Dynamic Update of Distributed Agreement Protocols*

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

March 29, 2005

Technical Report IC-2005-012

Abstract

In this paper, we address the problem of dynamic protocol update (DPU) that requires global coordination of local code replacements. We propose a novel approach to DPU. The key idea is the use of synchronization facilities of the services that get updated. This solution makes global update simple and efficient. We describe an experimental implementation of adaptable group communication middleware. It can switch between different distributed agreement protocols on-the-fly. All middleware services, 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.

Key-words: distributed algorithms, group communication, dynamic protocol update, reliable systems, modular composition and protocol frameworks

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

Contents

List of Figures

1 Introduction

Recent years have seen a growing interest in tools for building adaptable systems that can be reconfigured and adapted to a changing environment or user requirements (see, e.g. a survey paper [17] for examples of such tools and techniques). We are interested in *dynamically* adaptable middleware [28, 8]. It allows software modules or components that implement the middleware protocols to be replaced on-the-fly. The benefit is the decrease of software upgrade and maintenance costs in systems that must run non-stop.

In this paper, we focus on the problem of dynamic update that involves modification of the *protocol* implemented by the modules that get replaced. A few implementations of such type of adaptable middleware exist (e.g. [28, 8, 24]). However, they are not free from drawbacks, which we discuss below.

In the past, dynamic software update was confined mostly to domains such as telecommunication switches [1] and air traffic control. However, today more and more applications have similar requirements, including mobile embedded systems and the future generation of Internet-wired consumer electronics. Therefore it is important to investigate methods and algorithms that can efficiently and seamlessly replace software modules at runtime.

1.1 Dynamic Software Update (DSU)

There has been a large amount of work in the academic and commercial community addressing a *general* problem of *dynamic software update (DSU)*. This work has been usually conducted either in the context of dynamic module and class replacement in virtual machines [29, 16, 26], or in the context of programming language design for code replacement [1, 15, 5, 9, 12, 3].

However, relatively few current DSU implementations allow for updates on many machines in a coordinated manner, so that applications running on top of the system are not disrupted by the change of the underlying protocol. One reason for a slow development in this area, is perhaps because updating network protocols dynamically is considered to be difficult. Another reason is the belief that updating protocols on-the-fly may not be efficient nor scalable.

Indeed, most of the DSU approaches that we are aware of provide support only for updating code *locally*, i.e. on the same machine (a few exceptions will be discussed below). Moreover, the existing DSU implementations are often *unsafe*, e.g. local updates are not synchronized with method calls made by other threads, thus leading to system crash; or, at best, some form of synchronization is provided but only within the same machine. This support is however not sufficient to dynamically updating a distributed protocol running on a group of machines.

1.2 Dynamic Protocol Update (DPU)

In this paper we study the problem of *dynamic protocol update (DPU)* as a special case of DSU, which is focussed on the global synchronization aspects.

We address the following challenges:

- Updating protocols dynamically means that all local updates must be (eventually) consistently integrated on all or selected machines that are providing a service implemented by the protocol.
- It is required that the execution of applications that are using the protocol at the update-time will not be affected.
- For pragmatic reasons, it is also desirable that the whole system should not be blocked, and must remain available while protocols are updated.
- To avoid interference between concurrent versions of the protocol, some global synchronization may be however required. We would like to minimize the impact of this global synchronization, so that DPU efficiency and scalability is not degraded.

Although some implementations of DPU exist (see [28, 8, 24] among others), we think that the above challenges are still not completely solved, and more theoretical and practical work is needed. In particular, there is little common understanding of what properties of DPU should be considered and how they could be efficiently guaranteed at runtime. We made some initial step in [30], where we define a formal mathematical model of DPU, and use it to specify plausible update algorithms; the specifications make explicit different levels of synchrony between local updates (different updateable services may demand different levels).

In this paper we focus on the implementation of DPU, and introduce a novel approach to a fully synchronized protocol update, that improves over existing solutions. For instance, modular middleware systems, such as Ensemble [28] and Cactus [8], support some form of dynamic protocol update by allowing a set of (predefined) protocols to be switched between or reconfigured at run time. However, Ensemble allows only the whole protocol stack to be replaced at once. On the other hand, Cactus coordinates local updates of individual software components, but it uses a rather heavy global barrier synchronization for updating *all* types of modules (this synchronization is implemented as part of the update algorithm). Moreover, the structure of software components in the protocol stack cannot be easily changed (or decomposed) on-the-fly. Our goal is to design DPU support that is efficient and more flexible than in the above systems. Contrary to implementations that depend on a central update server (such as in [24]), we require our DPU algorithm to be fully distributed.

In this paper, we study the problem of DPU on the example of two *distributed agreement* protocols: distributed consensus and atomic broadcast. Distributed agreement protocols are good representatives of non-trivial distributed algorithms, where different problems of DPU can be seen rather clearly. The distributed consensus and atomic broadcast services are considered to be important building blocks for *group communication* middleware systems [18]. Such systems are used for replicating non-stop servers (to make them tolerant to crashes). It is therefore important that protocols can be updated dynamically and safely. The results developed in this paper can be valuable for developers of reliable, non-stop systems.

1.3 Contribution

We make several contributions in this paper:

- We identify two properties of dynamically updateable systems, i.e. *stack well-formedness* and *protocol operationability*. Preserving these properties during dynamic update guarantees that the update is transparent to the protocol that gets updated; this requirement forms our basic correctness condition.
- We describe a novel DPU algorithm, which can switch between different distributed agreement protocols (e.g. consensus and atomic broadcast). The key idea of our approach is to reuse services that get replaced to replace them, which makes the update algorithm simple, efficient and scalable.
- We prove that the update algorithm for agreement protocols satisfies the DPU correctness properties that we have defined, and some additional correctness properties that are specific to the services being replaced.

We have implemented the middleware and our update algorithm using SAMOA [31] – a protocol framework that has been developed in our previous work. We have experimented with switching on-the-fly between different implementations of the distributed consensus and atomic broadcast services. The former one uses: Chandra-Toueg's algorithm [7], Lamport's Paxos algorithm [14, 21], and Mostéfaoui-Raynal's algorithm [20]. We made several measurement tests to examine the impact of dynamic protocol replacement on system performance. We give some example results, showing that the delay in response time caused by switching between different protocols is negligible.

Our DPU algorithm is general, in the sense that it only depends on those properties of agreement protocols that are common to *all* current and future implementations of the distributed agreement problem. We believe also that the approach described in this paper applies to other protocol classes, too. For instance, we are currently designing an analogous DPU support for failure detector protocols and other building blocks of the group communication middleware.

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 dynamic protocol update. Section 4 presents our approach to DPU. Sections 6 and 7 describe the replacement algorithms for switching on-the-fly between different consensus and atomic broadcast protocols. Section 8 presents performance results. Section 9 contains related work, and Section 10 concludes.

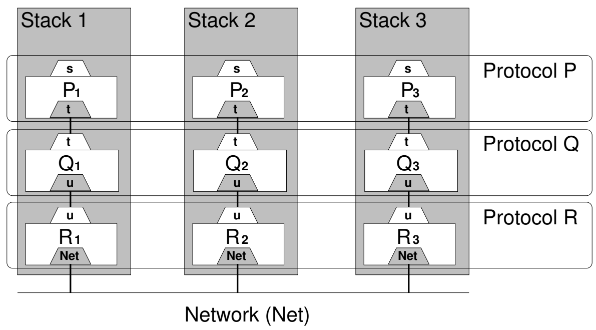

Figure 1: An example protocol architecture.

2 Model

In this section we introduce a simple model that we use in the following sections to describe the correctness properties and implementation of DPU.

Basic definitions In this paper we consider *services* that are provided by network protocols. A *protocol* describes the exchange of *messages* across an asynchronous network. Protocols are implemented by a set of identical *modules*, each module running on a different machine (or site). To provide a service a module may require other services. A module may maintain some data of the protocol, such as available TCP connections and logging information. A set of all modules that are located on a site is called a protocol *stack*.

In Figure 1, we show an example networked system. Protocols are represented with capital letters P, Q and R, and services with small letters s, t and u. 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 t and requires service u (see Fig. 1). Note that the network is also a service (named Net).

Module binding To call a service, a stack must contain a module that is *bound* to the service. Modules can be bound and unbound dynamically. Unbinding a module does not remove it from the stack. Stacks may contain several modules that can provide the same service. When we make a service call, however, only modules that are bound to the service can be chosen to provide the service. For simplicity, we assume that at most one module in a stack is bound to a service at a time. If no module is bound, a service call is blocked until some module is bound to the service. Note, however, that a module P_1 can complete a service call made using another module P_2 of the same protocol P, even if P_1 is unbound.

Module interactions Consider a *call* of a service t, which has been made by some module P_i. The service t is provided by module Q_i. We define *response*

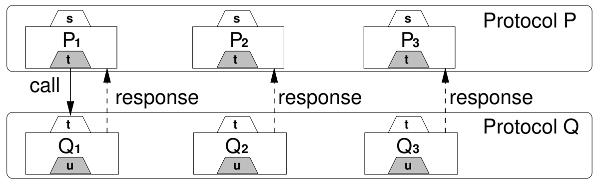

Figure 2: Service calls and responses.

to this call to be any invocation of a module P_j by Q_j in some stack j (where $j = i$ or $j \neq i$) that is related to the initial call. If P_j is not currently in stack j, then the invocation made by Q_j is completed when P_j will be added to stack j.

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

Figure 2 illustrates an example interaction. The call of a service t 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 t. Responses to the call of service t lead to a remote interaction of P_1 with P_2 and P_3.

Service call completeness We say that a service call is *complete* when all responses related to the call have been effectuated. The call completeness property is related to the semantics of the service. Thus, if a service is fault-tolerant, the service call is complete when all responses related to the call have been effectuated on correct processes.

For example, consider an *atomic broadcast* service (which is part of our example stack described in Section 5). The service can be used to broadcast a message, so that the message is delivered in all stacks in the same order.[1] The service can be called using the ABcast(m) primitive, where m is a message to be broadcast. The message can be delivered using the Adeliver(m) primitive. Execution of Adeliver(m) is a response to the service call. The call of the atomic broadcast service is complete when each stack has invoked the primitive Adeliver(m).

3 General Dynamic Update Properties

In this section, we consider several general correctness properties of dynamic replacement of distributed protocols. In the following sections, we describe our switching algorithm and show that it ensures these properties.

Firstly, we define a property that ensures correct local interactions. We define 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.

[1] To simplify presentation, we ignore crashes at the moment.

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 at least 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 at least one module.

Below we define a protocol-operationability property, which describes remote interactions. It ensures that whenever a service is called, then all plausible responses to this call (in non-crashed stacks) are guaranteed to occur. We can 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 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 stacks j in Π eventually contain a module P_j.

We assume that stacks may crash at any time. We therefore define the above properties with respect to a dynamic set of stacks that are non-crashed.

The strong protocol-operationability implies weak protocol-operationability. In the rest of the paper, we consider only the weak one, since it does not depend on the existence of global synchronization.

4 Our Approach to Dynamic Protocol Replacement

Below, we describe our approach to the replacement of network protocols at run time. In the following sections, we illustrate the approach using an example updateable middleware system that we have implemented.

Dynamic Protocol Replacement Consider two protocols P and P' that provide the same service s. We define replacement of P by P' in a set of stacks Π to be replacement of the module P_i by P'_i in every stack i in Π ($i = 1, 2, ..$) while maintaining the stack well-formedness, protocol operationability, and any other properties that are specific to the service s. Preserving these properties until the replacement completes ensures a *transparent* replacement, i.e. the users of service s are not able to notice any difference between before, while and after the replacement. Replacement of a protocol P by P' completes when modules P_i are unbound and P'_i are bound in all stacks i in Π.

The replacement module The main idea is to extend a protocol stack with a *replacement module* that implements a level of indirection between service calls and the service provider. 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 general properties described in Section 3, some additional

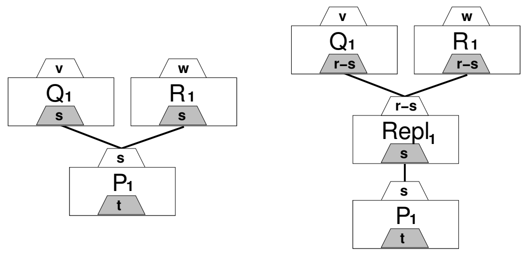

Figure 3: The module composition with and without a replacement module *Repl*.

properties must be satisfied; these properties are specific to the service being updated. The implementation of the replacement module is therefore tied to the specification of the updateable services.

Figure 3 shows an example stack without a replacement module (in the left) and with the replacement module $Repl_1$ (in the right), where 1 denotes a stack number. The $Repl_1$ module is used to replace a protocol P that provides service s in stack 1 (note that $Repl_1$ *requires* service s). Modules Q_1 and R_1 are two modules that may call service s. In the updateable system, the service s is not called directly, but via an interface r-s that is provided by $Repl_1$.

The advantage of our solution is that the implementation of the replacement module is orthogonal to the implementation of updateable protocols. Our approach is therefore modular. The switching algorithm for a given service is implemented entirely by the replacement module. Protocol modules are not even aware that the protocol replacement takes place. On the other hand, the protocol programmer using other approaches, e.g. [28, 8], must extend *each* new updateable protocol implementing the service, so that the protocol can explicitly interact with a switching manager.

Module removal Modules can be added to a stack, and also removed. Consider a call of a service t in stack 1 (see Fig. 2). Let us assume that a module Q_1 is chosen to execute the call. We cannot remove module Q_j ($j = 1, 2, 3$) from any stack j until the call of service t is complete (otherwise properties of service t may be violated). Removing a module of a protocol P from a stack without violating the properties of the corresponding service is possible only when all responses to any pending service calls of protocol P (made in this or another stack) have been effectuated. This requires some form of global synchronization.

One solution is to use *barrier synchronization.* We propose a slightly different approach: when a module Q_j is locally no more needed, the local replacement module sends a message (*removeModule*, Q_j) to all stacks. When a replacement module in a stack has received this message from all stacks in Π, it can remove the module Q_j locally. We assume the existence of a perfect failure detector [6] or group membership that can remove crashed stacks from Π.

For simplicity, the replacement algorithm described in Sections 6 and 7 does not remove modules; old modules simply remain in the stack.

5 DPU Example: Group Communication Middleware

To illustrate our approach to DPU, we have developed an example modular, *updateable group communication* middleware. It allows protocol modules that implement various group communication services to be replaced at run time; the services are continuously provided while the update takes place. In this section we define the architecture of our middleware.

The middleware protocols assume an asynchronous network, with distributed *processes* that may fail by crashing. In the context of this paper, we use the terms "process" and "stack" interchangeably. A *correct* process is a process that does not crash. Processes that have crashed can recover, but with a new identity.

Figure 4 shows the architecture of our middleware; it builds on the group communication stack in [18]. The meaning of modules in the top half of the figure is following:

- 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 [6].
- The *CT* module implements a *distributed consensus* service using the Chandra-Toueg algorithm [7] that is based on a rotating coordinator.
- The *ABcast* module implements an *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 implements a *group membership* service that maintains a consistent membership data among all group members; the module requires the atomic broadcast service (see [22] for the details).
- The *Repl* module implements the replacement protocol; we discuss this module in Sections 6 and 7.

6 Replacement of the Consensus Protocol

Let us consider a dynamic update of the distributed consensus service. Below we describe replacement of Chandra-Toueg's algorithm [7] with the Paxos consensus algorithm [14, 21]. We have also implemented Mostéfaoui-Raynal's consensus algorithm [20] and experimented with switching between these three protocols.

Fig. 4 illustrates two runtime configurations of our group communication stack; the first one uses the CT module while the second one uses the Paxos

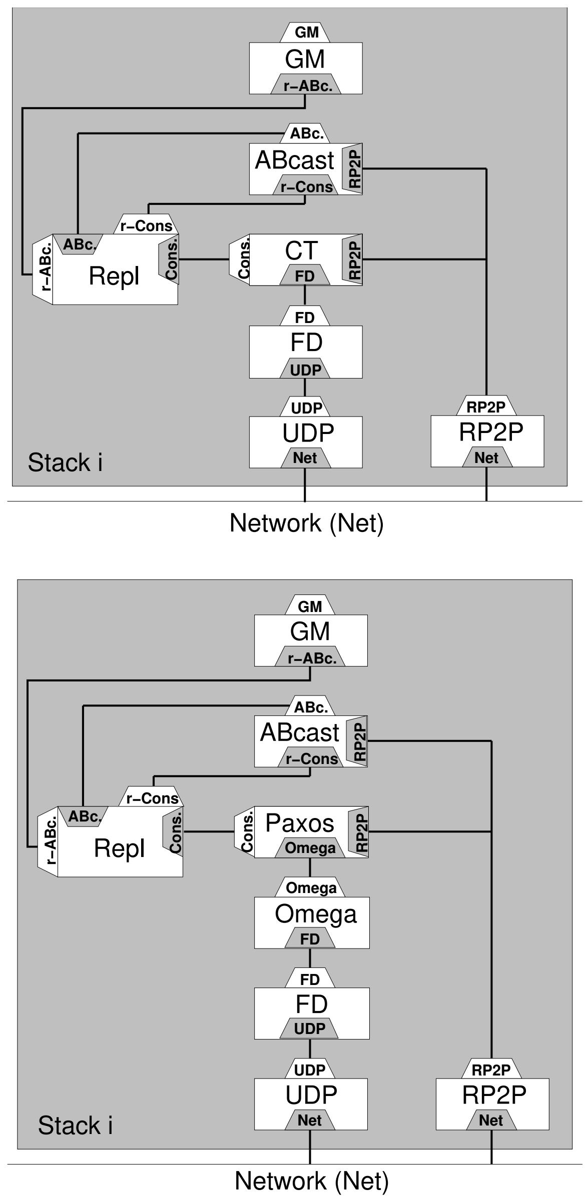

Figure 4: Two example runtime configurations of our group communication stack, using Chandra-Toueg algorithm (in the top half) and Paxos algorithm (in the bottom half of the figure).

module. Note that the latter stack (with Paxos) contains an additional module, named *Omega*, which implements the leader election service. This module, together with the failure detector FD, are designed to provide the guarantees of the Ω eventual leader oracle [6] that is required by the Paxos algorithm.

A runtime configuration of our middleware with Mostéfaoui-Raynal's algorithm is identical to the stack in Fig. 4 that uses Chandra-Toueg's algorithm (but with the CT module replaced accordingly).

6.1 Distributed Consensus

We can specify the distributed consensus problem as follows. Consider a set Π of distributed processes. Every process $p \in \Pi$ can *propose* a value v_p. We require that all non-crashed processes have to *decide* on the same value v. The decided value must be the value proposed by one of the processes in Π. More formally, consensus is defined by the following properties[2] [7]:

- *Termination:* Every correct process eventually decides some value.
- *Uniform agreement:* No two processes decide differently.
- *Uniform validity:* If a process decides v, then v was proposed by some process.

In our stack, the consensus service can be called many times during system lifetime. We identify different instances of consensus runs using a parameter k.

6.2 Replacement Algorithm

Below we describe an implementation of the *Repl* module; see Algorithm 1 for the pseudocode[3]. Then we show that the algorithm ensures weak stack well-formedness and weak protocol operationability. Note that the replacement algorithm uses the atomic broadcast service. Atomic broadcast is needed if the replacement of several protocols (e.g., consensus and atomic broadcast) can be initiated concurrently; atomic broadcast is not needed when only consensus service has to be updated. Note also that the replacement algorithm assumes that if the consensus algorithm is replaced several times (i.e., $prot_1$ by $prot_2$, then $prot_2$ by $prot_3$, etc.), then these replacements are not close to each other in time.(It is rather easy to lift this restriction, but this would make the replacement algorithm slightly more complex.)

The replacement is triggered by the call changeConsensus(*prot*), where *prot* is the new consensus protocol (see Algorithm 1, line 5). This call triggers a call to ABcast(*newConsensus*,*prot*) (line 6): *newConsensus* indicates the request to replace the consensus protocol. Upon Adelivery of this message (line 7),

[2] Actually, these properties define *uniform consensus* but the difference is not important in the context of this paper.

[3] Each *upon* block is executed in mutual exclusion. A *wait* statement (see e.g., line 16) releases mutual exclusion (similarly to a *wait* in a monitor).

the boolean variable *repRequested* is set to *true* (line 10). At that point the replacement algorithm does nothing: it just waits for the next call to consensus (line 11). Note that an instance of consensus can be underway, in which case the current protocol is still used. It is also possible that no instance of consensus is underway. The two cases do not need to be distinguished.

Whenever consensus is called (line 11[4]), the call is redirected to the module actually providing the consensus service, while modifying the *proposal* value to $(proposal, replacement)$: *replacement* is a boolean value indicating whether a replacement is requested or not (line 12).[5] When the consensus module returns a decision (line 13), the value of the *replacement* parameter is tested. If *true*, then the replacement of the consensus module takes place (lines 14-20). Line 16 is needed for the following reason: if atomic broadcast does not use consensus (which is not the case in our stack), then it is possible that some stack i terminates a consensus with $replacement = true$ while *newProtocol* is still *nil*. Once $newProtocol \neq nil$, the new consensus module is created by the call at line 17. The loop of line 25 recursively creates the modules needed by the module *newProtocol*. In the case of Paxos, the call to *create_module* creates the module Omega (see Fig. 4). Once the new consensus module is created, the decision value of the last consensus is returned (line 21).

We now show that the replacement protocol satisfies the required properties.

Weak protocol operationability We need to prove that, whenever the consensus module *new* is bound in a correct stack i, then the consensus module *new* is eventually added in all non-crashed stacks. This follows from (1) the property of atomic broadcast, and (2) the property of consensus.
(1) If the module *new* is bound in stack i, then stack i has Adelivered the message $(newConsensus, new)$ (line 7). Atomic broadcast ensures that the same message is eventually Adelivered by all other non-crashed stacks. So eventually the *newProtocol* variable equals to *new* on all non-crashed stacks.
(2) If the module *new* is bound in stack i, then on stack i some instance k of consensus has terminated with the decision $(-, true)$. By the termination property of consensus, all non-crashed stacks eventually terminate instance k of consensus. By the uniform agreement of consensus, the decision value is also $(-, true)$.
(1) ensures that the wait statement of line 16 eventually terminates; (2) ensures that at line 17, *new* is added in all stacks. This completes the proof. □

Weak stack well-formedness Similarly to the previous proof, we need to show that line 16 eventually terminates. Weak stack well-formedness is then trivially ensured by the *create_module* procedure (line 22).

[4] We follow the notation introduced in Figure 3: *r-propose* and *r-decide* denote the *propose* and *decide* interfaces of the replacement module.

[5] The consensus module solves consensus for a proposal of any type. So changing *proposal* to $(proposal, replacement)$ has no impact on the consensus module.

Algorithm 1 Replacement of consensus: code of stack i.

```
Initialisation:
  repRequested ← false                       {Is there a replacement requested?}
  curProtocol ← current consensus protocol
  newProtocol ← nil                          {new consensus protocol}

upon changeConsensus(prot) do
  ABcast(newConsensus, prot)

upon Adeliver(newConsensus, prot) do
  if curProtocol ≠ prot then
    newProtocol ← prot
    repRequested ← true

upon r-propose(k, proposal) do
  propose(k, (proposal, repRequested))

upon decide(k, (value, replacement)) do
  if replacement then
    unbind curProtocol from consensus service
    wait until newProtocol ≠ nil
    create_module(newProtocol)
    curProtocol ← newProtocol
    repRequested ← false
    newProtocol ← nil
  r-decide(k, value)

procedure create_module(p)
  create p
  bind p
  for all s ∈ services required by p do
    if if no module is bound to service s in stack i then
      find a module q providing service s
      create_module(q)
```

Protocol coherence Apart from these general properties, the replacement of consensus must satisfy a specific property that we call *protocol coherence*. This property requires that all stacks use the same consensus protocol for solving the same instance k of consensus. The need of this property is rather obvious. If stack i starts instance k of consensus using the CT module, it is clear that another stack j cannot start the same instance k of consensus using another consensus module, e.g., the Paxos module.

Protocol coherence holds under the assumption that consensus is executed sequentially (i.e., instance $k+1$ of consensus is started on stack i only after instance k has terminated on stack i). [6] However, this restriction can be easily removed. For presentation reasons, we choose to discuss the simplest version of our algorithm. Assume that stack i switches from the consensus protocol *cur* to the consensus protocol *new* between instance k and $k+1$. Thus the decision for consensus k on stack i is $(-, true)$. By the uniform agreement property of consensus, any stack j also decides $(-, true)$ for consensus k. So every stack uses the consensus module *new* for consensus $k+1$. □

[6]The algorithm implemented by the ABcast module used in our stack satisfies this property.

7 Replacement of the Atomic Broadcast Protocol

We describe now the replacement of the atomic broadcast (or ABcast) protocol in our group communication stack. Note that our ABcast protocol is not implemented on top of a view synchrony protocol as it is usually the case. However, our replacement algorithm is general and does not rely on this specificity.

7.1 Atomic Broadcast

Atomic broadcast, defined by the two primitives ABcast and Adeliver, satisfies the following properties [10]:

- *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.

7.2 Replacement Algorithm

We discuss now the replacement of the atomic broadcast protocol (see Algorithm 2).[7] The replacement is triggered by the call changeABcast(*prot*), where *prot* is the new atomic broadcast protocol (Algorithm 2, line 5). This call triggers a call to ABcast($newABcast$, $seqNumber$, *prot*) (line 6). The parameter $newABcast$ indicates the request to change the atomic broadcast protocol; the variable $seqNumber$ identifies the current atomic broadcast protocol (the variable is incremented each time the atomic broadcast protocol is changed). The call at line 6 has to be compared to lines 7-9, which corresponds to a standard call to ABcast(m): the message m that is added to the set *undelivered*, and then the call ABcast(*nil*, $seqNumber$, m) is issued, where *nil* indicates an ordinary call to ABcast. The Adelivery is handled by the lines 10-16 (for messages with the tag $newABcast$) and by the lines 17-21 (for messages with the tag *nil*).

If a replacement is requested (lines 10-16), then the $seqNumber$ variable is incremented (line 11), the new module is created (line 13), and all undelivered messages are reissued using the new protocol (lines 15-16).

[7]Algorithms 1 and 2 are part of the same replacement module, even though they are presented separately.

Algorithm 2 Replacement of ABcast: code of stack i.

```
1: Initialisation:
2:    undelivered ← ∅                                {set of messages not yet Adelivered}
3:    curABcast ← 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 ← undelivered ∪ m
9:    ABcast(nil, seqNumber, m)

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

17: upon Adeliver(nil, sn, m) do
18:    if (sn = seqNumber) then
19:       if (m ∈ undelivered) then
20:          undelivered ← undelivered \ m
21:       rAdeliver(m);

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

If no replacement is requested (lines 17-21), then a test is performed to avoid that a message is Adelivered twice (line 18): messages with a sequence number corresponding to an older ABcast protocol are discarded.

It is easy to see that the replacement protocol satisfies strong stack well-formedness and weak protocol operationability. Strong stack well-formedness is trivially ensured, since the switch of ABcast modules is done atomically (lines 12-13). Weak protocol operationability can be shown with the same arguments as in Section 6. 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. 7.1) are satisfied across the replacement protocol (assuming that each ABcast protocol satisfies the properties of Section 7.1).

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. $\square$

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. $\square$

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. $\square$

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 $(sn, true, prot)$. 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. $\square$

8 Performance

In this section, we present the results of measurements which show the impact of the Consensus or ABcast protocol update on the overall performance of the group communication stack that we have implemented. Our measurement tests use the same benchmark for updating each of the two agreement protocols. The benchmark simply broadcasts messages using the ABcast protocol. Since our stack implements an atomic broadcast by reduction to consensus, we can use this benchmark both to evaluate the cost of the replacement algorithm for Consensus and the replacement algorithm for ABcast.

8.1 Instrumentation

Our implementation of the updateable middleware uses SAMOA [31] – a library package in Java that we have developed in our previous work. The package can be used to implement network protocols as a collection of modules. Modules can call methods of other modules in a local stack directly, or indirectly via local events. Events can be communicated in messages to other stacks, and thus trigger module methods remotely. Moreover, events can be bound and unbound to methods dynamically, which enables to modify the structure of a protocol stack on-the-fly.

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 hub.

8.2 Benchmark

Our main performance metric for atomic broadcast is *average latency* [27], 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 message m and the moment of delivering m in a stack i. We define the *average latency* of the m delivery as the average of the values $t_i(m)$ for all stacks i. Similarly, we define *late latency* as the maximum value among the values $t_i(m)$ for all stacks i. In our experiments, we consider the system to be stable when the late latency stabilizes.

We have used a simple benchmark test: all processes broadcast messages under a constant load using the ABcast protocol, where the load is defined as the number of ABcast messages per second. Once a stable state has been reached, the number of ABcast messages is constant. Then, some process (it can be any process) triggers the replacement of either ABcast or Consensus (depending on an experiment) and continues to broadcast the ABcast messages.

To measure the impact of the replacement algorithm on the protocol switch, a given agreement protocol is replaced by the same protocol; however, all replacement steps that are required for switching between arbitrary protocols are performed (unbinding an old module, creating a new module, etc.).

8.3 Measurement Results

Figures 5 and 6 show aggregated results of several experiments, respectively for the replacement of ABcast and Consensus. We present the average latency of an atomic broadcast as a function of the time at which the broadcast is issued (in milliseconds). We have used a group of three processes ($n = 3$) that have been broadcasting 75 messages per second, where each message has the 4Mb size. The replacement algorithm is invoked at time 5000. The results of several experiments (with the same values of parameters) have been superimposed in the figures; that is why we can see many values on the vertical axis for a given time value on the horizontal axis.

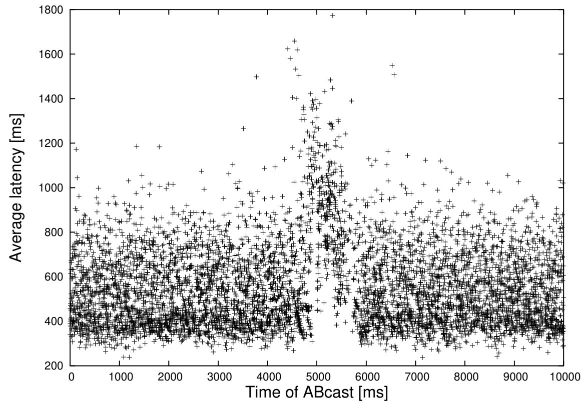

Figure 5: Average latency for the ABcast messages (with the replacement of ABcast starting at 5000).

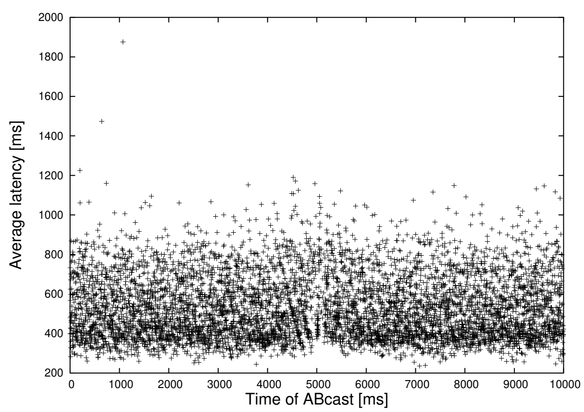

Figure 6: Average latency for the ABcast messages (with the replacement of Consensus starting at 5000).

We have also run our experiments with other values of the above parameters. The analysis of results however, which we present below, does not depend on the particular values of these parameters. In each case, the influence of dynamic replacement was similar.

During the replacement of ABcast, we can observe that the average latency increases around $t = 5000$, but quickly stabilizes to reach the level it had before the replacement. On the other hand, during the replacement of Consensus, the increase of average latency around $t = 5000$ is negligible. This difference can be easily explained. Delivery of messages from the *undelivered* set in the case of the ABcast replacement involves resending of these messages (see lines 15-16 in Algorithm 2). Message resending requires some additional network and CPU resources during the replacement, which explains that the latency for ABcast is larger. Note that in both experiments, there is no interruption of the service availability (see Figures 5 and 6), i.e. the users of the protocol that is simultaneously updated are never blocked.

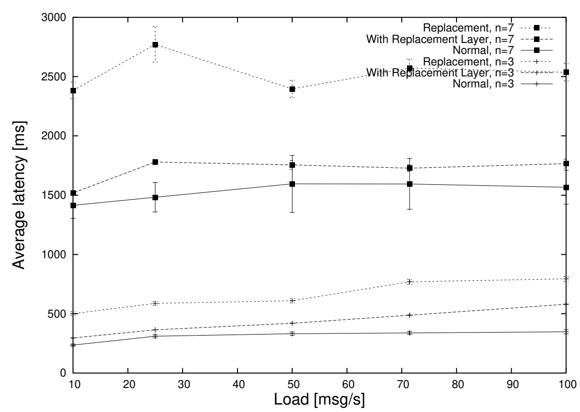

Figure 7: Average latency as a function of the message load.

Figure 7 shows the latency as a function of the message load for two group sizes ($n = 3$ and $n = 7$). The solid lines represent the normal latency values, i.e. for a group communication stack without a replacement layer. The dashed lines represent the latency in a stack with a replacement layer, however before any protocol replacement has commenced. These two graphs are therefore independent from the protocol that is going to be updated. The dotted lines represent the latency *during* the replacement of ABcast (i.e., after the replacement request has been issued and before old modules are replaced by new modules in all stacks). (We do not show the latency during the replacement of Consensus, since it is almost the same as the latency before the replacement, as previously shown in Figure 6.)

Figure 7 shows that the cost of adding a replacement layer is small and may be neglected. It also shows the influence of replacing the ABcast protocol on the latency. This influence has to be related with the short duration (approximatively one second) of the replacement.

It should be noted that the relatively large latency values are not caused by the SAMOA protocol framework, but they are due to the fact that the atomic broadcast algorithm (which is used by our benchmark) is not optimized. Our preliminary performance tests show that SAMOA is at least as efficient as the Cactus [32] and Appia [19] protocol frameworks.

9 Related Work

There are quite a number of implementations that support dynamic updating of software components. Below we describe related work in three closely related domains.

Component-based adaptable systems Component-based systems such as CORBA, COM, and JavaBeans provide limited support for updating modules (extending the state of objects), with clients being able to use run-time checks to determine which versions of modules are available; some extensions have been proposed, e.g. based on reflective ORBs [13]. The drawback is that old versions of the code may need to be maintained indefinitely. Java interfaces provide similar functionality to COM interfaces.

Many research groups have either extended the above technologies or have built their own component-based systems which support some features of dynamic (re)composition. They can allow the composition of components to be modified dynamically at compile time, link time, or runtime (although with none or few safety guarantees). They use different recomposition techniques, such as proxies, metaobject protocols, and aspect weaving. More details and references can be found in a survey paper on *compositional adaptation* [17].

Examples are the Conic [15] and Argus [5] programming environments; they have been used to build dynamically reconfigurable distributed systems. These approaches are however coupled to particular research languages that generally lack support for mainstream software development. Our experimental DPU implementation uses Java for pragmatic reasons but virtually any other language with dynamic class loading could be chosen.

Blair *et al.* [4] have investigated tools and techniques that can be used to implement dynamically adaptable systems in the context of Open-ORBs.

Local software update The programming language Erlang [1] allows software modules to be replaced at runtime, however with no safety guarantees.

A Java HotSpot Virtual Machine [26] allows a class instance to be replaced with the new instance in a running application through the debugger APIs.

There have been also work on languages for *safe* dynamic software updating by construction. This desirable property ensures that the system after dynamic update is type-correct, thus eliminating runtime errors due to type mismatch between old and new components. Let us briefly characterize some example work in this area.

Dynamic ML [29] enables type-safe module replacement at runtime; changes can include the alternation of abstract types at update-time, and the addition of module definitions via garbage-collection.

Dynamic Java classes [16] offer type safety preservation but compromise portability by modifying the Java Virtual Machine; also, class replacement is not synchronized with threads using old code.

Duggan [9] describes a type-safe approach that allows a new module to change the types exported by the original module; it however does not discuss the rebinding facility.

Hicks [12] describes a calculus of type-safe dynamic updating of native code.

Stoyle *et al.* [25] investigate type-safe dynamic updating in C-like languages. However, this work does not address the issues of global coordination of local updates.

Coordinated distributed update Few systems offer support for coordinating local updates. Below are some examples.

Van Renesse *et al.* [28] describe a "protocol switch" protocol, which synchronizes dynamic replacement of whole stacks in the Ensemble protocol framework, while we can replace individual modules.

Chen *et al.* [8] describe switching between network components within the Cactus protocol framework. A replacement manager on each host interacts explicitly with replaceable network components; it uses barrier synchronization for coordinating the beginning of the replacement across different hosts. We think that our separation of the replacement protocol from the protocols that get replaced is more elegant.

A similar solution to Cactus has been proposed in [24], but it uses a centralized manager, which limits its scope of applicability.

Properties of dynamic software updating To date relatively little work has been carried out on formalization of dynamic protocol update (DPU). In particular, none of the papers cited above formalizes conditions that are needed to guarantee the correctness of updating *distributed* protocols on-the-fly.

In our previous work [30], we have defined a formal mathematical model of DPU, and used it to specify the design space of update algorithms, focussing on the levels of synchrony between local updates (different updateable services may demand different levels).

There have been some attempts to specify the correct (static) composition of group communication building blocks, e.g. in the context of Ensemble [11]; see also [23]. However, we are not aware of much work that defines the properties of dynamically adaptable group communication systems.

The previous work closest to our own is by Bickford *at al.* [2] on designing a generic switching protocol for Ensemble using the Nupr logical programming environment. They have formally defined several communication (not structural, though) meta properties on traces of send and deliver events, that should be preserved by updateable protocols. They also describe briefly an example switching protocol. However, to the best of our knowledge the protocol has not been integrated with the Ensemble system. The main difference between Bickford's and our work is that our approach can capture properties that are specific to a class of middleware protocols (i.e. distributed agreement protocols), which allows us to optimize replacement algorithms.

10 Conclusion

Updating middleware protocols on-the-fly is more difficult than a local dynamic update of software modules since it requires a global synchronization or coordination of local updates. We proposed a novel approach to this problem, which assumes the use of synchronization facilities of the services that get updated.

We have validated our approach by implementing an example group communication middleware 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 are currently extending our replacement protocol to replace all modules of our group communication middleware. To support more applications, our updateable group communication stack has been extended to the crash/recovery model, instead of the crash/no-recovery model that we consider in this paper.

References

[1] J. L. Armstrong and S. R. Virding. Erlang – An experimental telephony switching language. In *Proc. XIII International Switching Symposium*, May 27–June 1, 1990.

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

[3] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *USE '03: Workshop on Unanticipated Software Evolution*, Apr. 2003.

[4] 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 *Middleware 2000*, pages 164–184. Springer, 2000.

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

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

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

[8] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *Proc. ICDCS '01*, Apr. 2001.

[9] D. Duggan. Type-based hot swapping of running modules. In *Proc. ICFP '01: The 6th ACM SIGPLAN Int'l Conference on Functional Programming*, Sept. 2001.

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

[11] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *Proc. TACAS '99*, LNCS 1579, Mar. 1999.

[12] M. Hicks. *Dynamic Software Updating*. PhD thesis, Computer and Information Science Department, University of Pennsylvania, Aug. 2001.

[13] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proc. Middleware 2000*, LNCS 1795, April 2000.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

[16] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP 2000*, LNCS 1850, June 2000.

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

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

[19] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. ICDCS 2001*, Apr. 2001.

[20] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proc. DISC '99: The 13th International Symposium on Distributed Computing*, LNCS 1693, Sept. 1999.

[21] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243(1–2):35–91, 2000.

[22] A. Schiper. Dynamic Group Communication. TR IC/2003/27, EPFL, April 2003.

[23] P. Sinha and N. Suri. Modular composition of redundancy management protocols in distributed systems: An outlook on simplifying protocol level formal specification and verification. In *Proc. ICDCS 2001*, Apr. 2001.

[24] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proc. ICDCS '03*, May 2003.

[25] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *POPL '05*, Jan. 2005.

[26] Sun Microsys., Inc. *Java HotSpot*, 2005. http://java.sun.com/products/hotspot/.

[27] 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.

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

[29] C. Walton, D. Kirli, and S. Gilmore. An abstract machine model of dynamic module replacement. *Future Generation Computer Systems*, 16:793–808, May 2000.

[30] 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*, LNCS. Springer, June 2005. To appear.

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

[32] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *Proc. INFOCOM 2001*, Apr. 2001.