On Correctness of Dynamic Protocol Update[*]

Paweł T. Wojciechowski and Olivier Rütti

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{Pawel.Wojciechowski,Olivier.Rutti}@epfl.ch

Abstract. Replacing or adding *network protocols* at runtime is problematic – it must involve synchronization of the protocol switch with ongoing local and network communication. We define a formal mathematical model of *dynamic protocol update (DPU)* and use it to define two DPU algorithms. The algorithms are based on fully-synchronized and lazy strategies. The two strategies implement updates with respectively, *strong* and *weak* safety properties. Our model allowed us to express the properties and the DPU algorithms clearly and abstractly, aiding algorithm design and correctness proofs.

1 Introduction

There is an important class of distributed applications that must run "non-stop". This is especially true of time-critical services, such as financial transactions, telephone switch systems, flight reservations and air traffic control systems. The service providers must be able to update their software, e.g. to fix program bugs, improve performance, and expand functionality. Unfortunately, stopping the system results in loss of service and revenue; it may also compromise safety. Moreover, systems that modify their behavior based on changes in the environment, require the ability to update their functionality *dynamically*, with minimal service interruption. There are quite a number of relevant implementations and techniques (e.g. [4, 1, 6, 9]). Software components can be rebound on-the-fly, using a mechanism of dynamic class loading and linking [10, 15].

In this paper, we focus on global update of *network components* that assumes modification of a network protocol implemented by the components. Such type of update introduces a new problem, however. Replacement of network components involves delicate synchronization (or coordination) of local updates which, if not handled appropriately, could easily prove so disruptive as to, at best, shut the system down, and, at worst, introduce malicious behaviour. Synchronizing local updates so that all software components in the distributed system end up updated in a consistent manner, and doing this while the system continuously provides service, represent serious challenges. It is therefore important to understand what are the minimal properties which must be satisfied by dynamic protocol update (DPU), and what is the range of possible DPU strategies?

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

Most of software update implementations concentrate on bug-fixes or software upgrades that do not alter the communication protocol. Thus, coordinating the protocol switch with ongoing communication can be done locally. Few implementations provide solutions to the problem described in this paper. Examples are implementations of dynamic protocol adaptation using Ensemble [21] and Cactus [6] protocol frameworks. They implement complex *DPU algorithms* for global synchronization of local updates. However, they lack both simplicity and generality and it is not clear what properties are actually guaranteed. Therefore, we believe a formal, mathematical model of DPU should be developed, in order to understand by both users and implementors of DPU technology what design choices can be considered, and what impact they have on DPU complexity and scalability. Unfortunately, little formalization work has been carried out to date (e.g. [8, 3, 19]; we discuss this work in §8). However, we are not aware of past efforts that have formalized the algorithms for global coordination of local updates or the DPU correctness properties.

A critical safety property of many network services is *message order* preservation. Consider *group communication* middleware [14] that can be used for replicating servers in order to make them tolerant to server crashes. Each replica in the system is guaranteed to receive all messages in the same order. Any update of middleware protocols must not affect this semantics. In this paper, we explain what it really means in case of updating *any* kind of protocols. For this, we construct a model of DPU and use it to define two synchronization-extreme DPU algorithms: a fully-synchronized algorithm that satisfies the message order property but seems impractical for Internet-wide update, and a synchronization-free, lazy algorithm that is scalable but does not guarantee this property.

What are the significant results of our work? Firstly, the lazy DPU strategy does not require any distributed infrastructure, which means that update with weak semantics is no more difficult than a local update. Secondly, the fully-synchronized and lazy strategies define the design space for more practical DPU algorithms that use only as much synchrony as required in a given case. We have actually designed and implemented such algorithms for updating protocols of the group communication middleware. In the end of this paper, we summarize results of this experiment; a complete report is also available [16]. It was somewhat surprising to us that the design space can be indeed usefully explored, leading to specialized DPU algorithms that are more efficient than the fully-synchronized strategy but preserving properties required by correct global update.

Our model abstracts away from any concrete implementation of modular protocols and of the DPU technology. We only make sure that our network communication is directly implementable above standard networks, such as Internet. For this, we assume that protocols use asynchronous, unordered, point-to-point messages; this is a realistic assumption about wide-area networks and common middleware services, where communication delays are not predictable. Our model abstracts away however from any unnecessary details of this communication. For instance, message addressing and message routing are the details of protocols themselves that we do not model here.

Symbols

Service names	x, y	$\in Mvar$
Required services	R	$\in 2^{Mvar}$
Messages	m	
Protocol modules	a, b, c	
Module types	t	$::= (x, R) \quad x \notin R$
Module bindings	w	$::= \updownarrow \mid \uparrow$

Terms

Protocol stacks	$P = \{a\ w, ..., b\ w'\}$
Module a in stack P	$P.a$
Peer modules of a	$\delta(a) = \{P.a'\ w \text{ for any } P \mid a' : (x, R)\}$ where $a : (x, R)$
Distributed protocols	$\mathcal{D} = \{P_1, ..., P_n\}$ or $\mathcal{D} = \{\delta(a), ..., \delta(b)\}$
Messages sent *to* $a, .., b$	$S_S = (m^{P.a}, ..., m'^{P'.b})$
Messages delivered *by* $a, .., b$	$S_D = (m^{P.a}, ..., m'^{P'.b})$
Message history	$S = (S_S,\ S_D)$
Sub-histories ($i = S, D$)	$S_i \vert a = \{m^{P.a'} \text{ for any } P \mid m^{P.a'} \in set(S_i),\ a' : (x, R)\}$ where $a : (x, R)$
	$S_i \vert P.a = \{m^{P.a} \mid m^{P.a} \in set(S_i)\}$
Protocol states	$\mathcal{D}$, S
Call service x in P	$P.x(m)$
Deliver m using a in P	$[m]P.a$

Fig. 1. The DPU model: Symbols and terms.

The paper is organized as follows. §2 and §3 define our model. §4 specifies DPU, and §5 defines its correctness properties. §6 defines two DPU algorithms. §7 sketches our implementation work, §8 contains related work, and §9 concludes.

2 Model

In this section, we define basic notions of protocol modules and protocol stacks in our DPU model. All symbols and terms are in Fig. 1.

Protocol stacks *Protocol services* (or *services* in short), denoted by metavariables x, y, are programming abstractions implemented by network protocols. Example services are "send a message reliably", "broadcast a message with FIFO guarantee", etc. Services can be *called* as functions, as in $x(m)$, where m is a *message*. No protocol data persist after the call returns.

We assume that protocols are *modular*, i.e. they can be composed from communicating *protocol modules* (or *modules* in short), denoted a, b. Modules are *typed* using pairs (x, R), where x is the name of a single service provided by the module, and R is a set of names of services that are required by the module in order to handle a call of x ($x \notin R$). We say that x is a module's *interface*.

Protocol stacks (or *stacks*), denoted P, are sets of modules accompanied by module bindings w, as in $P = \{a\ w, ..., b\ w'\}$; two kinds of bindings ($\updownarrow$ and $\uparrow$) will be explained in §3. We write $P.a\ w$ to denote a module a in stack P with binding w. We often omit bindings or stack names if we mean any binding or any stack, or the stack is known from the context. We abstract away in the model from physical machines – intuitively, stacks are located on machines that are interconnected via network.

Distributed protocols *Distributed protocols*, denoted $\mathcal{D}$, can be defined vertically or horizontally, i.e. either as sets of stacks $\{P_1, ..., P_n\}$, or as sets of logical protocols $\{\delta(a), ..., \delta(b)\}$, where a *logical protocol* $\delta(a)$ is a set of *peers* of module a, i.e. all modules in the system that have the same type as a. Each logical protocol defines a level of abstraction in a stack. Unless a distributed protocol is being updated, any two stacks are exactly the same.

For clarity, we assume in this paper a system model with no failures, where messages are not lost nor duplicated, and stacks are basically reliable. In our implementation of DPU, however, stacks may *crash* while a protocol is being updated, with a guarantee that all non-crashed stacks get updated.

3 Operational Semantics

Actions Interaction between protocol modules (and stacks) is by means of *asynchronous messages*. We use two kinds of actions to express the communication: a service call and a message delivery.

Service call $P.x(m)$ requests a service x of a stack P to deliver a fresh message m in one or many stacks, depending on if the communication is point-to-point or multicast. The call therefore appends to a global list of *sent messages* S_S, a list $(m^{P'.a'}, ..., m^{P''.a''})$ of duplicated messages m decorated with all modules $a', ..., a''$ of type (x, R) for some R, that should be used to deliver m in stacks $P', ..., P''$. Note that our stacks are *symmetric*: modules that are used to output and to deliver a message have the same type.

Message delivery $[m]P.a$ denotes delivery of a message m in a stack P using a module a. The intended semantics is that m is delivered by a to some other module in the local stack that can use m. We assume that the name of this module has been encoded in the message itself. We do not model it explicitly, however, as we only need to know who delivers a message. The delivery adds (using a Lisp-like constructor "::") an element $m^{P.a}$ to a global list of *delivered messages* S_D; the list has the same structure as S_S.

Protocol states are denoted as $\mathcal{D}$, S, where $\mathcal{D}$ is a distributed protocol and $S = (S_S,\ S_D)$ is a pair of the (initially empty) lists of the sent and delivered messages. We define the *execution* (or *evaluation*) of a protocol $\mathcal{D}$ as a state transition relation $\longrightarrow$, which transforms a state $\mathcal{D}$, S to $(\mathcal{D}, S)'$ as a result of a single action e, denoted $\xrightarrow{e}$; we sometimes omit the label e. The notation $(\mathcal{D}, S)'$ means $\mathcal{D}'$, S or $\mathcal{D}$, S' or $\mathcal{D}'$, S', depending on the context. We also use $\Longrightarrow$ to denote a possibly empty sequence of small step transitions.

Communication and freedom

$$\frac{\begin{array}{c} a : (x, R) \\ m^{P.a} \notin set(S_S) \quad m^{P.a} \notin set(S_D) \\ m^{P.a} \in set(S'_S) \\ S''' = (S''_S,\ m^{P.a} :: S''_D) \\ a\ w \in P \quad w \in \{\updownarrow, \uparrow\} \quad P \in \mathcal{D}' \end{array}}{\mathcal{D},\ (S_S, S_D) \xrightarrow{x(m)} \mathcal{D},\ (S'_S, S_D) \Longrightarrow \mathcal{D}',\ (S''_S, S''_D) \xrightarrow{[m]P.a} \mathcal{D}',\ S'''} \text{(Comm)}$$

$$\frac{S = (S_S,\ S_D) \quad S_S|a = S_D|a}{(a, (\mathcal{D},\ S))\ \textsc{Free}} \qquad \text{(Freedom)}$$

Module bindings

$$\frac{a\ w \in P \quad w \neq w'}{a\ w' \notin P} \text{(Sanity-1)} \qquad \frac{\begin{array}{c} a : (x, R) \quad b : (x, R') \quad a \neq b \\ a \updownarrow \in P \quad b\ w \in P \end{array}}{w = \uparrow} \text{(Sanity-2)}$$

Fig. 2. The DPU model: Operational semantics.

Communication We write $set(S_i)$ to denote a set of all elements in list S_i. The rule (Comm) in Fig. 2 says that each delivery action $[m]P.a$ in a stack P must be preceded in the execution trace by a corresponding call $x(m)$, where module a provides x (remember that our stacks are symmetric). There can be an arbitrary number of evaluation steps in-between since different protocol (or update) actions can be interleaved. The sets of sent, S_S, and delivered messages, S_D, are modified accordingly.

The details of message routing within a stack and between stacks are omitted here, as they are not useful in this paper. Example approaches can be found in [23], where we describe the semantics of module interaction and binding in Cactus and Appia – two example protocol frameworks that can be used to encode modular protocols in Java.

In network protocols, we can usually identify different levels of abstraction at which communication takes place. Consider delivery $[m]P.a$ of message m in stack P by module a, as the result of a service call $P'.x(m)$ in some other stack P'. The call may trigger several calls of services that x depends on (all that services are known from module type). In protocol frameworks, each of these calls gets as its argument a message m', that contains m and any additional data that are required in order to complete the call and to deliver the message using corresponding modules in stack P'.

Freedom In this paper, we consider network protocols that execute in terminating rounds, where a *round* is a sequence of reduction steps that has commenced with a service call $P.x(m)$ for some stack P, where message m must be fresh. The round terminates with delivery of message m for the last time. Or, more precisely, a round spawned with a fresh message m *terminates* (or *completes*) in a state $\mathcal{D}$, (S_S, S_D) if $S_S|m = S_D|m$, where $S_i|m$ $(i = S, D)$ is a list that is constructed from S_i by removing from it all messages other than m.

There can be many rounds executed concurrently. A distributed protocol $\mathcal{D}$ does not *get stuck* if all its rounds eventually complete.

A module a of a protocol $\mathcal{D}$ is *free* in a state $\mathcal{D}$, S, denoted $(a, (\mathcal{D}, S))$ Free, if there is currently no active round of the protocol that would deliver a message using either module a or any module (in any stack) that has the same type as a. We can define this property formally using sub-histories $S_i|a$ of messages that were sent $(i = S)$ and delivered $(i = D)$ by all modules of type of module a; see rule (Freedom) in Fig. 2 and the definition of $S_i|a$ in Fig. 1.

Bindings We assume that modules can be added and removed from stacks at runtime. This means that modules can be *dynamically bound* and *rebound*. Consider a module a of type (x, R) for some x and R. We write $a \updownarrow$ to denote the module a which has been *bound*, i.e. calls of service x to deliver a fresh message can use a, and messages can be also delivered by a.

We write $a \uparrow$ to denote a module a which is *passive*, i.e. calls of a service x provided by a are not allowed, unless there is another module b in the same stack that provides service x, and b is bound. A passive module can however deliver messages. Therefore, any round of a protocol $\delta(a)$ can complete using passive modules in $\delta(a)$, assuming that any services that are required by $\delta(a)$ to complete the round in a given stack, have bound modules in the stack.

Each module in a stack is either bound or passive. We also assume that for a given service in each stack, there can be at most one bound module at a time providing the service; these sanity conditions are defined formally in the bottom of Fig. 2.

4 Dynamic Protocol Update

We define *dynamic protocol update (DPU)* as a dynamic change of a distributed protocol, i.e. replacement or addition of its modules. We require that the change must eventually occur in all protocol stacks within, say, a cluster of servers or a large LAN. Below we use our model to formalize this definition.

Replaceability We can replace a module a by a module b in a protocol stack P only if b has the interface of a (or at least of a, if we had a notion of interface subtyping). This is motivated by requirements of real systems. If module b does not provide the service of a, then it means that the distributed protocol updated with b may get stuck since not all service calls can be effectuated, thus violating the (desirable) termination property of protocol rounds. We must also require that stack P provides all services that are required by b. These two requirements are expressed in Fig. 3 as a *replaceability property* $P\{b/a\}$, read "a replaceable by b in P". The property can be verified statically by checking module types.

Global update A *global update* $GU(\mathcal{D}, a, b)$ in Fig. 3, updates all stacks of a distributed protocol $\mathcal{D}$ with a module b, yielding an updated protocol $\mathcal{D}'$. To update a stack locally, GU calls a local update function LU (explained below).

$$\frac{\begin{array}{c} a:(x,R) \quad a \in P \\ b:(x,R') \quad R' = \{y \mid \exists c:(y,..) \;\; c \in P\} \end{array}}{P\{b/a\}} \qquad \text{(Replaceable)}$$

$$\frac{P\{b/a\} \quad w \in \{\updownarrow, \uparrow\}}{LU:(P,a,b) \to (P \setminus \{a\,w\}) \cup \{a\uparrow\} \cup \{b\updownarrow\}} \qquad \text{(Local-Update)}$$

$$\frac{\mathcal{D} = \{P_1, ..., P_n\} \quad \mathcal{D}' = \{LU(P_1,a,b), ..., LU(P_n,a,b)\}}{GU:(\mathcal{D},a,b) \to \mathcal{D}'} \qquad \text{(Global-Update)}$$

Fig. 3. The DPU specifications.

For practical reasons, global update should be concurrent with the execution of system services whenever possible. Blocking the whole system during update is unrealistic for large systems, and also not acceptable for non-stop systems. Thus, the transition $\mathcal{D}$, $S \overset{GU(\mathcal{D},a,b)}{\Longrightarrow} \mathcal{D}'$, S' consists of many evaluation steps that may be interleaved (under control of GU) with actions of the protocol that gets updated. Moreover, several global updates can occur concurrently.

Local update A *local update* function LU in Fig. 3 takes as arguments a stack P, an old module a, and a new module b, and yields a new stack in which the new module is bound and the old one is passive. This has the effect of replacing a by b in stack P in one atomic action. After a call of LU returns, any calls of the service provided by a and b will use the new implementation b instead of a. However, any pending rounds can still complete using the old module.

The definition of global update does not specify *when* a function LU is actually called. Updating some protocols at "wrong" moment may invalidate safety properties of these protocols. In §5, we identify two safety properties (strong and weak) that cover a broad range of distributed protocols. Then we describe in §6 two implementations of GU; the first one satisfies strong safety, while the second one satisfies weak safety (but not strong safety).

5 Dynamic Update Correctness

The static replaceability property is necessary but not sufficient for DPU correctness. In this section, we define some safety properties that formalize what we regard to be correct DPU.

Correctness Intuitively, global update GU is correct if updating a distributed protocol does not interfere with the concurrent execution of the protocol, i.e. the update cannot be observed by any services of the protocol. In our model, the only observable actions of the distributed protocol are message outputs and deliveries (since they can modify state S). Obviously, correct global update must not cause the updateable protocol to loose nor duplicate messages. Some applications may also require that GU does not change the order of message delivery.

Judgments

$x \vdash_h S$	S is a correct message history of service x
$\vdash_{dpu} GU$	GU is a correct DPU algorithm

DPU correctness

$$\frac{S = (nil, nil)}{x \vdash_h S} \text{ (Null-History)}$$

$$\frac{\mathcal{D},\, S \overset{GU(\mathcal{D},a,b)}{\Longrightarrow} \mathcal{D}',\, S' \quad a : (x, R) \quad x \vdash_h S \quad x \vdash_h S'}{\vdash_{dpu} GU} \text{ (Correct-Update)}$$

$$\frac{x \vdash_h (S'_S,\ S'_D) \quad x \vdash_h (S''_S,\ S''_D) \quad S = (S'_S @ S''_S,\ S'_D @ S''_D)}{x \vdash_h S} \text{ (Consistent-Cut)}$$

DPU properties

$$\frac{\mathcal{D},\, S \overset{GU(\mathcal{D},a,b)}{\Longrightarrow} \mathcal{D}',\, (S'_S,\ S'_D) \quad \mathcal{D},\, S \Longrightarrow \mathcal{D},\, (S''_S,\ S''_D) \quad set(S'_S) = set(S''_S)}{set(S'_D) = set(S''_D)} \text{(Weak-Update)}$$

$$\frac{\mathcal{D},\, S \overset{GU(\mathcal{D},a,b)}{\Longrightarrow} \mathcal{D}',\, S' \quad \mathcal{D},\, S \Longrightarrow\xrightarrow{[m]P.b}\Longrightarrow \mathcal{D}'',\, S'' \quad S'' = (S''_S,\ S''_D)}{S''_S|a = S''_D|a} \text{(Strong-Update)}$$

Fig. 4. Judgments and DPU properties.

We define DPU correctness using two judgments, one for message histories, and one for the *GU* algorithm; the judgments and the rules for reasoning about the judgments are given in Fig. 4. The message history judgment has the form $x \vdash_h S$, read "S is a correct message history of service x". The algorithm correctness judgment has the form $\vdash_{dpu} GU$, read "*GU* is a correct DPU algorithm".

The rule (Correct-Update) says that global update of a service x with a DPU algorithm *GU* is *correct*, if given a distributed protocol $\mathcal{D}$ and a correct history of messages S, the algorithm would transform the system $\mathcal{D}$, S into system $\mathcal{D}'$, S' where S' describes a correct message history from the point of view of x.

The rule (Consistent-Cut) is a core rule for reasoning about message histories. It states that a message history constructed by appending two (possibly empty) histories that are themselves correct is also correct. The Lisp-like append operation $S_i @ S'_i$ returns a new list whose elements are the elements in the given lists S_i and S'_i, in the order that they appear in the argument lists.

Properties We can identify at least two update safety properties: strong and weak; they specify some desirable guarantees on a message history. We can then say that global update *GU* is *correct for updating a service x*, if it satisfies safety properties that are required by the correct execution of x. It depends on the semantics of x which property the implementation of *GU* should hold. (Obviously, we assert that updateable services must be themselves correct.)

Below are the two safety properties defined informally; a precise semantics is given in the bottom half of Fig. 4.

Property 1 (Weak Update) *A global update GU of a distributed protocol $\mathcal{D}$ has the* weak update *property if: (i) GU eventually terminates, (ii) if $\mathcal{D}$ does not get stuck, then the updated $\mathcal{D}$ will deliver exactly the same set of messages as the non-updated $\mathcal{D}$ would.*

The DPU algorithms that only satisfy the weak safety property cannot be used to update services that order messages. Below we define a stronger property.

Property 2 (Strong Update) *A global update GU replacing old modules by new modules in a protocol $\mathcal{D}$ has the* strong update *property if: (i) GU eventually terminates, (ii) after a new module has been used to deliver a message in some stack, the old module will never be used to deliver messages in any stack.*

Theorem 1 (Strong Update Correctness). *Global update that ensures the strong update property is correct.*

Proof. Consider update $\mathcal{D}, S \overset{GU(\mathcal{D},a,b)}{\Longrightarrow} \mathcal{D}', S'$, where a, b provide x and $x \vdash_h S$. Take any state $\mathcal{D}'', S''$ such that $\mathcal{D}, S \Longrightarrow .., X \xrightarrow{[m]P.b} .., Y \Longrightarrow \mathcal{D}'', S''$ and b is used a first time. Then

1. $x \vdash_h X$ by $x \vdash_h S$ and premise that up to this state x can only use $\delta(a)$,
2. $x \vdash_h Y \setminus X$ [1] by premise that b can replace a,
3. $x \vdash_h S'' \setminus X$ by 2. and $S''_S|a = S''_D|a$ (from definition of (Strong-Update)),
4. $x \vdash_h S''$ by 1. and 3. and (Consistent-Cut),
5. $\vdash_{dpu} GU$ by premise $x \vdash_h S$ and 4. and (Correct-Update). □

Below are two examples of services that can be updated with a DPU algorithm that has the strong update property.

Consider a bug-fix of a security protocol that is used by a distributed transactions service to encrypt transaction-related communication. After a local update action terminated in a stack, and the newly added change has been applied to the security protocol, the strong update property guarantees that no transaction (on any node) will commit using the old erroneous security protocol.

Consider services that must deliver messages in a certain order. The strong update property guarantees that the old module is used only until the new module is used (somewhere) for the first time. Up to this (global) time, all messages are delivered (with order) by the old protocol, after this time all messages will be delivered (also with order) using only the new protocol.

Theorem 2 says that weak update can be always replaced by strong update (the opposite is obviously not true); the proof is straightforward.

Theorem 2 (Strong Update implies Weak Update). *Any implementation of global update satisfying strong update, also satisfies weak update.*

[1] We write $Y \setminus X$ to denote a prefix of list Y obtained by removing sublist X.

Synchronized DPU

$$\frac{a\updownarrow\ \in P \quad P\{b/a\}}{P,\,S \xrightarrow{P.\text{ABcast}(\text{S1},a,b)} P,\,S} \qquad \text{(S1)}$$

$$\frac{\begin{array}{c}[\text{S1},\,a,\,b]\,P.abcast \quad abcast : (\text{ABcast}, ..)\\ P' = (P \setminus \{a\updownarrow\}) \cup \{a\uparrow\} \cup \{b\uparrow\}\end{array}}{P,\,S \longrightarrow P',\,S \xrightarrow{P.\text{ABcast}(\text{S2},a,b)} P',\,S} \qquad \text{(S2)}$$

$$\frac{S = (S_S,\ S_D) \quad S_S|P.a = S_D|P.a}{(a, P\ S)\ \textsc{Idle}} \qquad \text{(Idle)}$$

$$\frac{\begin{array}{c}[\text{S2},\,a,\,b]\,P.abcast \text{ from all } P' \in \mathcal{D}\\ (a, P\ S)\ \textsc{Idle}\end{array}}{P,\,S \xrightarrow{P.\text{ABcast}(\text{S3},a,b,\text{Idle})} P,\,S} \qquad \text{(S3)}$$

$$\frac{[\text{S3},\,a,\,b,\,\text{Idle}]\,P.abcast \text{ from all } P' \in \mathcal{D}}{P,\,S \longrightarrow (P \setminus \{a\uparrow\} \setminus \{b\uparrow\}) \cup \{b\updownarrow\},\,S} \qquad \text{(S4)}$$

Lazy DPU

$$\frac{\begin{array}{c}[m,\,b]\,P.c \ \text{ for some } c \in P\\ b \notin P \quad a\updownarrow\ \in P \quad P\{b/a\}\\ P' = (P \setminus \{a\updownarrow\}) \cup \{a\uparrow\} \cup \{b\updownarrow\}\\ S = (S_S,\ S_D)\end{array}}{P,\,S \longrightarrow P',\,S \xrightarrow{[m]P.b} P',\,(S_S,\ m^{P.b} :: S_D)} \ \text{(L1)} \qquad \frac{\begin{array}{c}[m,\,b]\,P.c \ \text{ for some } c \in P\\ b \in P\\ S = (S_S,\ S_D)\end{array}}{P,\,S \xrightarrow{[m]P.b} P,\,(S_S,\ m^{P.b} :: S_D)} \ \text{(L2)}$$

Fig. 5. Synchronized and lazy DPU algorithms.

6 Dynamic Update Algorithms

Consider updating a distributed protocol $\mathcal{D} = \{P \mid P = \{a\updownarrow, b\updownarrow, ..\}\}$ with a new module b [1]. Below we describe two example DPU algorithms. They are defined using a set of transition rules, each rule describing a single or double (atomic) evaluation step. The rules are expressed using the syntax in Fig. 1, extended with polyadic messages, i.e. a message is a sequence of names. For readability, we give in each rule only part of the state, i.e. the name of a local protocol stack in which a given action occurs (instead of $\mathcal{D}$). The steps of the algorithms can be freely interleaved with the steps of the protocol $\mathcal{D}$ being updated.

Synchronized update The *Synchronized Dynamic Protocol Update (S-DPU)* algorithm in the upper part of Fig. 5, updates a distributed protocol by replacing *old* modules by *new* ones. Firstly, it "passivate" bindings of the old and new modules in each stack so that the modules are passive. Then, the old module is

[1] Updating $\mathcal{D}$ with b may involve adding new modules to each stack, so that all services required by b are eventually provided; the algorithm is similar to L-DPU in §6.

removed and the new module is bound in every stack; this takes place locally only after it can be guaranteed that the old module is not needed anymore to complete any round of the distributed protocol.

To support concurrent global updates and termination under stack crashes, our algorithm communicates control messages using a *totally ordered broadcast* [7, 14] service ABcast. We assume *abcast* to be some implementation of ABcast. Execution of ABcast(m), where m is a fresh message, broadcasts m to all stacks with a guarantee that the round of ABcast terminates and if some stack delivers m before another broadcast message m', then every stack delivers m before m'.

Formally, if we take any two stacks $P, P' \in \mathcal{D}$ and modules $a \in P$ and $a' \in P'$ of type (ABcast, R) for some R, then for any state $\mathcal{D}$, S and $S = (S_S, S_D)$ such that $(a, (\mathcal{D},\, S))$ FREE and $(a', (\mathcal{D},\, S))$ FREE, we have $S_D|P.a = S_D|P'.a'$, where the FREE property is defined in Fig. 2.

Below are steps of the S-DPU algorithm. Note that the output and delivery of update-related control messages *do not modify message histories*!

S1. Broadcast a fresh message (S1, a, b) to all stacks, where module a is bound in the local stack P and replaceable in P by module b. (We assume that initially, i.e. when a message history is (nil, nil), all stacks are identical.)

S2. Upon receipt of (S1, a, b), passivate module a in the local stack P and extend P with passive module b. Then, broadcast a fresh message (S2, a, b).

S3. A module a of stack P is *idle*, denoted $(a, P\ S)$ IDLE where S is a message history, if all messages sent to a (by any stack) have been delivered by a [1]. Upon receipt of (S2, a, b) from all stacks, wait until module a is idle in the local stack P, then broadcast a fresh message (S3, a, b, Idle).

S4. Upon receipt of (S3, a, b, Idle) from all stacks, remove module a from the local stack P and bind module b.

Lemma 1 (Safe Rebinding) *If S-DPU algorithm binds a new module in some state, then a module being replaced with the new module is free in this state.*

Proof. Consider binding of some module a in step (S4) of S-DPU. Then

1. by premise of (S4) and ABcast, each stack $P \in \mathcal{D}$ has executed (S3)
2. by 1. and (S3) and definition of $(a, (P,\, S))$ IDLE, each stack $P \in \mathcal{D}$ has been in a state $\mathcal{D}$, S, such that $S_S|P.a = S_D|P.a$,
3. by premise of (S3) and ABcast, each stack has executed (S2),
4. by 3. and (S2), each stack has unbound a in (S3), so $S_S|P.a = S_D|P.a$ is true not only in (S3) but also in (S4),
5. by 4. and premise of (S4) and ABcast, $S_S|P.a = S_D|P.a$ for all stacks $P \in \mathcal{D}$,
6. by 5. and definition of $S_i|a$ $(i = S, D)$ in Fig. 1, $S_S|a = S_D|a$,
7. by 6. and (Freedom), $(a, (\mathcal{D},\, S))$ FREE. $\square$

We conclude that the S-DPU algorithm satisfies strong update.

[1] We assume the existence of a *global snapshot* algorithm [11] to determine this predicate.

Theorem 3 (S-DPU Strong Safety). *Updating a distributed protocol with the S-DPU algorithm satisfies strong update.*

Proof. By Lemma 1 and (S4), when a new module is bound in $\mathcal{D}$, S, the old module a is free. By (S2), a is unbound in $\mathcal{D}$, S, i.e. for any state $\mathcal{D}'$, S' following $\mathcal{D}$, S, we have $S'_S|a = S'_D|a$. By (Strong-Update) this completes the proof. □

Lazy update The *Lazy Dynamic Protocol Update (L-DPU)* algorithm in the bottom of Fig. 5, updates a distributed protocol lazily, by extending stacks with a new module whenever needed.

We associate messages with modules that are used to deliver the messages. If a module required to deliver a message is not in a local stack, then it is added to the stack, bound, and the binding of the old module providing the same service is "passivated", so that any new protocol round in this stack can use the new module. The algorithm allows however the old and new modules to coexist in the distributed protocol, i.e. they can deliver their messages concurrently.

The L-DPU algorithm does not require any distributed infrastructure, except the one used by protocol $\mathcal{D}$ to communicate messages via network. Thus, it scales to large networks. Below are actions of the L-DPU algorithm.

L1. Upon delivery of a message (m, b) by some module c in the local stack P, if module b is not in P, then take any module a in P that is bound and replaceable by b, passivate a and bind b. Finally, deliver m using b.
L2. Upon delivery of a message (m, b) by some module c, if module b is available locally, then deliver m using b.

To guarantee termination of the global update, we could require that stacks periodically broadcast and deliver an "update" message containing new modules.

According to Theorem 4, the Lazy DPU algorithm guarantees that all protocol messages are delivered but message ordering is not preserved.

Theorem 4 (L-DPU Weak Safety). *Updating a distributed protocol with the L-DPU algorithm satisfies weak update.*

Proof. Straightforward by (Weak-Update) and atomicity of rebinding in (L1). □

7 Practical Experiment

To facilitate experimentation, we have designed and implemented DPU support for Fortika [13, 14] – a group communication middleware that is developed within our project. We have encoded middleware components using the SAMOA library [24]. The most complex components that Fortika uses are two agreement services: *distributed consensus* and *totally ordered broadcast* (ABcast). We have proposed DPU algorithms that can switch between different implementations of these services dynamically, while preserving safety properties of each service. By exploring the semantics of consensus and ABcast, the DPU algorithms can be less synchronous than the S-DPU algorithm in §6. In effect, the service is available almost continuously while it is updated.

Consider update of the consensus service [5]. The service ensures that given a group of distributed processes, after a round of consensus, all processes would agree on the same value, which has been chosen from values proposed individually by each process. Our DPU algorithm uses the semantics of consensus for replacement of the consensus implementation; it has three steps. Firstly, an intend to replace a consensus protocol $\delta(a)$ by $\delta(b)$ is broadcast. Then, all processes must decide when b can be bound locally. For this, b could be piggybacked on any message that must be also processed by the consensus service. Finally, when the decision about b has been delivered (that means *all* stacks reached consensus about binding b), a is passivated and b is bound. The time between binding a new module and making the old one passive is therefore maximally reduced.

The results of our practical experiment demonstrate that dynamic replacement of network protocols in a group communication system can be done efficiently. Description of the DPU algorithms for agreement protocols and performance measurements are in our companion paper [16].

8 Related Work

In this section, we describe some of the work most closely related to ours.

There are quite a number of implementations that support dynamic updating of software components. For example, the Erlang programming language [1] allows software modules to be replaced at runtime, however with no safety guarantees. A Java HotSpot VM [20] allows a class instance to be replaced with the new instance in a running application through the debugger APIs.

There have been work on safe dynamic software updating by construction, ensuring that if an update is accepted by the system, then the resulting program will be type-correct. Dynamic ML [22] enables type-safe module replacement at runtime; changes can include the alternation of abstract types at update-time, and the addition (and possibly removal) of module definitions via garbage-collection. Dynamic Java classes [12] 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 [8] 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. Bierman *et al.* [3] study dynamic software updating with a small extension of a lambda calculus that supports an Erlang-like updating features. A preliminary discussion of safety properties is included, however without considering the use of concurrency and coordinated updates. Stoyle *et al.* [19] investigate type-safe dynamic updating in C-like languages. However, this work does not address the issues of global coordination of local updates.

Few systems offer support for coordinating local updates. For example, Van Renesse *et al.* [21] describe a switching protocol, which synchronizes dynamic replacement of protocols in the Ensemble protocol framework, however it does so only for whole stacks, thus blocking applications on top of the stack during update. Chen *et al.* [6] 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. A similar solution has been proposed in [18], but it uses a centralized manager, which limits its scope of applicability. However, in none of the above systems is there any well developed evidence as to what conditions are needed to guarantee the correctness of updating distributed protocols on-the-fly.

To date relatively little work has been carried out on formalization of dynamic protocol update. 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. While we have identified space between lazy and synchronized updates, they only describe one example switching protocol. The algorithm is correct only for replacement of protocols that must exhibit all their (six) meta-properties; it cannot be applied for arbitrary protocols, contrary to the S-DPU algorithm presented in this paper.

Methods of distributed versioning, such as Sewell's [17] fine-grain versioning control of values of abstract types, could be used to support interoperation of old and new modules, and e.g. verify statically the replaceability property.

9 Conclusions and Future Work

In this paper we make several contributions. We have defined a simple but expressive model of dynamic protocol update (DPU). We use our model to define static and dynamic requirements that, we believe, should be considered by any valid dynamic protocol update support:

– The replaceability property specifies minimal structural, static requirements on module replacement;
– The strong and weak update safety properties specify that updating a distributed protocol must not cause message loss; the strong property additionally requires that message order is always preserved.

Based on the above requirements, we have constructed two DPU algorithms which are based on synchronized and lazy updating strategies. The former algorithm exhibits strong safety guarantees but requires a subtle distributed infrastructure (totally ordered broadcast) which does not scale to large networks. The latter algorithm scales well but the order of message delivery by updateable service is not respected that limits its applicability.

Our DPU algorithms work correctly also in the presence of stack crashes, in the sense that all non-crashed stacks are guaranteed to get eventually updated.

In the future work, it may be worthwhile to extend the model presented in this paper with system failures and message omissions; this would allow us to reason about such cases formally.

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. T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.
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*, Apr. 2001.
7. X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Tech. Report DSC-2000-036, Communication Systems Department, EPFL, Sept. 2000.
8. 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.
9. M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proc. PLDI '01: The ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.
10. S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. OOPSLA '98*, Oct. 1998.
11. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1997.
12. 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.
13. S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. Cactus: Comparing protocol composition frameworks. In *Proc. SRDS '03*. IEEE, Oct. 2003.
14. 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.
15. L. Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, Dec. 1998.
16. O. Rütti, P. T. Wojciechowski, and A. Schiper. Dynamic update of distributed agreement protocols. Tech. Report IC-2005-012, I&C, EPFL, Mar. 2005.
17. P. Sewell. Modules, abstract types, and distributed versioning. In *POPL '01*, 2001.
18. N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proc. ICDCS '03*, May 2003.
19. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *POPL '05*, Jan. 2005.
20. Sun Microsystems, Inc. *Java HotSpot.* http://java.sun.com/products/hotspot/.
21. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software Practice & Experience*, 28(9), 1998.
22. C. Walton, D. Kirli, and S. Gilmore. An abstract machine model of dynamic module replacement. *Future Generation Computer Systems*, 16:793–808, May 2000.
23. P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In *Proc. Coordination '02*, LNCS 2315, Apr. 2002.
24. 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.