Structural and Algorithmic Issues of Dynamic Protocol Update

Olivier Rütti[1], Paweł T. Wojciechowski[2], André Schiper[1]

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
1015 Lausanne, Switzerland
{Olivier.Rutti, Andre.Schiper}@epfl.ch

[2]Poznań University of Technology
Institute of Computing Science
60-965 Poznań, Poland
ptw@cs.put.poznan.pl

Abstract

In this paper, we study dynamic protocol update (DPU). Contrary to local code updates on-the-fly, DPU requires global coordination of local code replacements. We propose a novel solution to DPU. The key idea is to add a level of indirection between the service callers and the service provider. This indirection level facilitates an implementation of simple and efficient algorithms for DPU. For example, we describe an experimental implementation of adaptive group communication middleware. It can switch between different atomic broadcast protocols on-the-fly. All middleware protocols, including those that depend on the updated protocols, provide service correctly and with negligible delay while the global update takes places. The switching algorithm introduces very low overhead that we illustrate by showing example measurement results.

1 Introduction

Recent years have seen a growing interest in programming tools for *adaptable systems*, i.e., systems that can be reconfigured and adapted to new environments or changing user requirements (see [12] for examples of such tools and techniques). In this paper, we propose algorithmic tools for adaptable group communication middleware [20, 6]. They allow software modules or components of the middleware to be replaced on-the-fly without service interruption. The benefit is a decrease of software upgrade and maintenance costs in systems that must run non-stop. We believe that our work will be useful for practitioners and system developers.

Our work focuses on the problem of dynamic update of *distributed protocols*, i.e. protocols (e.g., group communication protocols) that are implemented by several identical modules located on different machines. The dynamic protocol update (DPU) requires that all local updates must be (eventually) consistently performed on all machines. To avoid interference between concurrent versions of the protocol, some global synchronization of local updates may be required. We would like to minimize the impact of this global synchronization, so that DPU efficiency and scalability is not degraded. For instance, it is desirable that the whole system is not blocked, and remains available while protocols are updated.

Many systems have been developed to allow dynamic adaptation in the context of distributed systems (see [8, 11, 3, 2] among others). However, majority of such systems do not address consistent update of distributed protocols. Examples of existing systems that support DPU will be discussed later in this paper. Moreover, little work exists on the theoretical foundations for DPU. We made some initial step in [21], where we defined a formal mathematical model, and used it to define different levels of synchrony between local updates (different updateable protocols may require different levels).

Liu *et al.* [10] define several meta properties on traces of the send and deliver events. These meta properties must be preserved by each updateable protocol (an example is described in [1]). However, this model of DPU strongly limits the scope of application. On the contrary, we define in this paper two generic correctness properties of dynamically updateable systems: *stack-well-formedness* and *protocol-operationability*. Preserving these properties and some additional correctness

---

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

properties specific to the protocols being replaced during dynamic update, guarantees that the update is transparent to the users of the protocols.

In this paper, we consider two complementary dimensions of DPU: (1) the structural dimension, and (2) the algorithmic dimension. The *structural* dimension of DPU deals with the way a replacement manager is integrated into each protocol stack. The *algorithmic* dimension of DPU deals with the algorithm of the replacement manager. A clever integration of the replacement manager facilitates and simplifies the implementation of DPU algorithms. In particular, the structural dimension of DPU in existing solutions (e.g., [20, 6]) is not satisfactorily addressed. For instance, most of the existing solutions require an explicit interaction between the updateable protocols and the replacement manager, which leads to poor modularity (since the implementation of DPU strongly depends on the updateable protocols). We propose a solution that solves this problem. Moreover, contrary to other solutions where DPU requires to *understand* the updateable protocols, our solution only requires to know the *specification* of the protocols that get replaced.

To validate our ideas, we have implemented support of the dynamic replacement of protocols that satisfy the *atomic broadcast* specification [7]. The choice of this type of protocols was not accidental: atomic broadcast protocols are good representatives of non-trivial distributed algorithms, and so our results (e.g. within the structural dimension) extend to other types of protocols. Moreover, atomic broadcast is considered to be an important building block for group communication middleware systems [13]. Such systems are used for implementing replicated non-stop services. Thus, the solutions presented in this paper can be valuable for developers of highly available non-stop systems.

We have implemented our adaptive group communication middleware using our SAMOA protocol framework [22], and have experimented with switching on-the-fly between different atomic broadcast protocols. In this paper, we present the results of experiments evaluating the impact of dynamic protocol replacement on system performance. The results show that the cost of switching between different protocols is negligible.

The rest of the paper is organized as follows. Section 2 describes the composition model that we use in the paper. Section 3 defines generic correctness properties related to DPU. Section 4 presents structural aspects of our solution for DPU, and compares it with existing solutions. Section 5 describes the replacement algorithm for switching on-the-fly between different atomic broadcast protocols. Section 6 presents performance results, and Section 7 concludes.

2 Model

In this section we introduce a simple model that differentiates *services* (specifications of distributed protocols) from *protocols* (implementations of distributed protocols). In the following sections, we use our model to describe the correctness properties and the implementation of DPU.

Basic definitions We consider *distributed protocols*. Protocols are implemented by a set of identical *modules*, each module running on a different machine (or site). A module describes the exchange of messages across the network, and may contain some local data. The set of all modules located on a machine is called a *protocol stack*.

A protocol P can be seen as the implementation of some *service* p. We say that protocol P *provides* service p on each stack. For example, the protocol *p-atomic-broadcast*, represented by a module *m-atomic-broadcast* on each stack, provides the service *s-atomic-broadcast* on each stack. A protocol providing some service may *require* some other services.

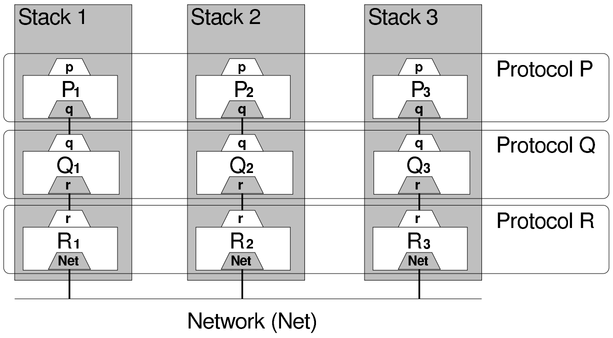

Figure 1. An example protocol architecture.

Figure 1 shows an example system. Protocols are represented with capital letters P, Q and R, and services with small letters p, q and r. We write P_i to denote a module of the protocol P, which is part of stack i ($i = 1, 2, ..$). Modules are illustrated in figures as boxes. Services that are required by a module are named in a gray trapezoid inside the box representing the module. Similarly, services that are provided by a module are named in white trapezoids that are aligned outside the box of the module. For example, module Q_1 provides service q and requires service r (see Fig. 1). Note that the network is also a service (named Net).

Module bindings A module can be dynamically *bound* to a service that it provides. It can be later *unbound*. Unbinding a module does not remove it from the stack. Stacks may contain several modules that provide the same service. At most one module in a stack is bound to a service at a time.

Service calls When we make a service call, the module that is bound to the service is executed. If no module is bound, the service call is blocked until some module is bound to the service.

Service responses Consider a *call* of a service q, which has been made by some module P_i (see Fig. 2). The service q is provided by module Q_i. We define the *response* to this call to be any invocation of a module P_j by Q_j in some stack j ($j = i$ or $j \neq i$) that results from the initial call. If P_j is not currently in stack j, then the invocation made by Q_j is completed when P_j is added to stack j. Note that a module Q_i can respond to a service call even if Q_i has been unbound.

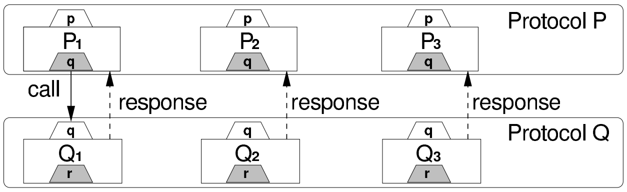

Figure 2. Service calls and responses.

Figure 2 illustrates service calls and responses. The call of a service q made by module P_1 is shown with a solid arrow. Responses to this call are represented with dashed arrows. Note that responses can occur in one or many stacks. We say that P_1 interacts locally with module Q_1 on every call of service q. Responses to the call of service q lead to a remote interaction of P_1 with P_2 and P_3.

Service calls and responses to service calls are the two kinds of interactions between modules. A service call is a local interaction between the service caller and the service provider. A response to a call is an interaction between the service caller and the (local or remote) module that is receiving the response.

3 Generic Dynamic Update Properties

In this section, we define several generic correctness properties of dynamic replacement of distributed protocols. Firstly, we define a property that ensures correct local interactions. We consider two levels of this property: strong and weak. The former one ensures that a service call is never blocked. Preserving the latter level means that a service call may be blocked, but not infinitely.

Strong stack-well-formedness A stack is *strongly well-formed* if and only if whenever a module calls a service, the service is bound to one module.

Weak stack-well-formedness A stack is *weakly well-formed* if and only if whenever a module calls a service, the service is eventually bound to one module.

Stack-well-formedness is a local property. Below we define the protocol-operationability property, which describes remote interactions. It ensures that whenever a service is called, then all possible responses to this call (in non-crashed stacks) are guaranteed to occur. We again consider two levels of this property: strong and weak.

Strong protocol-operationability A protocol P is *strongly operational* in a set of stacks Π, if and only if whenever a module P_i is bound in some stack i, then all non-crashed stacks j in Π contain a module P_j.

Weak protocol-operationability A protocol P is *weakly operational* in a set of stacks Π, if and only if whenever a module P_i is bound in some stack i, then all non-crashed stacks j in Π eventually contain a module P_j.

The strong protocol-operationability implies weak protocol-operationability. With synchronous networks, which impose time requirements on protocol interactions, the strong level of both stack-well-formedness and protocol-operationability must be ensured. In the remainder of the paper, we consider only the weak properties, since we consider asynchronous networks.

4 Structural Aspect of DPU

We describe now our solution to integrate a manager for dynamic protocol replacement. We illustrate our solution with an example group communication middleware. Then, we compare our solution with other existing solutions.

4.1 Our Solution

Description The main idea is to add a *replacement module* that implements a level of indirection between service calls and the protocol that provides the service. The replacement module intercepts service calls and responses to the service calls, so that it can provide synchronization, which is necessary to ensure the DPU correctness properties.

In addition to the generic properties described in Section 3, some additional properties must be satisfied; these properties are specific to the service provided by the modules being updated. The structural aspect of our solution (with interception of service calls and responses) facilitates the implementation of algorithms that ensure the properties specific to the service. Moreover, the interception of service calls and responses makes the algorithm dependent only on the specification of the protocol that gets replaced.

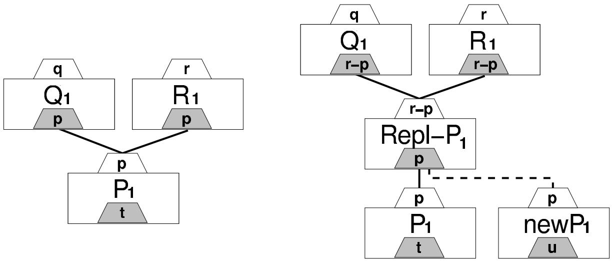

Figure 3. The module composition without a replacement module (left) and with the replacement module *Repl* (right).

Figure 3 shows an example stack without a replacement module (on the left) and with the replacement module *Repl*-P_1 (on the right), where 1 denotes a stack number. The modules *Repl*-P_i are used to replace a protocol P by a protocol *newP*: both provide service p but may require different services. Note that modules *Repl*-P_i *require* service p. Modules Q_1 and R_1 are two modules that may call service p. In the updateable system, the service p is not called directly, but via an interface r-p that is provided by *Repl*-P_1.

On the right of Figure 3, we show the replacement of protocol P_1 by protocol *newP*$_1$. Protocol P_1 is bound to the service p. The dashed lines connecting modules *Repl*-P_1 and *newP*$_1$ shows that *Repl*-P_1 will bind *newP*$_1$ to the service p after having unbound module P_1 from that service.

Example Figure 4 shows the architecture of our adaptive middleware; it builds on the *Fortika* group communication stack described in [13].

- The *UDP* module provides an interface to the UDP (unreliable) protocol.
- The *RP2P* module implements reliable point-to-point communication between distributed processes.
- The *FD* module implements a *failure detector*; we assume that it ensures the properties of the $\Diamond S$ failure detector [4].
- The *CT* module provides a *distributed consensus* service using the Chandra-Toueg $\Diamond S$ consensus algorithm [5] based on a rotating coordinator.
- The *ABcast* module implements *atomic broadcast*, a group communication primitive that delivers messages to all processes in the same order; the module requires the consensus service.
- The *GM* module provides a *group membership* service that maintains consistent membership among all group members; the module requires the atomic broadcast service (see [17] for the details).
- The *Repl* module implements the replacement algorithm dedicated to the atomic broadcast service (see Section 5).

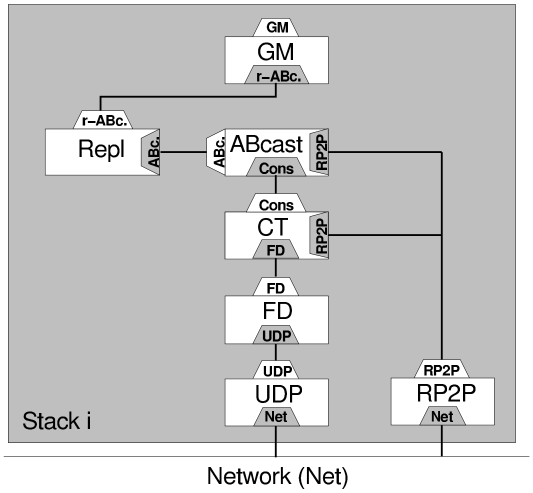

Figure 4. Architecture of the group communication stack.

Note that our ABcast module is not implemented on top of a view synchrony protocol as it is often the case. However, our replacement algorithm is general and works also for atomic broadcast protocols implemented on top of a view synchrony protocol.

4.2 Existing Solutions

Many solutions for DPU exist [20, 6, 18, 9, 15]. However, some of these solutions (e.g. [18] and [9]) are clearly not satisfactory. In [18] the authors propose a solution that uses a centralized manager, which limits its tolerance to failures. On the other hand, the solution proposed in [9] provides facilities to replace only a single module of a protocol.

We present now two example solutions to DPU, which are represented by Maestro [20] and Grace Adaptation [6], and compare them with our approach. An approach described in [15] is similar to Maestro but implemented within the Appia [14] protocol framework.

Maestro [20] Maestro supports only the replacement of complete protocol stacks, i.e. in order to replace a single protocol, the whole stack (containing the protocol) has to be replaced.

The main idea of their solution is to install on each machine a *stack switch module* (SS module). The SS module is in charge to dynamically replace stacks. Its main role is to (1) finalize the local old stack, and (2) coordinate the start of the new stack as soon as possible. In order to finalize the old stack, some protocol modules must be *extended* with a method finalize that properly terminates the protocols. The method finalize is called by the SS module each time a stack replacement is required.

Graceful Adaptation [6] In this solution, each adaptive module implementing some service consists of a Component Adaptor (CA), and several Adaptive-Aware Components (AACs) that provide alternative implementations of the service. Upon a service call, only the AAC component that is *activated* is executed. Only one AAC component can be activated at a time.

The role of the component adaptor is to dynamically switch between the different AAC components, thus changing the algorithm that is used to provide the service. This is done by (1) deactivating the AAC component that is currently activated, and (2) activating a new AAC. Each of these operations is performed by AAC itself. The CA component only coordinates the operations, as follows:

1. The CA asks the old and the new AAC to prepare respectively deactivation and activation.
2. Once all stacks terminate the preparation phase, the CA starts the deactivation of the old AAC.
3. Once the old AAC deactivates itself, it starts the activation of the new AAC.

In order to perform these three steps, the old AAC, the new AAC and the CA communicate with each other. Thus, each AAC must be *extended* in order to be able to communicate with the CA and some other AAC during the replacement procedure.

Note that each AAC in a module m can only use the services required by m. This limits the possible replacements, since AACs that require other services cannot be part of m.

Comparison with our solution Our solution has several advantages over existing solutions due to the way we address the structural aspect of DPU. The main advantage is that our implementation of the dynamic protocol update does not depend on the algorithm of the updateable protocols, but only on the specification of these protocols. In Maestro and Graceful Adaptation, for each dynamic protocol update, the programmer has to *extend* the protocol modules that get updated. In order to extend correctly these modules, the programmer of DPU must have a clear understanding of the algorithms of the protocol modules that get replaced. Moreover, in our solution, the switching algorithm is implemented entirely by the replacement module. Protocol modules are not even aware that the protocol replacement takes place. Our solution is therefore modular in contrast to existing solutions that require to extend *each* updateable module.

Another advantage of our solution is that it is highly flexible. In contrary to Graceful Adaptation, our solution does not limit the possible replacements by imposing any restrictions on the services that a newly added protocol may require. Unlike Maestro, replacement of a single protocol in our system does not require a whole protocol stack to be also replaced.

5 Algorithmic Aspect of the Atomic Broadcast Protocol Replacement

In this section we present the specification ensured by the atomic broadcast protocols. Then, we describe the algorithm for the replacement of Atomic Broadcast (ABcast) protocols. The algorithm is implemented by the *Repl* module presented in Figure 4. Finally, we discuss the advantages of our solution over other algorithmic solutions for DPU. These advantages are the result of an elegant integration of the replacement algorithms in our framework.

5.1 Atomic Broadcast

Atomic broadcast is defined by the two primitives ABcast and Adeliver, that satisfy the following properties [7]:

- *Validity:* If a correct process ABcasts a message m, then it eventually Adelivers m.
- *Uniform agreement:* If a process Adelivers a message m, then all correct processes eventually Adeliver m.
- *Uniform integrity:* For any message m, every process Adelivers m at most once, and only if m was previously ABcast.
- *Uniform total order:* If some process Adelivers message m before it Adelivers message m', then every process Adelivers m' only after it has Adelivered m.

Algorithm 1 Replacement of ABcast: code of stack i.

```
1: Initialisation:
2:   $undelivered \leftarrow \emptyset$        {set of messages not yet Adelivered}
3:   $curABcast \leftarrow$ current ABcast protocol
4:   $seqNumber = 0$                          {sequence number}

5: upon changeABcast($prot$) do
6:   ABcast($newABcast, seqNumber, prot$)

7: upon rABcast($m$) do
8:   $undelivered \leftarrow undelivered \cup m$
9:   ABcast($nil, seqNumber, m$)

10: upon Adeliver($newABcast, sn, prot$) do
11:   $seqNumber \leftarrow seqNumber + 1$
12:   unbind($curABcast$)
13:   create_module($prot$)
14:   $curABcast \leftarrow prot$
15:   for all $m \in undelivered$ do
16:     ABcast($nil, seqNumber, m$)

17: upon Adeliver($nil, sn, m$) do
18:   if ($sn = seqNumber$) then
19:     if ($m \in undelivered$) then
20:       $undelivered \leftarrow undelivered \setminus m$
21:     rAdeliver($m$);

22: procedure create_module($p$)
23:   create $p$
24:   bind $p$
25:   for all $s \in$ services required by p do
26:     if no module is bound to service $s$ in stack $i$ then
27:       find a module $q$ providing service $s$
28:       create_module($q$)
```

5.2 Our Solution

Below we describe an algorithm for replacement of all protocols that satisfy the specification in Section 5.1 (see Algorithm 1 and Figure 4). Then, we prove that it satisfies our generic correctness properties and some additional specific properties.

5.2.1 The Replacement Algorithm

Replacement of the Atomic Broadcast (ABcast) protocol is initiated by the call changeABcast(*prot*), where *prot* is the new ABcast protocol (see Algorithm 1, line 5). This call triggers a call ABcast($newABcast, seqNumber, prot$) (line 6), where *newABcast* indicates the request to replace the existing ABcast protocol, and *seqNumber* identifies the current version of the ABcast protocol. The global variable *seqNumber* is initiated to 0 (line 4) and incremented with every replacement of ABcast.

The lines 7-9 define a call rABcast(m): firstly, the message m is added to the set *undelivered* of undelivered messages, then the call ABcast($nil, seqNumber, m$) is made, where *nil* indicates an ordinary call of the ABcast primitive.

The lines 10-16 implement the Adeliver primitive for the messages with tag *newABcast*, and the lines 17-21 for the messages with tag *nil*, as follows.

If a replacement is requested (lines 10-16), then the *seqNumber* global variable is incremented (line 11), the old module (*curABcast*) is unbound and the new module *prot* (of the new ABcast protocol) is created and bound (lines 12-14). Finally, all undelivered messages are reissued using the new ABcast protocol (lines 15-16).

If no replacement is requested (lines 17-21), then a test is performed (in line 18) to avoid that message m is Adelivered twice: a message with a sequence number corresponding to an older ABcast protocol is discarded, otherwise it is delivered by rDeliver(m) (line 21).

5.2.2 Proof

It is easy to see that the replacement protocol satisfies weak stack-well-formedness and weak protocol-operationability.

Weak stack-well-formedness This property is trivially ensured by the fact that the unbind of line 12 is immediately followed by a new binding triggered by line 13. □

Weak protocol-operationability If a module *newABcast* is created and bound in stack i (line 13), then stack i has Adelivered the message (*newABcast*, *sn*, *prot*) (line 10). Since the uniform agreement property of atomic broadcast ensures that a message Adelivered in a correct stack is also Adelivered by all other correct stacks, all non-crashed stacks eventually create module *newABcast*. □

In addition, we need to prove properties specific to the replacement of atomic broadcast: we need to prove that the properties of atomic broadcast (Sect. 5.1) are satisfied across the replacement protocol (assuming that each ABcast protocol satisfies the properties of Section 5.1).

The first observation is that, since the protocol change is handled by ABcast, the protocol identified by the sequence number *sn* in stack i is the same as the protocol identified with *sn* in stack j. So we can unambiguously identify a protocol by a sequence number *sn*.

Validity Consider a correct process p_i that executes ABcast(m) using protocol *sn* of stack i. Since the ABcast protocol satisfies validity, the only reason for m not to be Adelivered is the replacement of the protocol *sn* by a new protocol $sn' > sn$ (by line 18, m can be discarded). However, if m is discarded by line 18, m is reissued by the new protocol sn' (line 16). By the validity property of the new protocol, m is eventually Adelivered by p_i. $\Box$

Uniform agreement Consider a process p_i that Adelivers m using protocol sn of stack i. Since the ABcast protocol satisfies uniform agreement, all correct processes eventually Adeliver m, unless m is discarded by line 18. However, the protocol sn can only be changed by issuing an ABcast with the same protocol sn. By the uniform total order property of sn, if p_i Adelivers m before a protocol change message, then every process Adelivers the protocol change message only after it has Adelivered m. So no stack discards m by line 18 in the context of the protocol sn, i.e., all correct processes eventually Adeliver m. $\Box$

Uniform integrity Since every atomic broadcast protocol satisfies integrity, we have only to prove that the replacement of atomic broadcast does not lead some message m to be Adelivered twice, i.e., by two different protocols sn and sn'. Let $sn < sn'$, and assume that m is Adelivered by protocol sn. Since m is Adelivered by the protocol sn, message m is not reissued at line 16. Moreover, since m is issued by the protocol sn, line 18 prevents m from being Adelivered by a protocol different from sn. $\Box$

Uniform total order Let message m be Adelivered before message m' by process p_i using stack i. The uniform total order property trivially holds if the two messages are Adelivered by the same protocol. So assume that m is delivered in stack i by protocol sn and m' by protocol sn', with $sn < sn'$. Since stack i has changed its ABcast protocol, it must have Adelivered a protocol change message $(newABcast, sn, prot)$ at line 10 (after m and before m'). Assume now that stack j Adelivers m'. Stack i Adelivers m' by protocol sn'; so, because of line 18, stack j can only Adeliver m' by protocol sn'. So stack j must have Adelivered the message $(newABcast, sn, prot)$ before Adelivering m' (otherwise m' would be delivered by the same protocol sn) (*). However, the protocol sn satisfies the uniform total order property, and has Adelivered m before $(newABcast, sn, prot)$. So stack j can only Adeliver $(newABcast, sn, prot)$ after it has Adelivered m (**). By (*) and (**), if stack j it has Adelivered m', it must have Adelivered m earlier. $\Box$

5.3 Comparison with Existing Solutions

Our solution has advantages over existing solutions. Firstly, the replacement protocol only requires ABcast. Contrary to other solutions, it does not require additional mechanisms such as barrier synchronization (Graceful Adaptation [6]) or group membership (Maestro [20] and Appia [15]). Note that even if the barrier synchronization is run in parallel with message flow in Graceful Adaptation, the use of barrier synchronization should be avoided because of its implementation complexity in an asynchronous network. The second main advantage of our solution is that the application on top of the stack is never blocked, which is not the case in the Maestro solution.

6 Performance

In this section, we present measurements showing the impact of updating the atomic broadcast protocol on the overall performance of our group communication stack (see Figure 4).

6.1 Instrumentation

Our implementation uses SAMOA [22] – a Java protocol framework that we have designed in our previous work. The framework can be used to implement network protocols as a collection of modules as described in Section 2.

We have made performance tests using a cluster of 7 PCs running Red Hat Linux 7.2 (kernel 2.4.18), 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 switch.

6.2 Benchmark

In our experiment, we have compared the *average latency* [19] 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 machine (stack) i. We define the *average latency* of m as the average of $t_i(m)$ for all machines (stacks) i.

In our experiments – involving 3 or 7 machines (stacks) – messages of 4Mb were ABcast under a constant load by all machines (stacks). In the middle of the experiment, any process triggers the replacement of ABcast and continues to issue ABcast messages. We consider that the replacement starts when any process triggers a replacement and finishes when all machines have replaced the old modules by new modules.

In our experiment, we replace the Chandra-Toueg ABcast [5] protocol by the same protocol, while performing all steps of the replacement algorithm (e.g., unbinding the old module, creating a new module, etc).

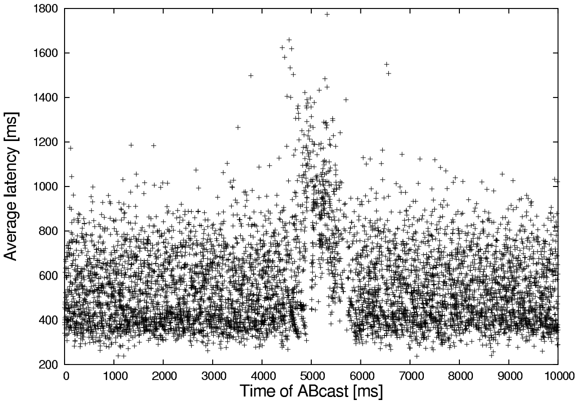

Figure 5. Latency as a function of the time at which ABcast is issued.

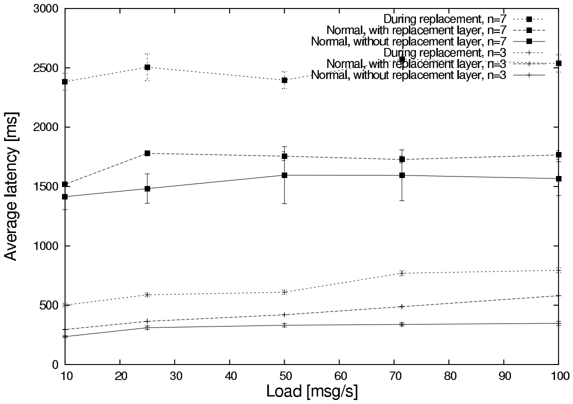

Figure 6. Latency as a function of the load.

This allows us to measure the exact impact of the replacement algorithm.

6.3 Measurement Results

The benchmark was executed for several values of the parameters. Figure 5 shows the results of an experiment where the impact of the replacement is clearly visible. The figure shows the average latency of atomic broadcast as a function of the time (in milliseconds) at which the atomic broadcast is issued. The experiment is with 3 machines, 75 messages ABcast per second, and the replacement algorithm invoked at time 5000. We show the result of several experiments with the same parameters, which is why several latency values are shown on the vertical axis for a given time t on the horizontal axis. We can observe that the average latency increases around $t = 5000$, but quickly stabilizes to reach the level it had before the replacement. Moreover, there is no interruption in the service availability. It should be added that the relatively large latency values are due to a non-optimized atomic broadcast algorithm (e.g., consensus is executed on messages and not on message identifiers).

Figure 6 shows the latency as a function of the load for various group sizes n ($n = 3$ or $n = 7$), where the *load* is the number of ABcast calls per second. The solid graphs represent the normal latency values, i.e., a group communication stack without a replacement layer. The dashed graphs represent the latency before the replacement in a group communication stack with a replacement layer. The dotted graphs represent the latency during the replacement (i.e., after the replacement request and before the new module replaces the old module in all stacks). Figure 6 shows that the cost of adding a replacement layer (approximately 15%) is not so important. It also shows the overhead of the replacement on the latencies. Note that this overhead lasts during a short period (approximately one second).

7 Conclusion

Updating middleware protocols on-the-fly is more difficult than purely local updates of software modules, since it requires global synchronization or coordination of local updates. We proposed a novel approach to this problem that is fully modular and highly flexible.

We have validated our approach by implementing a group communication middleware system using the SAMOA protocol framework. Our middleware enjoys a clear separation of concerns: updateable protocols can be implemented as usual, with the replacement algorithm implemented separately and executed in the background. We made several experiments in a LAN. The results of these experiments are very encouraging. The overhead of switching on-the-fly between different implementations of distributed agreement protocols is negligible.

We plan to work in the future on more generic replacement algorithms to allow replacement of a larger set of protocols. We have already designed an algorithm to replace consensus protocols [16], another essential building block of our group communication middleware.

Acknowledgments

We would like to thank Richard Ekwall for his comments on an earlier version of the paper. We also thank Sergio Mena for the implementation of several modules of our adaptive group communication middleware.

References

[1] M. Bickford, C. Kreitz, R. van Renesse, and R. L. Constable. An experiment in formal design using metaproperties. In *Proc. DISCEX-II '01: the 2nd DARPA Information Survivability Conference and Exposition*. IEEE, June 2001.

[2] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Proc. Middleware 2000*, volume 1795 of *LNCS*. Springer, Apr. 2000.

[3] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

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

[6] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *Proc. ICDCS '01: the 21st IEEE International Conference on Distributed Computing Systems*, Apr. 2001.

[7] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[8] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the IEICE: the Institute for Electronics, Information and Communication Engineers, IEICE/IEEE Joint Special Issue on Assurance Systems and Networks*, E86-B(10):2154–2166, 2003.

[9] Y.-F. Lee and R.-C. Chang. Developing dynamic-reconfigurable communication protocol stacks using Java. *Software Practice & Experience*, 35(6):601–620, 2005.

[10] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting metaproperties. In *Proc. Workshop on Applied Reliable Group Communication (WARGC '01)*, Apr. 2001.

[11] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[12] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[13] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. Middleware '03*, volume 2672 of *LNCS*. Springer, June 2003.

[14] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. ICDCS '01: the 21st IEEE International Conference on Distributed Computing Systems*, Apr. 2001.

[15] J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and A. Lopes. Context adaptation of the communication stack. In *Proc. the 3rd Workshop on Mobile Distributed Computing (MDC '05)*, June 2005.

[16] O. Rütti, P. T. Wojciechowski, and A. Schiper. Dynamic update of distributed agreement protocols. Technical Report IC-2005-012, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Mar. 2005.

[17] A. Schiper. Dynamic Group Communication. To appear in *ACM Distributed Computing*, 2006.

[18] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proc. ICDCS '03: the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003.

[19] P. Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Aug. 2003.

[20] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software Practice & Experience*, 28(9):963–979, 1998.

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

[22] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proc. IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.