

Protocol Switching: Exploiting Meta-Properties

Xiaoming Liu, Robbert van Renesse, Mark Bickford
Christoph Kreitz, Robert Constable

Department of Computer Science
Cornell University, Ithaca, NY 14853
work: (607) 255-1021; fax: (607) 255-4428
{xliu,rvr,markb,kreitz,rc}@cs.cornell.edu

Abstract

As we see a growing variety of network and application behaviors, it becomes more important that protocols adapt to their surroundings. Building adaptive protocols is complicated, and therefore we have considered building hybrid protocols that switch between specialized protocols. In this paper, we show for which communication properties this is a correct solution, and classify these using a new concept called meta-properties. We also show how well these switches perform.

1 Introduction and Related Work

Many networking protocols accomplish the same thing, such as recovery from message loss, but are optimized for different environments or applications. Developing hybrid protocols that combine the advantages of the various protocols is difficult, and the resulting protocols are complex. As a result, most existing adaptive protocols only change certain run-time parameters such as flow window size in TCP [7] or energy consumption in mobile network [9], but not the actual logic of the protocol.

Most protocols that do change the logic have focussed on building specialized hybrid protocols. For example, H-RMC [10] has investigated a hybrid between rate and credit-based flow control protocols, while Rodrigues describes a hybrid total ordering protocol in [11].

A different approach is to switch between protocols at run-time when necessary. For example, in [5] the authors discuss the benefits of switching dynamically between primary-backup and the state machine approach to replication. Applying this to protocols in general was pioneered in the Horus [12], Ensemble [13], and Cactus [6] systems, but it was never quite clear whether the result

was actually *correct*. In this paper we will take a closer look at this.

We designed a new generic switching protocol, and determined which communication properties are preserved by it. For this, we introduced *meta-properties* (i.e., properties of properties) in order to classify communication properties. We have focussed on group multicast protocols, since many interesting properties for such protocols exist, but our work can easily be specialized for point-to-point communication.

The kinds of uses we envision include the following:

- *Performance*. By using the best protocol for a particular network and application behavior, performance can always be optimal.
- *On-line Upgrading*. Protocol switching can be used to upgrade networking protocols at run-time without having to restart applications. Even minor bug fixes may be done in this way.
- *Security*. System managers will be able to increase security at run-time, for example when an intrusion detection system notices unusual behavior, or when it gets close to April 1st.

Without loss of generality, we will assume we will be switching between two protocols. We are not concerned here with which protocol is best, but just with preserving communication properties under switching. Which protocol is best at any time is an orthogonal problem. We assume that some kind of oracle decides when a switch is necessary.

A *switching protocol* (SP) is basically yet another protocol layered over the two protocols of interest. The application only interacts with the SP. The SP is supposed to be *transparent*, that is, the application cannot tell easily that it is running on the SP rather than on one of the underlying protocols, even as the SP switches between protocols.

⁰This work is supported in part by ARPA/ONR grant N00014-92-J-1866, DARPA/AFRL-IFGA grant F30602-99-1-0532, NSF-CISE grant EIA 97-03470, the AFRL-IFGA Information Assurance Institute, Microsoft Research, and Intel Corporation.

2 The Switching Protocol

Although many switching protocols are possible, and some may preserve different communication properties, we will be focusing in this paper on just one. This SP guarantees that, when switching from one protocol to another, any process will deliver all messages for the previous protocol before delivering messages for the new one. This SP works in one of two modes. In *normal mode*, when the application submits a message for sending to the SP, the SP in turn offers the message to the current protocol. Whenever receiving a message from the current protocol, the SP simply forwards the message to the application. However, when the oracle requests the SP to switch (at one of the processes called the *manager*), the SP goes into *switching mode*.

First, the manager broadcasts a PREPARE message to the other members. On receipt, a member returns an OK(member, count) message that includes the number of messages that the member has sent so far over the current protocol. New data messages will be sent over the new protocol, and messages received over this protocol will be buffered rather than delivered.

The manager awaits all OK messages, and then broadcasts a SWITCH(vector) message, including a vector with the message send count of each member. On receipt, a member knows how many messages it should have delivered from each other member. When it has received and delivered all messages of the current protocol from each member, the member switches over to the new protocol, and delivers any messages that were buffered.

Note that the SP makes a number of assumptions about the underlying protocols. In particular, it assumes that all messages that are delivered were sent (no spurious deliveries), and that messages are delivered at most once. If switches are supposed to complete (*liveness*), messages have to be delivered exactly once.

In order to avoid congestion on the network, our implementation of SP does not actually do network-level broadcasts, but rotates a token message in a logical ring of the group members. As a bonus, it also avoids complicating issues with multiple members trying to switch protocols concurrently. The token itself has a mode based on the phase of the protocol. A member that wants to initiate a switch has to await a NORMAL token. This member, henceforth called the *initiator*, will then change the token to a PREPARE token, and sends it around the ring. Every receiver will act the same way as if it received the PREPARE message described above, and piggybacks the OK message on the token. When the initiator receives the token, it now knows all the message counts and changes it into a SWITCH(vector) token in order to disseminate these.

Reliability	Every message that is sent is delivered to all receivers
Total Order	Processes that deliver the same two messages deliver them in the same order
Integrity	Messages cannot be forged; they are sent by trusted processes
Confidentiality	Non-trusted processes cannot see messages from trusted processes
No Replay	A message body can be delivered at most once to a process
Prioritized Delivery	The <i>master process</i> always delivers a message before any one else
Amoeba	A process is blocked from sending while it is awaiting its own messages
Virtual Synchrony	A process only delivers messages from processes in some common view

Table 1: Examples of properties

When this token comes back to the initiator, it changes the token into a FLUSH token, and sends it around the ring once more. Unlike the other tokens, a member only forwards this token if it has delivered all messages from the old protocol. Thus, when the token comes back to the initiator, the switch has truly completed at each member, and the initiator can change the token back into a NORMAL token. Thus the token has to travel around the ring three times in order to execute the entire switch.

3 System Model

In order to see what properties are preserved by SP, we will develop a model of a distributed system and its properties.

Processes multicast messages that contain a body and a sender. We will consider two types of events. A $Send(m)$ event models that process $m.sender$ has multicast a message m . A $Deliver(p : m)$ event models that process p has delivered message m . A *trace* is an ordered sequence of $Send$ and $Deliver$ events such that there are no duplicate $Send$ events. A property is a predicate on traces, dividing all traces into two categories: those traces for which the property holds, and those for which it does not. See Table 1 for a collection of examples of properties.

A protocol is a module available at every process that implements certain properties on behalf of the set of processes. It can be thought of as having a top and a bottom side, applications sitting at the top, and the network sitting at the bottom. Applications submit $Send$ events to it, and the protocol submits $Send$ events to the network below it. Vice versa, the network submits $Deliver$ events to the

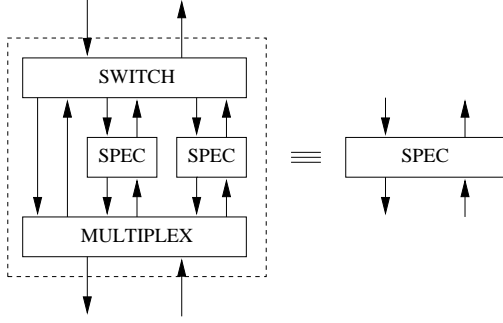


Figure 1: The composed protocol has to preserve the specification SPEC.

protocol, and the protocol submits *Deliver* events to the application.

This symmetry makes it possible for protocols to be composed by layering them on top of one another, much like Lego[™] blocks. In effect, protocols are closed under composition: a *stack* of protocols is another protocol. It is also possible to view the application and the network as instances of protocols. In the context of a stack, we call a protocol a *layer*. Note that every process is required to have the same stack of layers.

Layers can be loosely divided into two types: those that guarantee properties for the layers above them, and those that guarantee properties for the layers below them (some may do both). In this paper, we will only be interested in the first type.

4 Components

We are interested in investigating which properties are preserved by SP. More precisely, we are interested in the properties of a communication system composed of a set of components, as depicted in Figure 1. Each rectangle in this figure is the specification of some protocol layer. In this figure, there are three types of specifications:

- *SWITCH*. This component is the specification of SP.
- *SPEC*. The specification of some interesting protocol, such as a protocol that provides total order or integrity.
- *MULTIPLEX*. A protocol that simulates multiple connections over a single communication channel.

The figure shows that a composition with SP has to have the same specification as that of the protocols of interest. Notice that SWITCH requires a private communication channel for itself, while each “underlying” protocol also needs a private channel. Each specification has

incoming *Send* and *Deliver* events, as well as outgoing *Send* and *Deliver* events.

Interestingly, several of the difficulties with the composition are *not* because of switching, but because of delays incurred by *layering*. These delays re-organize event traces and can potentially violate properties.

5 Meta-properties

In this section, we introduce the concept *meta-property*, which is a predicate on properties and can therefore classify them. As we will see, we can use this to classify which properties are preserved by protocol switching. For the purpose of this paper, we are interested in a particular form of meta-property which is defined by preservation of properties through a protocol layer. A property P of traces is *preserved* by a (reflexive and transitive) relation R on traces if whenever traces tr_{above} and tr_{below} are related by R , and P holds of trace tr_{below} , then it also holds of trace tr_{above} .¹ Formally (for all tr_{below}, tr_{above}):

$$P(tr_{below}) \wedge tr_{above} R tr_{below} \Rightarrow P(tr_{above}) \quad (1)$$

Preservation by R is thus a meta-property. In this section, we will discuss four such relations that deal with delays in a layered communication system. Later, we will discuss two additional relations that deal with the actual switching.

5.1 Safety

Safety [2] is probably the best-known meta-property. Safety means that if the property holds for a trace, then it is also satisfied for every prefix of that trace. An example of a safe property is total order: taking events of the end of a trace cannot reorder message delivery. As an example of a property that is not safe, consider *reliability*. A reliable trace is one in which all sent messages have been delivered everywhere. However, if we chop off a suffix containing a *Deliver* event without the corresponding *Send* event, the resulting trace is no longer reliable. The corresponding relation R for safety in Equation (1) is R_{safety} which specifies that tr_{above} is a prefix of tr_{below} .

A non-safety property may not be preserved by switching. So far, however, we have only come up with fairly contrived examples. For example, consider the property “every second message is eventually delivered.” If an application sends two messages, and a switch occurs in between, the property may well be violated since the un-

¹Note that here we focus on properties to the layer above, not to the layer below, but our technique can be easily generalized.

derlying protocols have no requirement to deliver either message.

5.2 Asynchrony

Any global ordering that a protocol implements on events can get lost due to delays in the send and deliver streams through layers above. Only properties that do not depend on the relative order of events at different processes are preserved under the effects of layering. The corresponding relation $R_{asynchrony}$ specifies that two traces are related if they can be formed by swapping events that are adjacent and that belong to different processes. Events belonging to the same process may not be swapped. An example of a property that is not asynchronous is *Prioritized Delivery*, and this property is in fact not preserved by our switching protocol.

5.3 Delayable

Another effect of delay by a layer is local: at any process, *Send* events are delayed on the way down, and *Deliver* events are delayed on the way up. For example, when an application sends two messages in a row, and then another message is delivered to it, these events may well happen in a different order below the layer directly underneath the application. A property that held there may no longer be true at the application.

We call a property that survives these delays *delayable*. (This property is similar to *delay-insensitivity* in asynchronous circuits.) The corresponding relation $R_{delayable}$ specifies that adjacent *Send* and *Deliver* events in tr_{below} may be swapped in tr_{above} if they belong to the same process. An example of a property that is not delayable is *Amoeba* [8], and this property is indeed not preserved by switching.

5.4 Send Enabled

A protocol that implements a property for the layer above typically does not restrict when the layer above sends messages. We call a property *Send Enabled* if it is preserved by appending new *Send* events to traces. The corresponding relation $R_{send-enabled}$ specifies that tr_{below} and tr_{above} are related if tr_{above} is formed by adding only *Send* events to the end of tr_{below} .

In practice, *Send Enabled* and *Delayable* are related. Both are concerned with not being able to control when the application sends messages. For example, the *Amoeba* property is neither *Delayable* nor *Send Enabled*.

	Safety	Asynchronous	Send Enabled	Delayable	Memoryless	Composable
Total Order	+	+	+	+	+	+
Integrity	+	+	+	+	+	+
Confidentiality	+	+	+	+	+	+
Reliability	-	+	-	+	+	+
Prioritized Delivery	+	-	+	+	+	+
Amoeba	+	+	-	-	+	-
Virtual Synchrony	+	+	+	+	-	+
No Replay	+	+	+	+	+	-

Table 2: Which properties satisfy which meta-properties?

6 Hybrid Protocols

The four meta-properties discussed so far, *Safety*, *Asynchrony*, *Send Enabled*, and *Delayable*, deal with surviving the effects of delay in a layered environment. Since SP does introduce delays, these meta-properties are also useful for a property to be preserved by SP. But they are not sufficient. Below, we present two more meta-properties.

6.1 Memoryless

A property is *memoryless* if we can remove all events pertaining to a particular message from it without violating the property. $R_{memoryless}$ defines that tr_{above} can be formed from tr_{below} by removing all events related to certain messages from it. That is, whether such a message was ever sent or delivered is no longer of importance. This does not imply that the protocol is *stateless* and can forget about the message, however. Consider, for example, *No Replay*, which is memoryless but certainly any implementation cannot be stateless.

The *Virtual Synchrony* property is not memoryless, and indeed switching between virtually synchronous executions does not guarantee that the resulting execution is virtually synchronous.

6.2 Composable

A property is *composable* if for any two traces for which the property holds and which have no messages in common, the property also holds for the concatenation. An example of a property that is not composable is *No Replay*: even if a message body is delivered at most once in tr_1 and tr_2 , and tr_1 and tr_2 have no messages in common, the message *body* may be delivered twice in the concatenation. This may well happen when switching between

two protocols that each guarantee, individually, No Replay.

6.3 Discussion

Our SP can support, at least, those properties that have all meta-properties discussed so far. Using the Nuprl theorem prover [1], we have shown that these six meta-properties are sufficient [3]. From Table 2 we can see that this class includes many interesting communication properties. Some properties outside of this class, like Reliability, are also preserved by our switching protocol. Nevertheless, we believe that the class we have defined is fairly "tight," because each meta-property describes an important restriction:

1. Safety: Liveness properties require that the input satisfy some fairness condition. Since we divide the input between the two protocols, we no longer guarantee that the fairness condition holds.
2. Asynchrony: we need this because delays in distributed systems can re-order global orderings, even between delivery operations.
3. Delayable: we need this because SP introduces a delay which can re-order local orderings between send and deliver operations.
4. Send Enabled: we need this because when we switch between protocols, any restriction on the relative order of sending is lost.
5. Memoryless: we need this because when we switch between protocols, a protocol may not see part of the history of events, and thus has to be able to work as if these events never happened.
6. Composable: we need this because when we switch between protocols, we are going to glue traces together. The result will still have to satisfy the property.

7 Performance

In this section, we will have a brief look at the performance implications of using our switching protocol. As an example, we will look at switching between total ordering protocols. There are two well-known mechanisms for implementing total order. The first uses a centralized sequencer. Messages are sent in FIFO order to the sequencer, and then the sequencer forwards these messages

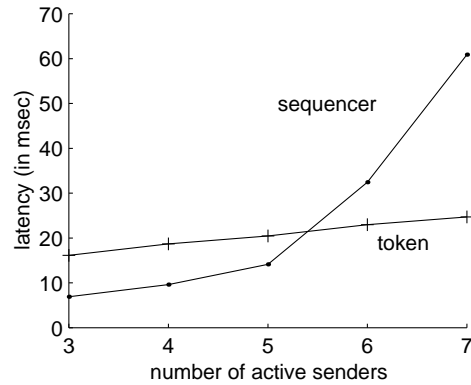


Figure 2: Message latency vs. number of active senders. In this case, the size of the group is 10 members, and the message rate for each sender is 50 msgs/sec.

by multicast, again in FIFO order [8]. The second technique uses a rotating token with a sequence number. Processes that wish to multicast have to await the token before they can send. The sequence number on the token is incremented in that case [4].

These two mechanisms have an interesting trade-off. The sequencer-based algorithm has low latency (basically twice the network latency), but the sequencer may become a bottleneck when there are many active senders. The token-based algorithm does not have a bottleneck, but the latency is relatively high under low load since processes have to await the token before they can send. This trade-off is clearly visible in Figure 2. In this experiment we have a group of ten processes (on a variety of SparcStation-20s running Solaris, and connected by a 10 Mbit Ethernet). A subgroup of varying size is sending 50 messages per second per member. In this case, there is a cross-over point when the size of the subset is between 5 and 6 active senders.

Clearly, a hybrid protocol formed by switching at the cross-over point would achieve the best of both worlds. However, some care needs to be taken in practice. If switching too aggressively, the resulting protocol starts oscillating. If we make our protocol less aggressive (by adding a hysteresis), we ran into an unexpected hitch. The overhead of switching depends on the latency of the current protocol (the one that is being switched away from), since the SP waits until all the corresponding messages have been delivered. If we wait too long with switching, the latency of switching will depend heavily on this. In the case of Figure 2, the overhead of switching near the cross-over point is about 31 msec. Processes are never blocked from sending during switching, so the perceived hiccup is often less than that.

8 Conclusion and Future Work

We have shown that constructing adaptive protocols out of specialized protocols is feasible and worthwhile, and that, given the switching protocol that we use, the class of communication properties for which this works is interesting. A formal treatise, which includes a proof of correctness, can be found in [3]. This work has limited itself to investigating safety properties, but we would also like to consider liveness properties. Also, we would like to investigate other switching protocols that possibly can support different classes of properties. For example, virtually synchronous view changes [12] can be used to switch protocols, and this more complicated mechanism does support the Virtual Synchrony property.

Acknowledgments

We would like to thank Idit Keidar for a discussion we had about this work. We also thank Mark Hayden for having the idea to switch between protocols, and Jason Hickey and Ken Birman for inspiration. Finally, thanks to the anonymous reviewers for comments that greatly improved the paper.

References

- [1] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The nuprl open logical environment. In D. McAllester, editor, *17th Int. Conf. on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.
- [2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- [3] M. Bickford, R. Van Renesse, X. Liu, C. Kreitz, and R. Constable. Proving hybrid protocols correct. 2000. In preparation.
- [4] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.
- [5] L. Gong and J. Goldberg. Implementing adaptive fault-tolerant services for hybrid faults. Technical Report SRI-CSL-94-04, CS Laboratory, SRI International, Menlo Park, CA, March 1994.
- [6] M. Hiltunen and R. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):125–133, September 1996.
- [7] V. Jacobson. Congestion avoidance and control. In *Proc. of the Symp. on Communications Architectures & Protocols*, Stanford, CA, August 1988. ACM SIGCOMM.
- [8] M. F. Kaashoek, A. S. Tanenbaum, S. Flynn-Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
- [9] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. of the 5th ACM/IEEE Mobicom Conf.*, Seattle, WA, August 1999.
- [10] P. K. McKinley, R. T. Rao, and R. F. Wright. HRMC: A hybrid reliable multicast protocol for the Linux kernel. In *Proc. of the Conf. on Supercomputing '99*, 1999.
- [11] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proc. of the 16th Int. Conf. on Distributed Computing Systems*. IEEE, 1996.
- [12] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the Fourteenth ACM Symp. on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.
- [13] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, August 1998.