Towards Flexible Finite-State-Machine-Based Protocol Composition*

Richard Ekwall Sergio Mena Stefan Pleisch André Schiper

École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

Abstract

We propose a novel approach to the composition of group communication protocols. In this approach, components are modelled as finite state machines communicating via signals. We introduce two building blocks, called adaptor and adaplexor, that ease the development and the composition of group communication protocol stacks, and we discuss how isolation can be achieved in this setting. To validate our architectural concepts, we have implemented the proposed group communication architecture in SDL.

1 Introduction

The designer of protocol stacks has the option to design a stack as a monolithic block or to compose it from a set of protocol modules. The second option has obvious advantages: it makes reuse of protocol modules possible.

In this paper we consider the design of protocol stacks for group communication [3], although our design is also valid for a number of other protocol stacks. Group communication provides abstractions for developing replicated, fault-tolerant applications. Until recently, most group communication systems have been built as a monolithic block with the drawback of reduced component reusability [1]. In contrast, reusability allows to solve the growing need for adaptability.

To achieve reusability and adaptability, protocol modules can be composed in a static way, i.e., the connections between modules have to be known at the compilation of the modules. However, this solution lacks flexibility, and may even make the approach impractical. This is the case especially if some protocol module M may potentially be used by several other protocol modules M' unknown to M when M is built. Either a solution is found to handle this case, or M has to be redesigned later.

In this paper, we present a flexible protocol composition approach, based on finite state machines (FSM), and we identify three additional building blocks that are required for flexible composition: *adaptors*, *adaplexors*, and *isolators*. While the use of adaptors is fairly obvious and has been proposed as a pattern in [6], adaplexors and isolators are less obvious but important.

To validate our composition approach, we have implemented a prototype of a group communication stack. Each layer of the group communication protocol stack is modelled as a finite state machine. These finite state machines communicate by exchanging *signals*. A signal is a notification that a FSM sends to another FSM. From a composition perspective, we found that this approach has considerable advantages over approaches chosen by other composition frameworks. For example, composing FSMs into a group communication protocol stack does not impose any restrictions on the way the different layers of the protocol stack are implemented: the entire group communication stack can be implemented by a single process, by one process per layer, or any number of layers per process.

The implementation of the protocol stack was done using the Specification and Description Language (SDL) [5]. We discovered that SDL is a natural choice for implementing group communication protocol stacks, as its concepts and models nicely fit into our composition approach. Using SDL, the development of the group communication protocol stack became straightforward and the possibility for programming errors was thus greatly reduced. As a result we believe that SDL is much better suited for the implementation of group communication protocol stacks than the general-purpose programming languages such as C, C++ or Java, traditionally used for this purpose. The problem is that these languages are lacking composition features, and had thus to be patched with frameworks such as Cactus or Appia [1, 12] that handle the module composition problem. Due to space constraints, we refer the reader to [4] for further details concerning our prototype implementation in SDL.

The rest of the paper is structured as follows: Section 2

*Research supported partially by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002); partially by the EPFL grant "Semantics-Guided Design and Implementation of Group Communication Middleware"; and partially by the Swiss National Science Foundation under grant number 21-67715.02

briefly overviews group communication. The model based on finite state machines and signal exchanges is presented in Section 3. In Section 4, we show how protocol modules can be composed in this model. We identify the need for interconnection modules, which are presented in Section 5. Section 6 discusses the issue of isolation in signal processing that occurs in a protocol stack. Section 7 overviews relevant related work. Finally, Section 8 concludes the paper.

2. Group Communication

Protocol composition can be applied to any type of communication stack. In this paper, we focus on group communication protocol stacks, which present more complex interactions between the protocol modules than strictly layered stacks (such as TCP/IP, for example).

Group communication (GC) provides abstractions for the development of replicated, fault-tolerant applications such as replicated databases or replicated web servers. It specifies primitives that ensure reliable and totally-ordered communication among a group of processes.

Higher-level abstractions such as Total Ordering of message delivery (*Amcast*) can be built on top of lower level abstractions, such as Reliable Multicast (*Rmcast*) and Consensus. A protocol stack implementing a particular abstraction thus consists of multiple protocols. A more detailed description of these protocols can be found in [8] and [2]. In this paper, we only consider the protocols needed to understand the contribution of the paper. An example atomic multicast protocol stack is depicted in Fig. 1.

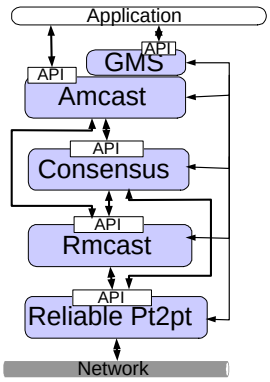


Figure 1. Atomic multicast protocol stack.

3 Model and Definitions

Finite State Machines, Signals and Gates. A *protocol module* implements a particular protocol of the group communication stack. A key idea in our approach is to model each protocol module as a finite state machine. The finite state machines (i.e., the protocol modules) communicate sending *signals* via *gates* (called APIs in Fig. 1). A signal is a notification that a FSM sends to another FSM. It is sent through a gate g_1 of the issuing FSM, to the gate g_2 of the destination FSM. Every FSM has its own thread of control.

We do not need to make any assumptions with respect to the time a signal needs to be received by the destination layer, i.e., the time it spends in a gate queue. However, we assume that signals that are sent via the same gates are received in FIFO order.

Moreover, we assume that within a protocol module an incoming signal is processed atomically. In other words, no interleaved signal processing occurs. Hence, the module reads a signal from one of its input gates, processes it, and writes the resulting signal(s), if any, into the corresponding output gate(s). Only then can the processing of the next input signal start. Our model naturally fits that of SDL, which we describe in the following paragraph.

SDL. SDL (Specification and Description Language) [5] is a widespread, ITU-standardized language in the telecommunications industry. The programming model used by SDL is based on extended finite state machines (FSM) communicating by exchanging signals, without any shared memory.

A SDL system is composed of a set of concurrently-running, finite state machines, that are connected to each other. To facilitate the design and the implementation of large systems, SDL allows the developer to group SDL processes into blocks, which can then be used by higher-level blocks to hierarchically form larger systems. This feature is used for protocol composition in SDL. Each protocol is encapsulated within a block. Blocks are then interconnected to form higher-level protocols. This approach yields a flexible composition model that can be used for strictly hierarchical stacks (such as a TCP stack for example), and also for stacks with more complex dependencies between the different layers, as is often the case with group communication.

4 Protocol Composition

The aim of protocol composition is to compose an entire protocol stack from single protocol modules. Traditionally, this has been done in an ad-hoc fashion for each protocol stack. We propose a new approach where this task is achieved by a piece of code that we call *protocol composer*. The protocol composer creates a particular protocol stack out of protocol modules. The protocol composer also provides the API to the application. This API includes the APIs of all protocol modules that are exposed to the application. Considering the protocol stack in Fig. 1, the protocol composer provides an interface that is identical to the interface of Amcast and GMS (Group Membership Service).

To allow for the initialization of the protocol stack (via the protocol composer) we have added a *configuration module* with the single method *init*. Signals sent to the *init* method are processed by the *configuration module*. Hence, the group communication stack composer consists of a set of protocol modules and a configuration module.

Configuration Module In classical protocol stacks, the protocol modules are responsible for initializing their lower-layer protocol module(s), which, in turn, initialize their lower-layer protocol modules. Therefore, each module has to encapsulate the configuration parameters of the lower-layer protocol modules into its own configuration. However, this creates dependencies between the protocol modules that we want to avoid.

In our composition approach, the initialization of the group communication protocol stack is handled by the configuration module. The configuration module defines the configuration for the particular protocol stack. It parses the configuration parameters and provides initial configuration verification. This verification can detect configuration parameters that conflict between different protocol modules.

The Protocol Stack Composer The protocol composer assembles the protocol modules into a working group communication protocol stack. As each protocol module is represented by a FSM with a set of gates and signals, protocol stacks are composed by connecting these FSMs together, i.e., by matching the signals of corresponding FSMs.

In simple cases, protocol modules can be easily connected since they have the same interface. In the next section we discuss the problem of connecting protocol modules that have different interfaces. This requires a special type of modules, which we call *interconnection modules*.

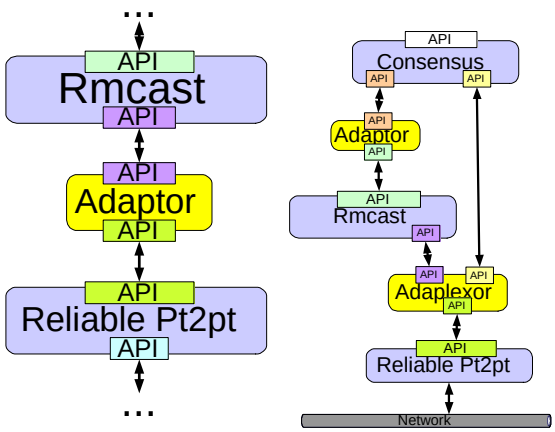
5 Interconnection Modules

In this section, we describe two interconnection modules: *adaptor* and *adaplexor*. We propose the use of these modules as a way to ease the composition of group communication protocol stacks.

5.1 Adaptors

The concept of *adaptor* has been proposed as a design pattern in [6]. It allows two protocol modules to communicate although their interfaces may be (slightly) different.

The adaptor matches the input signals of one module to the output signals of the other module, and vice-versa (see Fig. 2(a)): an adaptor is simply a FSM. The adaptor matching can range from simply connecting corresponding signals, over syntactical matching based on ontologies, towards semantical matching.



(a) The adaptor matches the APIs of two depending modules.

(b) The adaplexor connects Rmcast to Consensus and Amcast

Figure 2. Example of an adaptor and an adaplexor interconnection module.

Consider the common case of an adaptor that only needs to perform minor modifications of the messages sent between two protocol modules. In such a setting, the logic inside the adaptor is often close to trivial, which in turn yields an easy implementation that is not prone to errors.

5.2 Adaplexors

An adaplexor is a more complex interconnection module. Beside providing the same functionality as the adaptor, it also allows signal de-/multiplexing. We thus propose the name *adaplexor* as a combination of *ada*ptor and multi*plexor*. Adaplexors solve the problem that arises when a lower-layer protocol module is used by multiple upper-layer modules (e.g., Reliable Pt2pt or Rmcast in Figure 1).

The Limitation of Hard-Coded Signal Multiplexing and Demultiplexing. The problem can be solved by the developer of such a module, but this requires the developer to be aware of the potential multiple usage of the module (in order to ensure the delivery of signals to the correct upper-layer protocol module). Hence, the module developer would build signal multiplexing and demultiplexing into the module. However, this solution is not ideal, as the module developer has to foresee how the protocol module will be used in future applications. The module should only need to comply with its specification, and not worry about other issues.

Note that another approach is to instantiate such protocol modules multiple times, once for every module that uses it.

However, in this case the state of each instance evolves in an uncoordinated manner. This is undesirable most of the time, although there are some particular cases where this may work. Moreover, multiple instances do not solve the demultiplexing problem, rather, the problem is just shifted to the module below. Revisiting the example in Figure 1, assume we instantiate Rmcast twice in order to solve a multiplexing problem at a higher level (e.g., Consensus and Amcast). Connecting both instances of Rmcast to the same instance of the Reliable Pt2pt protocol module leads to the same problem at this level. Hence, each instance of Rmcast also needs its own instantiation of Reliable Pt2pt. However, multiple module instances create a large overhead and eventually also result in the allocation of many network resources (e.g., sockets, ports) at the lowest level. Besides, protocols at low levels end up having many instances with no coordination among them.

Flexible Multiplexing using Adaplexors. Adaplexors elegantly address multiplexing and demultiplexing externally to the protocol modules. They demultiplex upcall signals, i.e., signals sent from a lower-layer to an upper-layer protocol module, to the corresponding module (e.g., Rmcast or Consensus, see Fig. 2(b)). Downcall signals, i.e., signals from the upper-layer to the lower-layer protocol module, are multiplexed to the single lower-layer module (e.g., Reliable Pt2pt). For this purpose, every message originating at a higher-level module is tagged with the name of this module. This tag is then used by the adaplexor to route the signal to the corresponding module in the receiving group communication protocol stack. An adaplexor is implemented as a FSM, i.e. as an SDL process encapsulated into a block. It offers the same interface as the corresponding lower and upper interfaces it connects.

6 Signal Isolation in the Protocol Stack

Composing protocol stacks from FSM-based protocol modules leads to interleaved signal processing in the protocol stack. While this may be appropriate for some protocol stack, others require some isolation between the processing of signals. To achieve signal processing isolation within the protocol stack, we propose a novel protocol module, called *isolator*. The isolator relies on well-known algorithms to ensure isolation [7]. The main contribution of this section is the definition of the isolator and mechanisms required in order to allow these algorithms to be applied.

The Problem of Interleaved Signal Processing In our model, we assume that signals are processed atomically within a protocol module (see Section 3). However, this is not the case for the entire protocol stack, where different protocol modules can each process a signal concurrently. Indeed, a particular signal may need to be processed by a set of protocol modules before any other signal can be processed by any module in the same set. Consider the example (partial) stack consisting of Reliable Pt2pt and Rmcast [14]. Both modules receive signals from the Group Membership Service (GMS) (see Fig. 1), denoted VC_1 and VC_2.[1] Both Rmcast and Reliable Pt2pt must process these signals without processing any other signal in-between. Hence, S_1 followed by S_1' or S_2 followed by S_2' need to be processed either before or after both signals VC_1 and VC_2 (see Fig. 3(a)). We say that S_1 and S_1', S_2 and S_2', and VC_1 and VC_2 run in isolation, respectively [7]. Consequently, we exclude, for instance, the following sequence of signal processing: S_1, VC_2, S_1', VC_1.

The Isolator Module A protocol stack thus needs to provide a mechanism that enforces isolation in the signal processing. The most stringent mechanism only allows a single signal to be processed at any time in both Reliable Pt2pt and Rmcast. Revisiting the example in Fig. 3(a), only one single signal can be processed within the gray area. In other words, the signals are processed in a strictly sequential order (here, we consider VC_1 and VC_2 as *one* signal). Clearly, the strictly sequential order is not needed in all cases, as it results in reduced performance. Rather, signal processing can be interleaved as long as the outcome corresponds to the result of some sequential processing [7, 14].

It is instrumental for any isolation mechanism to know when the signal processing in a protocol module has terminated. Other approaches have thus built the isolation mechanism into the runtime environment, where they have access to the execution threads that process the signals [14]. In our model (FSM with signal exchange), we do not have access to the runtime system and isolation thus needs to be ensured based only on the information gained from the input signals to and output signals from the protocol modules. We thus propose a novel protocol module, called *isolator*, that is added to the protocol stack. Fig. 3(b) shows the isolator for the Rmcast and Reliable Pt2pt modules. All input and output signals to and from these two protocol modules are routed through the isolator. By controlling the module's input and output signal, the isolator enforces signal processing isolation within a protocol stack. For this purpose, it can use any of the isolation approaches to general transaction processing proposed in [7] and also in [14].

However, it is not always possible to clearly determine when an input signal processing is terminated. In some protocol modules an input signal not always generates an output signal, or may generate multiple output signals. Indeed, assume that protocol module Rmcast *consumes* some

[1] These signals indicate view changes and convey information about processes that need to be added or removed from the group. Please see [4] for a more detailed explanation.

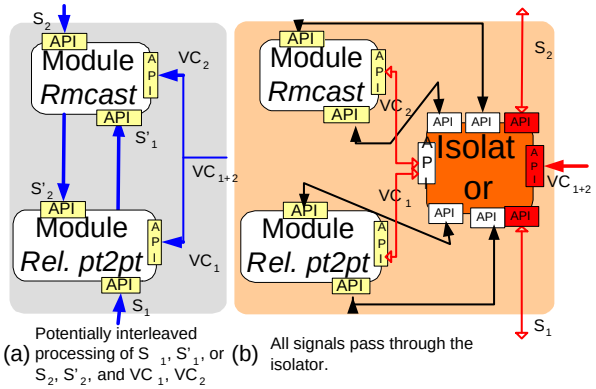


Figure 3. The isolator module for the Rmcast and Reliable Pt2pt modules.

signal, i.e., an input signal does not trigger a corresponding output signal. If this occurs from time to time, the isolator cannot know when the processing of an input signal is terminated and thus when to process the next signal. This is a consequence of the assumption that no bounds can be placed on the time a signal spends in the signal queue. Hence, isolation cannot be ensured without requiring such modules to explicitly notify the termination of the input signal processing.

The following two approaches are possible: (1) positive termination signal or (2) negative termination signal. In the *positive termination signal* approach, every module sends a termination signal when it has finished processing an input signal. The termination signal contains the ID of the corresponding input signal, and the IDs of the resulting output signals. However, this approach has a high overhead, as for every module and every input signal, a corresponding termination signal is generated.

In the *negative termination signal* approach, a termination signal is only generated in components that potentially consume a signal. Hence, each module specifies a so-called *normal* behavior, i.e., it indicates how many output signals are usually generated from a corresponding input signal. If in some particular cases, not as many signals are generated, then a termination message is generated instead of the missing output signal(s).

Clearly, this approach is not applicable if more signals are generated than in the so-called normal behavior. In such a case, the normal behavior should be defined as corresponding to the highest number of output signals.

Both approaches require the collaboration of the protocol modules and thus impose restraints on the developers of these modules. Protocol modules that do not comply with these requirements cannot be integrated into a protocol stack that supports isolation. This limits to a certain extent the applicability of third-party protocol modules and the flexibility of the protocol stack.

7 Related Work

General-purpose languages such as Java or C are lacking composition features, and had thus to be patched with composition frameworks [12, 9, 15, 13, 11]. These frameworks offer abstractions to bridge the gap between the group communication system model (message exchange) and the functionality offered by general-purpose languages (function calls).

The Appia framework [12] reuses and extends the protocol composition framework designed for Ensemble [9, 10]. Appia and Ensemble provide hierarchical protocol composition. Protocols interact by means of events, which play the role of signals in these frameworks. Events traverse a number of protocols following a route defined at event creation time. In contrast, as we have shown in previous sections, group communication protocols use mainly point-to-point events, i.e., events that do not traverse any protocol: they are created at a protocol and disposed of at the first protocol they reach. As a result, group communication protocols implemented in Appia do not profit from its complex mechanisms that route events across several layers.

An important feature of Appia is the possibility to have several possible predefined routes for events, called *channels*. Indeed, having several Appia channels allows event multiplexing in a similar way as adaplexors proposed here. However, this channel-based technique is not as flexible as adaplexors, since an event sets its channel (i.e., its route) at the time it is first created (or injected from the network) and the event cannot change its route later on.

Cactus [15], extends the x-kernel [13] protocol framework to allow for a finer-grain level of composition. In Cactus, the internal structure of an x-kernel protocol consists of the composition of several protocols (called *micro-protocols* in [15]). Similar to our model, these protocols are event-driven[2] and their composition is not hierarchical, allowing them to directly interact without artificial restrictions imposed by procotol stack hierarchy. Cactus allows several event handlers to be bound to the same event so that all these handlers are executed upon occurrence of this event. Although this interaction pattern is simpler than that provided by Appia, it is still more complex than the one needed by group communication protocols, in which (point-to-point) events do not need to execute more than one handler.

In the paper, we have presented some important compositional problems and proposed adaptors and adaplexors as solutions for them. Appia and Cactus do not provide solutions for these compositional problems. Hence, when

[2] *Events* in Cactus are similar to *events* in Appia and *signals* described in this paper.

developers of protocol stacks are confronted to these compositional problems, they need to implement adaptors or adaplexors as ad-hoc protocols [11].

The problem of isolation in the context of transactions is well-studied [7]. However, a transaction starts with a *begin transaction* and ends with an *end transaction*. Consequently, the scope of the transaction is well-known and its termination is a clearly specified event (the *end transaction* command). In our composition, this is not the case. In Appia and Ensemble, isolation is trivially achieved. Only one thread executes all events so that at most one event handler is executed among all protocols at a given time. This thread executes an event and all its resulting events to completion before executing the following event that comes from the network/application.

In Cactus, when a protocol needs to trigger a new event, there are two ways to do it. Synchronous event triggering (called *invoke*) blocks until the framework has finished executing all event handlers bound to the triggered event. Hence, it does not increase concurrency. In contrast, asynchronous event triggering (called *raise*) spawns a concurrent computation and returns immediately after the call. However, the implementation of concurrency control is left to the protocol developer, who has to do it by means of standard operating system synchronization mechanisms (locks, semaphores, monitors, etc).

Isolation in the context of group communication stacks has been discussed in [14], where an extension to the Java programming language is proposed. The paper assumes that the end of an event execution is explicitly known, i.e., by having access to all signal queues. In our approach, we do not assume access to the signal queues, and thus our approach is more general. We do, however, rely on the techniques proposed in [14] and [7] to achieve isolation; the specific problem addressed here is to determine the termination of the event processing.

8. Conclusion

We have presented a flexible composition approach for communication protocols. In this approach, components are finite state machines communicating via signals. We have introduced three interconnection modules: *adaptors*, *adaplexors* and *isolators*. *Adaptors* match module interfaces that otherwise would not be compatible. *Adaplexors* allow several modules to interact with the same lower-lever module that was designed without this multiplicity in mind. We also showed how isolation among signals can be achieved with *isolators*. Finally, we have validated our architectural concepts by implementing a prototype group communication stack [4] using the SDL language.

References

[1] N.T. Bhatti and R.D. Schlichting. A system for constructing configurable high-level protocols. In *SIGCOMM*, pages 138–150, 1995.

[2] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J ACM*, 43(2):225–267, March 1996.

[3] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, May 2001.

[4] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. Technical report, IC/2004/63. École Polytechnique Fédérale de Lausanne, July 2004.

[5] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-Oriented Language For Communicating Systems*. Prentrice Hall, Harlow, England, 1997.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[8] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[9] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 8, 1998.

[10] J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble layers. In R. Cleaveland, editor, *5th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 119–133. Springer Verlag, March 1999.

[11] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia *vs.* cactus. In *Proc. of 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, October 2003.

[12] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS'01)*, pages 707–710, Phoenix, Arizona, USA, April 2001.

[13] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The X-kernel: A platform for accessing Internet resources. *Computer*, 23(5):23–33, May 1990.

[14] P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: Framework for synchronization augmented microprotocol approach. In *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fee, US, 2004.

[15] G.T. Wong, M.A. Hiltunen, and R.D. Schlichting. CTP: A configurable and extensible transport protocol. In *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001), Anchorage, Alaska*, April 2001.