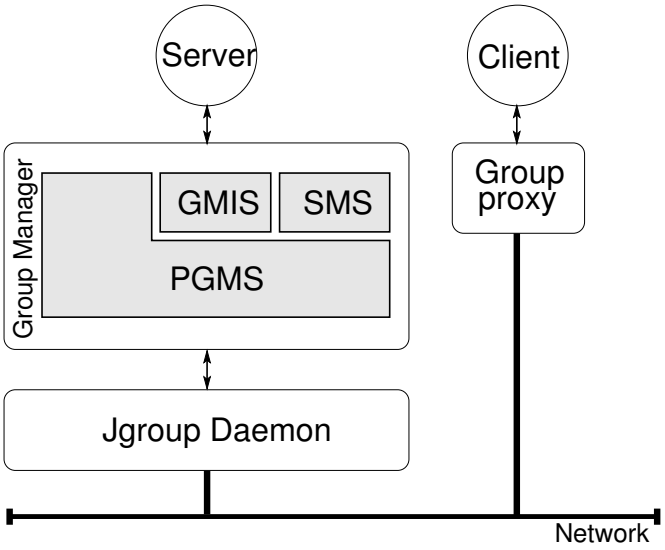

Figure 1. Overview of Jgroup services.

Figure 1 gives a high-level overview of the composition of the core Jgroup services. The main component of Jgroup is the *Jgroup daemon*; it implements basic group communication services such as failure detection, group membership and reliable communication. Server replicas must connect to a Jgroup daemon to access to the group communication services. Each server replica is associated with a *group manager* (GM), whose task is to act as an interface between the Jgroup daemon and the replica.

The ARM framework [8] provides mechanisms for automated fault treatment and management activities such as distributing replicas on sites and nodes, and recovering from failures, reducing the need for human intervention. These mechanisms are essential to operate a system with strict dependability requirements, and are largely missing from existing group communication systems [11, 10, 3]. Much of the ARM functionality is implemented by separate protocol modules integrated into the GM component.

4 Protocol Modules

The *group manager* (GM) is the glue between an application and the core group communication services; it encapsulates all protocol modules associated with the application. It allows the application to interface with the various Jgroup services to perform group-specific tasks. The GM is based on an *event-driven non-hierarchical composition model*[1], and consists of a set of *weakly coupled protocol modules*. Each protocol module implements a group-specific function, which *may* require the collaboration of all group members, e.g. the membership service (PGMS). In fact, all the basic Jgroup services discussed in Section 3 and several other generic group-specific functions are implemented as GM protocol modules.

The advantages of a non-hierarchical set of protocol modules over a strictly vertically layered architecture, as used in many other group communication systems (e.g. [16, 5, 3, 10]), is that events being passed from one layer (module) does not have to be processed by any intermediate layers. Events can simply be passed from one module to another without any processing delay and addition/removal of header fields, thus also reducing the complexity of implementing a module. Our approach is also flexible in that a module can intercept commands/events from another module, delay and/or modify them, before delivery to the destination module. Interception rules are specified inline in the modules using annotations, and the corresponding implementations must adhere to these rules.

Protocol modules communicate with the application, or other modules, by means of *commands* (downcalls) and *events* (upcalls) through a set of well-defined interfaces. Typically, a module *provides* a set of services to other modules and/or the application, and *requires* another set of services from other modules to perform its services. A module may also *substitute* the services provided by another module, by intercepting, delaying and/or modifying the commands/events passed on to the substituted module.

Each module implements one or more well-defined *service interfaces*, through which the module can be controlled, and it may also generate events to listening modules (or the application) through one or more *listener interfaces*. Usually, a module implements one service interface and provide events to other modules through one listener interface. As an example, consider the MembershipModule which is defined by the MembershipService and MembershipListener interfaces, shown in Figure 2. Servers can access the service interfaces of protocol modules by querying the GM. However, to be notified of events generated by a module, a server only needs to implement the module's listener interface.

The set of GM protocol modules required by an application is configured through the ARM policy management [8]. Based on this configuration, the protocol modules are constructed dynamically at runtime. The advantage of dynamic construction is that it enables developers to easily build generic group-specific functions and augment the system with new modules without having to recompile the complete framework. There is no strict ordering in which the modules have to be constructed, except that the set of required modules must have been constructed *a priori*. During construction, each module is checked for structural correctness, and required modules are constructed on-demand.

Figure 2 illustrates a protocol composition containing the basic Jgroup services, except the GMIS. For readability only the most important commands/events are shown in the interfaces. The DispatcherModule is responsible for queuing and dispatching events to/from the daemon, and is the interface between the GM protocol modules and the daemon. The MulticastModule implements the MulticastService through which other modules (and the server) can send multicast messages to the group members. To receive messages, a module must implement the MulticastListener interface. The main task of the MulticastModule is to multiplex and demultiplex messages to/from the internal modules

[1] In [9] this is called *cooperative composition*.

or the server. The actual low-level IP multicast is performed by the daemon. Other modules (or the server) can join() or leave() a group by invoking the MembershipService interface, which is implemented by the MembershipModule. Variations in the group membership are reported through viewChange() events. Any number of modules, and the server, may register its interest in such events simply by implementing the MembershipListener interface. The MembershipModule mainly keeps track of various state information and provides an interface to the PGMS, whereas the view agreement protocol [11] is implemented in the daemon. The DispatcherModule, MulticastModule and MembershipModule are mandatory, and must always be included for any sensible group communication support.

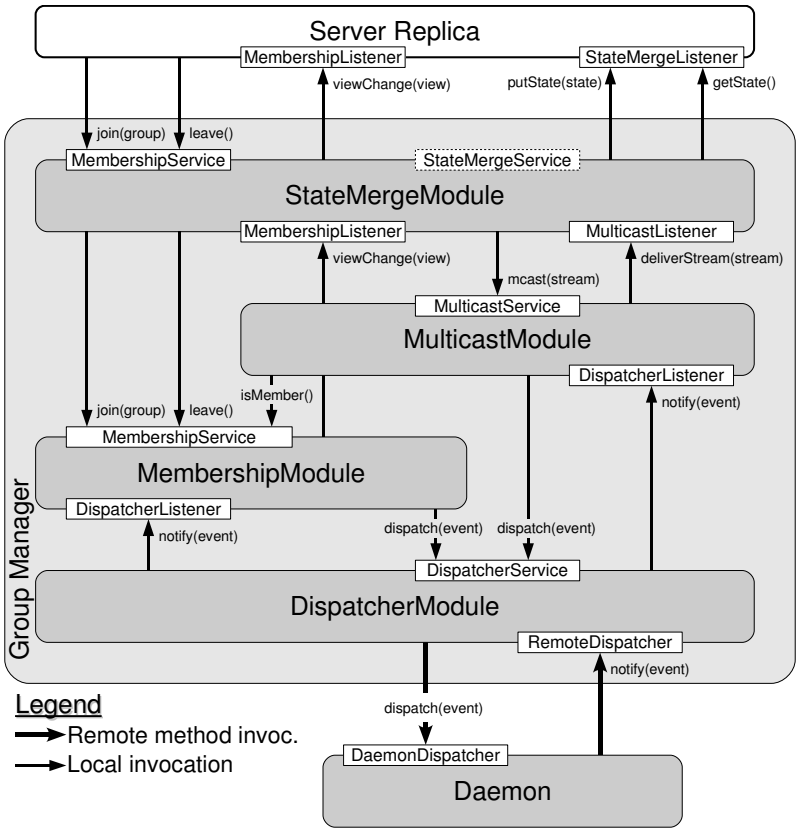

Figure 2. A sample group manager composition with the basic Jgroup services.

Note that the StateMergeModule also implements the MembershipService interface, and provides events through the MembershipListener interface. This is since the StateMergeModule substitutes the membership service by intercepting and delaying the delivery of viewChange() events to the server until after the state has been merged. The main task of the StateMergeModule is to drive the state reconciliation protocol by calling getState() and putState() on the StateMergeListener interface to obtain and merge the state of server replicas. It also handles coordinator election and information diffusion. State reconciliation is only activated when needed, i.e. in response to viewChange() events generated by the MembershipModule. Hence, the StateMergeService interface (dashed box) does not provide commands as a means for activating it. As Figure 2 illustrates, the StateMergeModule requires both the MembershipModule and the MulticastModule, and substitutes the MembershipModule.

5 Module Interactions

Protocol modules may interact in a number of different ways, both with external entities and other protocol modules. Hence, to construct the protocol modules dynamically, it is necessary to understand how the modules can interact so as to dynamically establish the necessary links between them.

Figure 3 illustrates inter-module and server-to-module interactions. Inter-module interactions may occur both within the same GM, and also across distinct GMs. Mostly, only GMs that belong to the same group needs to communicate. GMs belonging to the same group should be composed of an identical set of protocol modules. The arrows in Figure 3 represents a *may* communicate relation. That is, a module may or may not communicate with another module in one or both directions. The servers may also interact *directly* with one or more of the modules within its local GM, without passing through any intermediate modules. The thicker arrows represent remote interaction between peer modules.

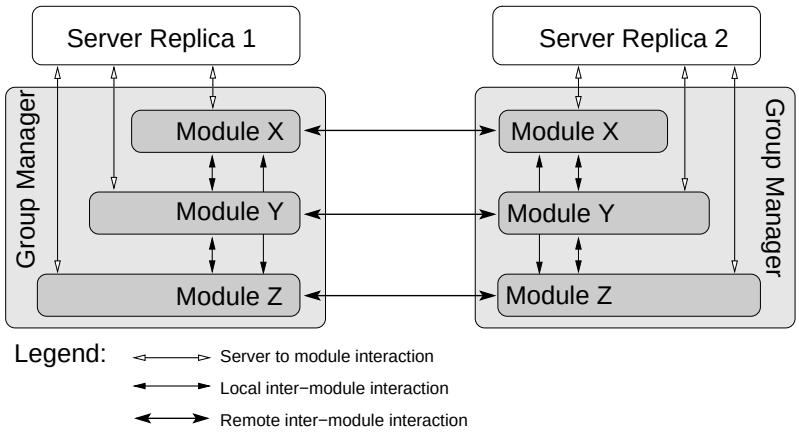

Figure 3. Inter-module and server-to-module interactions.

Four distinct forms of interaction styles involving protocol modules have been identified, as listed below. The first three are shown in Figure 3.

1. Local inter-module interactions between modules internal to the same GM.
2. Remote inter-module interactions between peer modules in distinct GMs.
3. Interactions between the server and local protocol modules.
4. Interactions between an external entity and a protocol module.

The last interaction style allows a protocol module to notify or to be notified by an external entity. In the following, each of these interaction styles are discussed individually. Although commonplace, server-to-server interactions are not considered here.

Local Inter-module Interactions As mentioned above, the GM is composed of a collection of protocol modules, each of which may provide a service to other modules in the same GM. In addition, a protocol module may also listen to events from other modules. Figure 4 illustrates a generic view of the internal inter-module interaction interfaces, through which local protocol modules communicate. In the figure, the service interface implemented by module A is used by modules B and C within the same GM to invoke commands offered through the service interface (e.g. to join() a group). Module A also implements a set of listener interfaces through which it can be notified of events generated by modules D and E (e.g. a viewChange() event.)

A module *must* implement at least one service interface, but may also implement more than one service (not shown in Figure 4). Implementing multiple service interfaces is useful when a module

intercept and substitute the services of another module, e.g. the StateMergeModule in Figure 2. For most other circumstances a module should implement only a single service interface to encourage reuse.

The service interface typically contains one or more commands ($c_1, \ldots, c_k$), that can be invoked by the server or other modules. The service interface may also be empty in that it does not provide any commands (methods). Such empty interfaces are often called *marker* interfaces, and serves to identify the module internally in the GM. The dashed box around the StateMergeService interface in Figure 2 is one example of an empty marker interface.

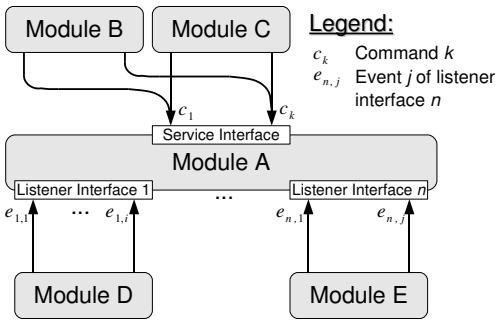

Figure 4. A generic view of the interfaces used for local inter-module interactions.

A module *may* have one or more associated listener interfaces through which module generated events can be passed to its listeners (other modules or the server). Figure 2 illustrates the use of multiple listener interfaces; the StateMergeModule generates events through both StateMergeListener and MembershipListener, since the StateMergeModule substitutes the MembershipModule. Usually however, a module generate events through a single listener interface (see Figure 4). A module without any associated listener interfaces is useful only when the module provides services commands. A module *may* receive events ($e_{1,1}, \ldots, e_{1,i}$) generated by other modules by implementing one or more listener interfaces.

The service and listener interfaces are defined in terms of Java interfaces (ensuring type safety), and arrows in Figure 4 represents Java methods (commands/events).

Remote Inter-module Interactions Modules within one GM may interact with its remote peer modules in other GMs belonging to the same group. Two approaches can be used by module developers to support interaction between peer modules:

- Message multicasting (using the MulticastModule)
- Internal group method invocations (using the InternalGMIModule).

The advantage of the former approach is primarily efficiency, since it adds no overhead to the messages being sent by the module, except for a small header used to route multicast messages to the appropriate peer modules. The drawback with message multicasting is that module complexity increases, since the developer must implement marshalling and unmarshalling routines for the different message types to be exchanged between peer modules.

Contrarily, the InternalGMIModule takes care of marshalling and unmarshalling, reducing the module complexity to pure algorithmic considerations. The InternalGMIModule do however impose an additional overhead compared to that of message multicasting. The overhead is mostly due to the use of dynamically generated proxies [1, Ch.16]. Albeit not confirmed through measurements, the expected overhead imposed by the proxy mechanism is small compared to the communication latencies between the peer modules. Details of the workings of the InternalGMIModule as a means for communication between peer modules is given in [8].

Server to Module Interactions The server implementation may interact with the local modules. Figure 5 shows a generic view of the server-to-module interactions. A server replica *may* choose to listen to an arbitrary set of events generated by its associated protocol modules. To accomplish this, the server must implement the listener interfaces associated with the modules whose events are of interest. The server may also choose to not implement any listener interfaces if it does not need to process events generated by modules. In a similar manner, the server *may* invoke any one of the commands provided through the service interfaces of the protocol modules associated with the server.

As Figure 5 demonstrates, various combinations of using services and listening to events are possible. The server may both listen to events of a module, and invoke its service commands (middle module), or it may just listen to its events (left module), or just invoke its service commands (right module).

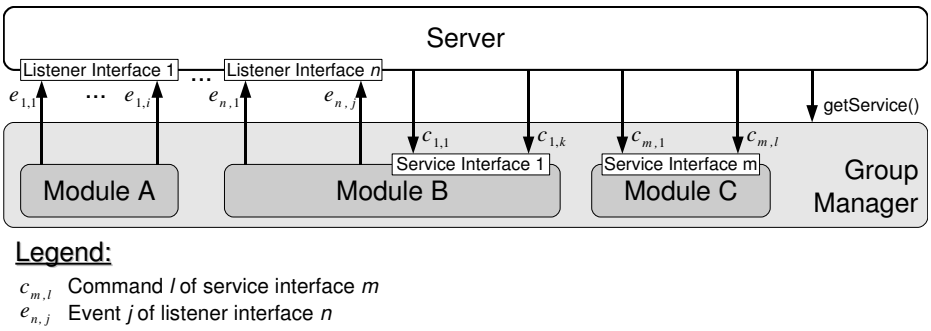

Figure 5. A generic view of the server-to-module interaction interfaces.

Establishing the connections between the server and its associated set of protocol modules is done through the GroupManager object. The GroupManager object wraps the protocol modules and acts as an interface between the modules and the server. Initially, when the server requests group communication support it will invoke the getGroupManager() factory method, passing its own reference (**this**). Given the server reference, the GM establishes upcall connections between the server and the modules whose listener interfaces are implemented by the server. On the other hand, establishing connections between the server and the service interfaces of modules are done on-demand by the server implementation itself. This is accomplished using the GroupManager.getService() method shown in Figure 5. Given a reference to the service interface of a module, the server can easily invoke its commands.

External Entity to Module Interactions Protocol modules may also interact directly with (possibly replicated) external entities. For instance, a protocol module could invoke methods on an external entity or vice versa. This interaction style is useful for a number of purposes, such as event logging, event notifications or triggering some action, e.g. recovery [8] or upgrade [13].

External entities and modules can interact in both directions, as shown in Figure 6. External Entity 1 invokes Module X to perform some operation implemented by the module, whereas External Entity 2 allows a module to invoke methods on it to perform some operation. Interaction with external entities relies on the dependable registry for looking up the reference of the external entity (or the module) with which to communicate. Prior to such lookups, the receiving end must bind() its reference in the dependable registry. The two interactions shown in Figure 6 are both based on EGMI, and hence the receiving end must include the ExternalGMIModule in its set of protocol modules.

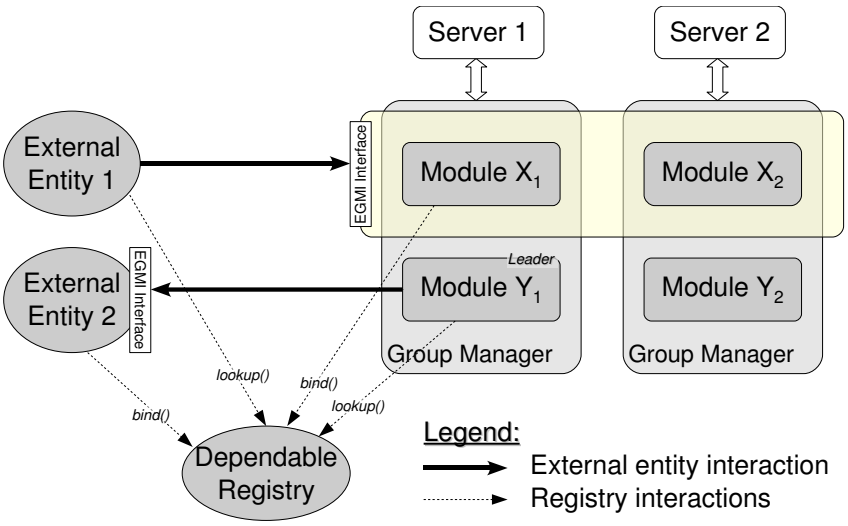

Figure 6. External entity to module interactions.

6 The Dynamic Construction of Protocol Modules

The group manager encapsulates the set of protocol modules associated with an application. Protocol modules are configured using an application-specific replication policy [8]. The policy supports specifying the set of protocol modules to be constructed, as well as supplying configuration parameters to the modules, e.g. timeout values and the redundancy level to maintain.

Protocol modules are constructed dynamically at runtime based on the replication policy of the application requesting a protocol composition. This is essentially all a server developer needs to know about the construction of protocol modules. However, a module developer needs to have more intimate knowledge of the architecture which simplifies the following tasks:

- Automatic construction of protocol modules.
- Establishing links between dependent modules.
- Establishing links between the server and its dependent modules.
- Reconfiguration of links for module substitution.

The dynamic construction facility requires that module developers adhere to these rules:

1. The module must contain a single constructor, whose signature contains the set of services *required* by the module.
2. The module *must* implement the Link interface.
3. The module *may* implement the Bootstrap interface.
4. The module *may* implement listener interfaces of other modules.
5. The module *may* declare that it *substitutes* the services/listeners provided by another module.

Figure 7 illustrates the rules in terms of interfaces. Solid boxes indicate required interfaces, while dashed boxes denote optional interfaces which may be implemented by a module depending on its requirements.

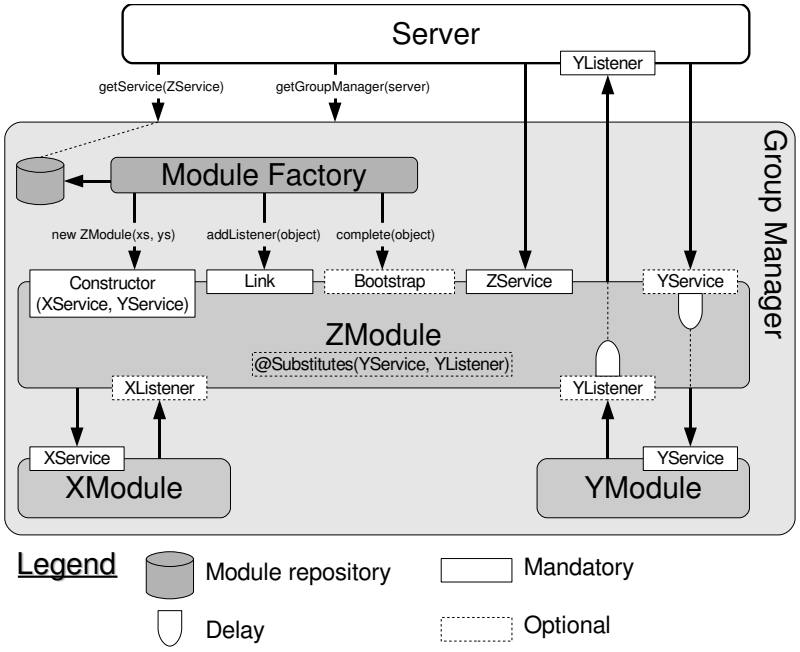

Figure 7. The module factory and interfaces used for module construction.

Module Instantiation As shown in Figure 7, ZModule requires two other services, XService and YService, which are implemented by XModule and YModule, respectively. These two modules must have been constructed prior to ZModule, and are thus passed to the ZModule constructor. The *module factory* uses reflection [1, Ch.16] to examine the constructor signature of the ZModule to determine its required module dependencies, and queries the *module repository* to obtain the required module instances. If a required module instance is not found, it will be created on-demand and stored in the repository. Note that cyclic module dependencies are not possible with this approach, i.e. if ZModule depends on XModule and vice versa, they cannot be constructed using the scheme above. However, dependency cycles are still possible through minor supplements to the mutually dependent modules.

Construction order may sometimes be important for correct functioning of a protocol stack. Construction follows the bottom-up order specified in the replication policy [8]. Referring to Figure 2, this means that the DispatcherModule is constructed first, followed by the MembershipModule and so on. Note that the DispatcherModule does not depend on other modules, but is instead responsible for establishing a connection with the daemon.

Link Configuration Once all the protocol modules associated with an application have been instantiated, links between the modules are established by the module factory through the mandatory Link interface. The addListener() method shown in Figure 7 serves two primary purposes:

- To establish upcall links with other modules and the server; links are only established with modules (or the server) implementing the listener interface associated with the module.
- To perform bootstrap operations that cannot be performed during module construction.

In Figure 7 the object passed to the addListener() method may be either the server object or a module. Note that the server object is always passed to the addListener() method, independent of it implementing the listener interface associated with the module. Thus the module can exploit the server reference type as a means to obtain necessary configuration data from the replication policy, e.g. timeout values, to configure/bootstrap the module. If the server object does not implement the listener interface of the module, it cannot receive any events from the module. Furthermore, the addListener() method may be invoked several times for distinct modules, allowing multiple modules to receive the same set of events. The order in which addListener() is invoked follows the construction order defined above, with the server object passed in last.

Bootstrapping Some modules may need to perform supplementary bootstrap operations after all the links have been established. The final task performed by the module factory is to find modules that implement the optional Bootstrap interface, and invoke its complete() method to perform the final bootstrap operations. For instance, the server could configure its replication policy to automatically join() its group during the bootstrap phase, simplifying the server implementation even further. Joining the group requires that all the links have been set up between all the protocol modules, and hence it cannot be bootstrapped through the Link interface. Given this bootstrap mechanism, some modules may replace its service interface with an empty marker interface, and instead bootstrap automatically.

Event Interception As advocated initially in this paper, some modules need to intercept commands/events originated in other modules. Such interception may be necessary for a number of reasons, e.g. if delivery of events must be delayed until after the intercepting module has completed its tasks. For example, a total ordering module needs to delay the delivery of messages pending agreement among group members on the sequence in which to deliver messages.

Modules that wish to intercept the commands/events of another module must declare that it substitutes the other module. The @Substitute declaration uses annotations [1, Ch.15] to indicate which service and listener interfaces to substitute. As shown in Figure 7, the ZModule substitutes both interfaces associated with the YModule. The module factory will analyze the substitute declarations and reconfigure the links accordingly, hiding the presence of the YModule from other modules and the server.

Implementing a module which substitutes another can be accomplished by inheriting from the substituted module, or by wrapping it. Note that it is essential that substituting modules be ordered appropriately in the replication policy so as to ensure correct interception.

7 Conclusions and Future Work

In this paper, the design and implementation of a protocol composition framework for Jgroup/ARM has been presented. The main feature is the direct communication between protocol modules, saving costly processing in intermediate modules. It also supports dynamic construction of protocol compositions based on a simple configuration file. A future project aims to provide support for runtime adaption and to redesign the system into a generic protocol composition framework that can be used by other middleware platforms as well, and not restricted to group communication toolkits. Another future project involves evaluating and comparing the cost of vertically stacked protocols using Appia [10], Cactus [6] and JavaGroups [3] vs. Jgroup/ARM protocol compositions.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.

[2] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Trans. Software Eng.*, 27(4):308–336, Apr. 2001.

[3] B. Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University, July 1998.

[4] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, Jan. 2004. Available from: http://www.martinfowler.com/articles/injection.html.

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

[6] M. A. Hiltunen and R. D. Schlichting. The Cactus Approach to Building Configurable Middleware Services. In *Proc. Workshop on Dep. Sys. Middleware and Group Comm.*, Nuremberg, Germany, Oct. 2000.

[7] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Eng.*, 17(1):64–76, Jan. 1991.

[8] H. Meling. *Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation*. PhD thesis, Norwegian University of Science and Technology, Department of Telematics, May 2006.

[9] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. Cactus: Comparing Protocol Composition Frameworks. In *Proc. 22nd Symp. on Reliable Distributed Systems*, Florence, Italy, Oct. 2003.

[10] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. 21st Int. Conf. on Distributed Computing Systems*, Phoenix, Arizona, Apr. 2001.

[11] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.

[12] O. Rütti, P. T. Wojciechowski, and A. Schiper. Service Interface: A New Abstraction for Implementing and Composing Protocols. In *Proc. 21st Annual ACM Symp. on Applied Computing*, Dijon, France, Apr. 2006.

[13] M. Solarski and H. Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *Proc. Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC 2002*, Oxford, England, Aug. 2002.

[14] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, November 2002.

[15] R. van Renesse. Masking the Overhead of Layering. In *Proc. 1996 ACM SIGCOMM Conference*, Stanford University, Aug. 1996.

[16] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, Apr. 1996.