Service Interface: A New Abstraction for Implementing and Composing Protocols*

Olivier Rütti Paweł T. Wojciechowski André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{Olivier.Rutti, Pawel.Wojciechowski, Andre.Schiper}@epfl.ch

ABSTRACT

In this paper we compare two approaches to the design of *protocol frameworks* – tools for implementing modular network protocols. The most common approach uses events as the main abstraction for a local interaction between protocol modules. We argue that an alternative approach, that is based on service abstraction, is more suitable for expressing modular protocols. It also facilitates advanced features in the design of protocols, such as dynamic update of distributed protocols. We then describe an experimental implementation of a service-based protocol framework in Java.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Applications

Keywords

protocol frameworks, modularity, dynamic protocol replacement

1. INTRODUCTION

Protocol frameworks, such Cactus [5, 2], Appia [1, 16], Ensemble [12, 17], Eva [3], SDL [8] and Neko[6, 20], are programming tools for developing modular network protocols. They allow complex protocols to be implemented by decomposing them into several modules cooperating together. This approach facilitates code reuse and customization of distributed protocols in order to fit the needs of different applications. Moreover, protocol modules can be plugged in to the system dynamically. All these features of protocol frameworks make them an interesting enabling technology for implementing adaptable systems [14] - an important class of applications.

*Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

SAC'06 April 23-27, 2006, Dijon, France

Most protocol frameworks are based on events (all frameworks cited above are based on this abstraction). Events are used for asynchronous communication between different modules on the same machine. However, the use of events raises some problems [4, 13]. For instance, the composition of modules may require connectors to route events, which introduces burden for a protocol composer [4]. Protocol frameworks such as Appia and Eva extend the event-based approach with channels. However, in our opinion, this solution is not satisfactory since composition of complex protocol stacks becomes more difficult.

In this paper, we propose a new approach for building modular protocols, that is based on a service abstraction. We compare this new approach with the common, event-based approach. We show that protocol frameworks based on services have several advantages, e.g. allow for a fairly straightforward protocol composition, clear implementation, and better support of dynamic replacement of distributed protocols. To validate our claims, we have implemented SAMOA – an experimental protocol framework that is purely based on the service-based approach to module composition and implementation. The framework allowed us to compare the service- and event-based implementations of an adaptive group communication middleware.

The paper is organized as follows. Section 2 defines general notions. Section 3 presents the main characteristics of event-based frameworks, and features that are distinct for each framework. Section 4 describes our new approach, which is based on service abstraction. Section 5 discusses the advantages of a service-based protocol framework compared to an event-based protocol framework. The description of our experimental implementation is presented in Section 6. Finally, we conclude in Section 7.

2. PROTOCOL FRAMEWORKS

In this section, we describe notions that are common to all protocol frameworks.

Protocols and Protocol Modules. A *protocol* is a distributed algorithm that solves a specific problem in a distributed system, e.g. a TCP protocol solves the reliable channel problem. A protocol is implemented as a set of identical *protocol modules* located on different machines.

Protocol Stacks. A *stack* is a set of protocol modules (of different protocols) that are located on the same machine. Note that, despite its name, a stack is not strictly layered,

i.e. a protocol module can interact with all other protocol modules in the same stack, not only with the protocol modules directly above and below. In the remainder of this paper, we use the terms *machine* and *stack* interchangeably.

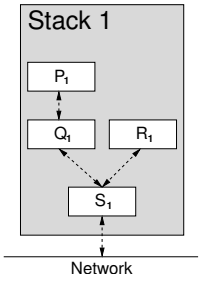

Figure 1: Example of a protocol stack.

In Figure 1, we show an example protocol stack. We represent protocol modules by capital letters indexed with a natural number, e.g. P_1, Q_1, R_1 and S_1. We write P_i to denote the protocol module of a protocol P in stack i. We use this notation throughout the paper. Modules are represented as white boxes. Arrows show module interactions. For instance, protocol module P_1 interacts with the protocol module Q_1 and conversely (See Fig. 1).

Protocol Module Interactions. Below, we define the different kinds of interaction between protocol modules.

- **Requests** are issued by protocol modules. A request by a protocol module P_i is an asynchronous call by P_i of another protocol module.
- **Replies** are the results of a request. A single request can generate several replies. Only protocol modules belonging to the same protocol as the module that has issued the request are concerned by the corresponding replies. For example, a request by P_i generates replies that concern only protocol modules P_j.
- **Notifications** can be used by a protocol module to inform (possibly many) protocol modules in the same stack about the occurrence of a specific event. Notifications may also be the results of a request.

3. EVENT-BASED PROTOCOL FRAMEWORK DESIGN

Most existing protocol frameworks are event-based. Examples are Cactus [5, 2], Appia [1, 16] and Ensemble [12, 17]. In this section, we define the notion of an event in protocol frameworks. We also explain how protocol modules are structured in event-based frameworks.

Events. An *event* is a special object for indirect communication between protocol modules in the same stack. Events may transport some information, e.g. a network message or some other data. With events, the communication is indirect, i.e. a protocol module that *triggers* an event is not aware of the module(s) that *handle* the event. Events enable one-to-many communication within a protocol stack. Triggering an event can be done either synchronously or asynchronously. In the former case, the thread that triggers an event e is blocked until all protocol modules that handle e have terminated handling of event e. In the latter case, the thread that triggers the event is not blocked.

Protocol Modules. In event-based protocol frameworks, a protocol module consists of a set of handlers. Each handler is dedicated to handling of a specific event. Handlers of the same protocol module may share data. Handlers can be dynamically *bound* to events. Handlers can also be *unbound* dynamically. Upon triggering some event e, all handlers bound to e are executed. If no handler is bound, the behavior is usually unspecified.

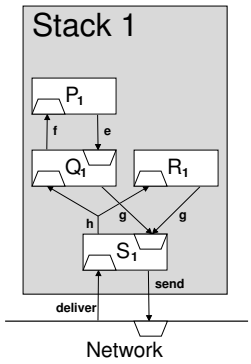

Figure 2: Example of an event-based protocol stack.

In Figure 2, we show an example of an event-based stack. Events are represented by small letters, e.g. e, f, ... The fact that a protocol module can trigger an event is represented by an arrow starting from the module. A white trapezoid inside a module box represents a handler defined by the protocol module. To mark that some handler is bound to event e, we use an arrow pointing to the handler (the label on the arrow represents the event e). For example, the protocol module P_1 triggers event e and handles event f (see Fig. 2). Note that the network is represented as a special protocol module that handles the send event (to send a message to another machine) and triggers the deliver event (upon receipt of a message from another machine).

Specific Features. Some protocol frameworks have unique features. Below, we present the features that influence composition and implementation of protocol modules.

In Cactus [5, 2], the programmer can give a priority number to a handler upon binding it to an event. When an event is triggered, all handlers are executed following the order of priority. A handler h is also able to *cancel* the execution of an event trigger: all handlers that should be executed after h according to the priority are not executed.

Appia [1, 16] and Eva [3] introduce the notion of channels. *Channels* allow to build routes of events in protocol stacks. Each protocol module has to *subscribe* to one or many channels. All events are triggered by specifying a channel they belong to. When a protocol module triggers an event e specifying channel c, all handlers bound to e that are part of a protocol that subscribes to c are executed (in the order prescribed by the definition of channel c).

4. SERVICE-BASED PROTOCOL FRAMEWORK

In this section, we describe our new approach for implementing and composing protocols that is based on services.

We show in Section 5 the advantages of service-based protocol frameworks over event-based protocol frameworks.

Service Interface. In our service-based framework, protocol modules in the same stack communicate through objects called *service interfaces*. Requests, replies and notifications are all issued to service interfaces.

Protocol Modules. A protocol module is a set of *executers*, *listeners* and *interceptors*.

Executers handle requests. An *executer* can be dynamically *bound* to a service interface. It can be later *unbound*. A request issued to a service interface *si* leads to the execution of the executer bound to *si*. If no executer is bound to *si*, the request is delayed until some executer is bound to *si*. Contrary to events, at most one executer at any time can be bound to a service interface on every machine.

Listeners handle replies and notifications. A listener can be dynamically *bound* and *unbound* to/from a service interface *si*. A *notification* issued to a service interface *si* is handled by all listeners bound to *si* in the local stack. A *reply* issued to a service interface is handled by one single listener. To ensure that one single listener handles a reply, a module P_i has to identify each time it issues a request, the listener to handle the possible reply. If the request and the reply occur respectively, in stack i and in stack j, the service interface *si* on i communicates to the service interface si' on j the listener that must handle the reply. If the listener that must handle the reply does not exist, the reply is delayed until the listener is created.

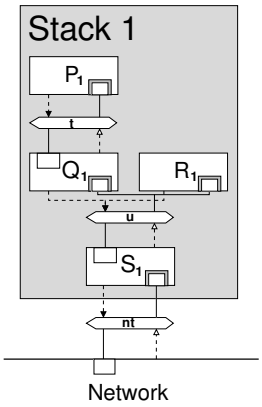

Figure 3: Example of a service-based protocol stack.

In Figure 3, we show an example of a service-based stack. We denote a service interface by a small letter (e.g. t, u and nt) in a hexagonal box. The fact that a module P_i can generate a request to a service interface *si* is represented by a dashed black arrow going from P_i to *si*. Similarly, a dashed white arrow going from module P_i to service interface *si* represents the fact that P_i can generate a reply or a notification to *si*. We represent executers with white boxes inside protocol modules, listeners with white boxes with a gray border. A connecting line between a service interface *si* and an executer *e* (resp. a listener *l*) shows that *e* (resp. *l*) is bound to *si*.

In Figure 3, module Q_1 contains an executer bound to service interface t and a listener bound to service interface u. Module Q_1 can generate replies and notifications to service interface t and request to service interface u. Note that the service interface nt allows to access the network.

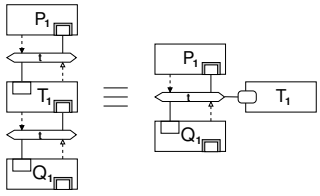

Figure 4: Execution of protocol interactions with interceptors.

An *interceptor* plays a special rôle. Similarly to executers, interceptors can be dynamically bound or unbound to a service interface. They are activated each time a request, a reply or a notification is issued to the service interface they are bound to. This is illustrated in Figure 4. In the right part of the figure, the interceptor of the protocol module T_1 is represented by a rounded box. The interceptor is bound to service interface t. The left part of the figure shows that an interceptor can be seen as an executer plus a listener. When P_1 issues a request *req* to the service interface t, the executer-interceptor of T_1 is executed. Then, module T_1 may forward a request req' to the service interface t, where we can have $req \neq req'$[1]. When module Q_1 issues a reply or a notification, a similar mechanism is used, except that this time the listener-interceptor of T_1 is executed. Note that a protocol module T_i that has an interceptor bound to a service interface is able to modify requests, replies and notifications.

Upon requests, if several interceptors are bound to the same service interface, they are executed in the order of binding. Upon replies and notifications, the order is reversed.

5. ADVANTAGES OF SERVICE-BASED PROTOCOL FRAMEWORK DESIGN

We show in this section the advantages of service-based protocol frameworks over event-based protocol frameworks. We structure our discussion in three parts. Firstly, we present how protocol interactions are modeled in each of the protocol frameworks. Then, we discuss the composition of protocol modules in each of these frameworks. Finally, we present the problem of dynamic protocol replacement and the advantages of service interfaces in order to implement it. The discussion is summarized in Table 1.

5.1 Protocol Module Interactions

A natural model of protocol interactions (as presented in Section 2) facilitates the implementation of protocol modules. For each protocol interaction, we show how it is modeled in both frameworks. We also explain that an inaccurate model may lead to problems.

Requests. In service-based frameworks, a request is generated to a service interface. Each request is handled by at most one executer, since we allow only one executer to be bound to a service interface at any time. On the other hand, in event-based frameworks, a protocol module emulates a request by triggering an event. There is no guarantee

[1]The two service interfaces t in the left part of Figure 4 represent the *same* service interface t. The duplication is only to make the figure readable.

that this event is bound to only one handler, which may lead to programming errors.

Replies. When a protocol module generates a reply in a service-based framework, only the correct listener (identified at the time the corresponding request was issued) is executed. This ensures that a request issued by some protocol module Q_i, leads to replies handled by protocol modules Q_j (i.e. protocol modules of the same protocol).

This is not the case in event-based frameworks, as we now show. Consider protocol module Q_1 in Figure 2 that triggers event g to emulate a request. Module S_1 handles the request. When modules S_i triggers event h to emulate a reply (remember that a reply can occur in many stacks), both modules Q_i and R_i will handle the reply (they both contain a handler bound to h). This behavior is not correct: only protocol modules Q_i should handle the reply. Moreover, as modules R_i are not necessarily implemented to interact with modules Q_i, this behavior may lead to errors.

Solutions to solve this problem exist. However, they introduce an unnecessary burden on the protocol programmers and the stack composer. For instance, channels allow to route events to ensure that modules handle only events concerning them. However, the protocol programmer must take channels into account when implementing protocols. Moreover, the composition of complex stacks becomes more difficult due to the fact that the composer has to create many channels to ensure that modules handle events correctly. An addition of special protocol modules (named *connectors*) for routing events is also not satisfactory, since it requires additional work from the composer and introduces overhead.

Notifications. Contrary to requests and replies, notifications are well modeled in event-based frameworks. The reason is that notifications correspond to the one-to-many communication scheme provided by events. In service-based frameworks, notifications are also well modeled. When a module generates a notification to a service interface *si*, all listeners bound to *s* are executed. Note that in this case, service interfaces provide the same pattern of communication as events.

5.2 Protocol Module Composition

Replies (and sometimes notifications) are the results of a request. Thus, there is a semantic link between them. The composer of protocol modules must preserve this link in order to compose correct stacks. We explain now that service based frameworks provide a mechanism to preserve this link, while in event-based frameworks, the lack of such mechanism leads to error-prone composition.

In service-based frameworks, requests, replies and notifications are issued to a service interface. Thus, a service interface introduces a link between these interactions. To compose a correct stack, the composer has to bound a listener to service interface *si* for each module that issues a request to *si*. The same must be done for one executer that is part of a module that issues replies or notifications. Applying this simple methodology ensures that every request issued to a service interface *si* eventually results in several replies or notifications issued to the same service interface *si*.

In event-based frameworks, all protocol interactions are issued through different events: there is no explicit link between an event triggered upon requests and an event triggered upon the corresponding replies. Thus, the composer of a protocol stack must know the meaning of each event in order to preserve the semantic link between replies (and notifications) and requests. Moreover, nothing prevents from binding a handler that should handle a request to an event used to issue a reply. Note that these problems can be partially solved by typing events and handlers. However, it does not prevent from errors if there are several instances of the same event type.

Note that protocol composition is clearer in the protocol frameworks that are based on services, rather than on events. The reason is that several events that are used to model different protocol interactions can be modeled by a single service interface.

5.3 Dynamic Replacement of Protocols

Dynamic replacement of protocols consists in switching on-the-fly between protocols that solve the same problem. Replacement of a protocol P by a new protocol $newP$ means that a protocol module P_i is replaced by $newP_i$ in every stack i. This replacement is problematic since the local replacements (within stacks) must be synchronized in order to guarantee protocol correctness [21, 18].

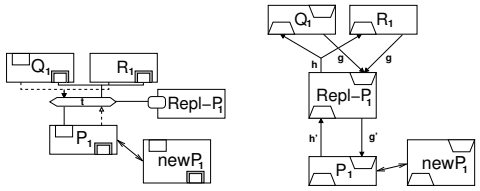

Figure 5: Dynamic replacement of protocol P.

For the synchronization algorithms to work, module interactions are *intercepted* in order to detect a time when P_i should be replaced by $newP_i$. (Other solutions, e.g. in [11], are more complex.) In Fig. 5, we show how this interception can be implemented in protocol frameworks that are based on services (in the left part of the figure) and events (in the right part of the figure). The two-sided arrows point to the protocol modules P_1 and $newP_1$ that are switched.

It can be seen that the approach that uses the Service Interface mechanism has advantages. The intercepting module *Repl-*P_1 has an interceptor bound to service interface t that intercepts every request handled by modules P_1 and all replies and notifications issued by P_1. The code of the module P_1 can therefore remain unchanged.

In event-based frameworks, the solution is to add an intermediate module *Repl-*P_1 that intercepts the requests issued to P_1 and also the replies and notifications issued by P_1. Although this *ad-hoc* solution may seem similar to the service-based approach, there is an important difference. The event-based solution requires to modify the module P_1 since instead of handling event g and triggering event h, P_1 must now handle different events g' and h' (see Fig. 5).

6. IMPLEMENTATION

We have implemented an experimental service-based protocol framework (called SAMOA) [7]. Our implementation is light-weight: it consists of approximately 1200 lines of code in Java 1.5 (with generics).

In this section, we describe the main two classes of our implementation: *Service* (encoding the Service Interface) and

	service-based	event-based
Protocol Interaction	an adequate representation	an inadequate representation
Protocol Composition	clear and safe	complex and error-prone
Dynamic Replacement	an integrated mechanism	*ad-hoc* solutions

Table 1: Service-based *vs.* event-based

Protocol (encoding protocol modules). Finally, we present an example protocol stack that we have implemented to validate the service-based approach.

The Service Class. A *Service* object is characterized by the arguments of requests and the arguments of responses. A response is either a reply or a notification. A special argument, called *message*, determines the kind of interactions modeled by the response. A message represents a piece of information sent over the network. When a protocol module issues a request, it can give a message as an argument. The message can specify the listener that must handle the reply. When a protocol module issues a response to a service interface, a reply is issued if one of the arguments of the response is a message specifying a listener. Otherwise, a notification is issued.

Executers, listeners and interceptors are encoded as inner-classes of the *Service* class. This allows to provide *type-safe* protocol interactions. For instance, executers can only be bound to the *Service* object, they belong to. Thus, the parameters passed to requests (that are verified statically) always correspond to the parameters accepted by the corresponding executers.

The type of a *Service* object is determined by the type of the arguments of requests and responses. A *Service* object t is *compatible* with another *Service* object s if the type of the arguments of requests (and responses) of t is a subtype of the arguments of requests (and responses) of s. In practice, e.g., if a protocol module P_i can issue a request to a protocol UDP (compatible with TCP), then it may also issue a request to TCP due to the subtyping relation on parameters of communicating modules.

The Protocol Class. A *Protocol* object consists of three sets of components, one set for each component type (a listener, an executer, and an interceptor). *Protocol* objects are characterized by names to retrieve them easily. Moreover, we have added some features to bind and unbind all executers or interceptors to/from the corresponding *Service* objects. *Protocol* objects can be loaded to a stack dynamically. All these features made it easy to implement dynamic replacement of network protocols.

Protocol Stack Implementation. To validate our ideas, we have developed an Adaptive Group Communication (AGC) middleware, adopting both the service- and the event-based approach. Fig. 6 shows the corresponding stacks of the AGC middleware. Both stacks allow the *Consensus* and *Atomic Broadcast* protocols to be dynamically updated. Two modes of failure are supported: crash-stop and crash-recovery.

The architecture of our middleware, shown in Fig. 6, builds on the group communication stack described in [15]. The *UDP* and *RP2P* modules provide respectively, unreliable and reliable point-to-point transport. The *FD* module

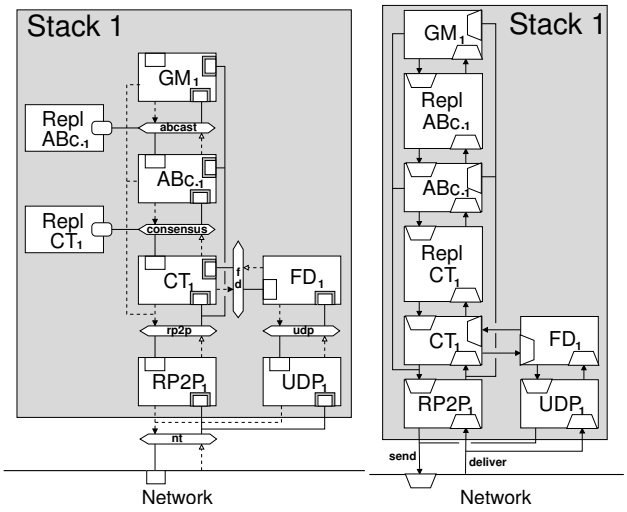

Figure 6: Adaptive Group Communication Middleware: service-based (left) *vs.* event-based (right).

implements a *failure detector*; we assume that it ensures the properties of the $\diamond S$ failure detector [9]. The CT module provides a *distributed consensus* service using the Chandra-Toueg algorithm [10]. The *ABc.* module implements *atomic broadcast* – a group communication primitive that delivers messages to all processes in the same order. The GM module provides a *group membership* service that maintains consistent membership data among group members (see [19] for details). The *Repl ABc.* and the *Repl CT* modules implement the replacement algorithms [18] for, respectively, the *ABc.* and the *CT* protocol modules. Note that each arrow in the event-based architecture represents an event. We do not name events in the figure for readability.

The left stack in Figure 6 shows the implementation of AGC with our service-based framework. The right stack shows the same implementation with an event-based framework.

Performance Evaluation. To evaluate the overhead of service interfaces, we compared performance of the service- and event-based implementations of the AGC middleware. The latter implementation of AGC uses the Cactus protocol framework [5, 2].

In our experiment, we have compared the *average latency* of *Atomic Broadcast* (*ABcast*), which is defined as follows. Consider a message m sent using ABcast. We denote by $t_i(m)$ the time between the moment of sending m and the moment of delivering m on a machine (stack) i. We define the *average latency* of m as the average of $t_i(m)$ for all machines (stacks) i within a group of stacks.

Performance tests have been made using a cluster of PCs running Red Hat Linux 7.2, where each PC has a Pentium III 766 MHz processor and 128MB of RAM. All PCs are interconnected by a 100 Base-TX duplex Ethernet hub. Our experiment has involved 7 machines (stacks) that ABcast messages of 4Mb under a constant load, where a *load* is a number of messages per second. In Figure 7, we show the results of our experiment for different loads. Latencies are shown on the vertical axis, while message loads are shown on the horizontal axis. The solid line shows the results obtained with our service-based framework. The dashed line

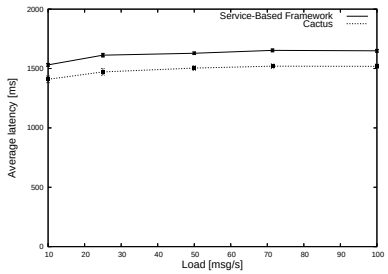

Figure 7: Comparison between our service-based framework and Cactus.

shows the results obtained with the Cactus framework. The overhead of the service-based framework is approximately 10%. This can be explained as follows. Firstly, the service-based framework provides a higher level abstraction, which has a small cost. Secondly, the AGC middleware was initially implemented and optimized for the event-based Cactus framework. However, it is possible to optimize the AGC middleware for the service-based framework.

7. CONCLUSION

In the paper, we have proposed a new approach to the protocol composition that is based on the notion of *Service Interface*, instead of events. We believe that the service-based framework has several advantages over event-based frameworks. It allows us to: (1) model accurately protocol interactions, (2) reduce the risk of errors during the composition phase, and (3) simply implement dynamic protocol updates. A prototype implementation allowed us to validate our ideas.

8. REFERENCES

[1] *The Appia project*. Documentation available electronically at `http://appia.di.fc.ul.pt/`.

[2] 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.

[3] Francisco Vilar Brasileiro, Fabíola Greve, Frederic Tronel, Michel Hurfin, and Jean-Pierre Le Narzul. Eva: An event-based framework for developing specialized communication protocols. In *Proceedings of the 1st IEEE International Symposium on Network Computing and Applications (NCA '01)*, 2001.

[4] Daniel C. Bünzli, Sergio Mena, and Uwe Nestmann. Protocol composition frameworks. A header-driven model. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA '05)*, July 2005.

[5] *The Cactus project*. Documentation available electronically at `http://www.cs.arizona.edu/cactus/`.

[6] *The Neko project*. Documentation available electronically at `http://lsrwww.epfl.ch/neko/`.

[7] *The SAMOA project*. Documentation available electronically at `http://lsrwww.epfl.ch/samoa/`.

[8] *The SDL project*. Documentation available electronically at `http://www.sdl-forum.org/SDL/`.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[11] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st IEEE International Conference on Distributed Computing System (ICDCS '01)*, April 2001.

[12] *The Ensemble project*. Documentation available electronically at `http://www.cs.cornell.edu/Info/Projects/Ensemble/`.

[13] Richard Ekwall, Sergio Mena, Stefan Pleisch, and André Schiper. Towards flexible finite-state-machine-based protocol composition. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA '04)*, August 2004.

[14] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[15] Sergio Mena, André Schiper, and Paweł T. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware '03)*, LNCS 2672, June 2003.

[16] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS '01)*, April 2001.

[17] Ohad Rodeh, Kenneth P. Birman, Mark Hayden, Zhen Xiao, and Danny Dolev. The architecture and performance of security protocols in the Ensemble group communication system. Technical Report TR-98-1703, Computer Science Department, Cornell University, September 1998.

[18] Olivier Rütti, Paweł T. Wojciechowski, and André Schiper. Dynamic update of distributed agreement protocols. TR IC-2005-12, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), March 2005.

[19] André Schiper. Dynamic Group Communication. Technical Report IC-2003-27, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), April 2003. To appear in ACM Distributed Computing.

[20] Péter Urbán, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proceedings of the 15th International Conference on Information Networking (ICOIN '01)*, February 2001.

[21] Paweł T. Wojciechowski and Olivier Rütti. On correctness of dynamic protocol update. In *Proceedings of the 7th IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '05)*, LNCS 3535. Springer, June 2005.