Concurrency in Microprotocol Frameworks

Péter Urbán[1], Sergio Mena[2], Xavier Défago[3,4],
and Takuya Katayama[3]

[1]Facultad de Informática, Universidad Politécnica de Madrid, Spain
[2]Faculté Informatique & Communications,
École Polytechnique Fédérale de Lausanne, Switzerland
[3]School of Information Science, Japan Advanced Institute of Science and Technology
[4]"Information & Systems," PRESTO, Japan Science and Technology Agency

February 28, 2006
IS-RR-2006-004

Research Report

JAIST

School of Information Science
Japan Advanced Institute of Science and Technology

ISSN 0918-7553

Concurrency in Microprotocol Frameworks

Péter Urbán[†] Sergio Mena[‡] Xavier Défago[§⋆] Takuya Katayama[§]

[†]Universidad Politécnica de Madrid, Spain
Email: urban@jaist.ac.jp

[‡]École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Email: sergio.mena@epfl.ch

[§]Japan Advanced Institute of Science and Technology
Email: {defago,katayama}@jaist.ac.jp

[⋆]PRESTO, Japan Science and Technology Agency (JST)

Abstract

Protocol stacks and other distributed applications have been structured as a set of collaborating components with more or less well-defined interfaces. Recent frameworks provide flexible interfaces, arrangements and communication patterns, and thus allow for finer-grained components, called microprotocols*. Multi-threaded programming is the key to high performance in these frameworks. This paper investigates what support for multi-threaded programming such frameworks provide and should provide for programmers. Along with a survey and detailed discussions of the features of existing frameworks, we propose features that can be offered without significant changes in programs, and that have a negligible performance impact. This includes the following: (1) sets of single-threaded microprotocols that coexist with multi-threaded microprotocols, thus taking the best of two worlds; (2) non-overlapping execution of microprotocols involved in a chain of asynchronous communication, to avoid inconsistencies; and (3) ordering guarantees for asynchronous communication among microprotocols. To our knowledge, our definition for a particular extension of causal order is the simplest so far.*

Keywords: *Protocols, distributed applications, microprotocol frameworks, middleware, components, concurrency, asynchronous communication, causal order.*
Technical areas: *Operating Systems and Middleware*
Corresponding author's email: *urban@jaist.ac.jp*

1. Introduction

Context. For several decades, protocol stacks (and more complex distributed applications) have been structured as a set of collaborating components with more or less well-defined interfaces. In traditional protocol stacks, these components are called layers and their arrangement and communication pattern is fixed: each layer can only communicate with the layers directly above and below. At every level, a layer uses the one below it as a virtual medium to communicate to its peer layers at other sites, offering itself, in turn, as an enhanced virtual medium to the layers above.

Newer research frameworks, called *microprotocol frameworks* in this paper, extend this model by having more flexible interfaces, arrangements and communication patterns, thus permitting the use of finer grained components, called *microprotocols*. The basic promise that modular microprotocol frameworks make is a full separation of concerns between programming microprotocols and composing them, to the extent that these two tasks can be carried out by different people with a minimal interaction.

Multiprocessor machines are commonly used as servers, and are increasingly present on the desktop, as well. The dominant programming model that can fully take advantage of multiple processors is called multi-threaded programming.[1] This programming model is extensively used in microprotocol frameworks.

Contribution. Multi-threaded programming is significantly harder than single-threaded programming. Hence it is worthwhile investigating what support is available for programmers of distributed applications. This paper concentrates on what support microprotocol frameworks pro-

1 Specialized languages, compilers, and software and hardware environments may provide concurrency that, unlike multi-threading, is transparent to the programmer. This is not the case with the most popular languages like Java and C, their software environments, and common desktop, server and embedded hardware.

vide to programmers. We survey the features of existing frameworks for multi-threaded programming, and extensively discuss the usefulness of each of the features. We then propose a set of features that can be offered without significant changes in programs, and that have a negligible performance hit. Among other things, we propose the following features: (1) sets of single-threaded microprotocols that can coexist with multi-threaded microprotocols, thus taking the best of two worlds; (2) non-overlapping execution of microprotocols involved in a chain of asynchronous communication, to avoid inconsistencies; and (3) providing ordering guarantees for asynchronous communication among microprotocols, including first-in-first-out (FIFO) and causal order, and an extension to causal order. To our knowledge, our definition for this particular extension of causal order is the simplest so far.

Structure. The paper is structured as follows. Section 3 lists the microprotocol frameworks we investigated, as well as other related work. Section 2 introduces our terminology for frameworks and their main features. Section 4 presents models of concurrency. Section 5 compares the models and proposes a combination. Section 6 is concerned with overlapping executions of microprotocol code. Section 7 defines and discusses ordering guarantees for the communication between microprotocols and their implementation. Finally, Section 8 concludes the paper.

2. Terminology and features of microprotocol frameworks

The terminology used for describing microprotocol frameworks varies wildly from paper to paper. Also, the same terms often mean very different things for different people. Hence we felt it was necessary to introduce a uniform terminology, along with describing the usual features of such frameworks. Throughout this paper, we use the terms as defined in this section, even if the definition of the same term differs in papers describing the frameworks.

Microprotocols. The code that implements protocols in such frameworks is organized into units called *microprotocols*. Microprotocols are software components, i.e., it is clear what services they offer to other microprotocols, and they usually use a few standard communication mechanisms. The basic promise that microprotocol frameworks (and general component frameworks) make is that people other than the programmers of microprotocols are able to plug microprotocols together, without having to know or change the code of the microprotocols. We call such people *composers*. The set of all microprotocols and their interconnections is called the *composition*.

Asynchronous communication. The main form of communication between microprotocols is asynchronous (even for communication within a process). The data transferred during an instance of communication is called an *event*. Initiating the communication is called *sending* an event; the initiator is called the *sender thread* that executes the *sender microprotocol*. Sending an event results in *handling* the event: some piece of code, called *handler*, is executed; the *handler* is part of the *handler microprotocol*.

The fact that communication is asynchronous means that (1) handling the event may take place after sending is finished; (2) sending never blocks; (3) and sending never returns data from the destination handler, such as a return value or an exception that the sender thread is expected to handle (but, of course, the handler microprotocol might send another event to the sender microprotocol if such communication is necessary).

In our terminology, an event only exists for the duration of the communication. If the handler microprotocol decides to communicate with another microprotocol, we consider the associated event a new event that is different from the original one. Of course, the new event might carry the same information as the original event.

Let us put asynchronous communication into context. Asynchronous communication between microprotocols is widespread because it maps to the message passing model of communication over networks. It often allows for a greater decoupling of microprotocols than synchronous communication, just like message passing allows for greater decoupling than remote procedure calls. The focus on asynchronous communication is an important feature that distinguishes microprotocol frameworks from general component frameworks.

Communication over the network. Protocols and distributed applications usually have code in distinct address spaces. We call these address spaces *processes*. The communication mechanisms for microprotocols do not work between processes: a networking technology, such as IP networking, must be used. In this paper, we focus on communication within a given process: in particular, concurrency issues that arise within a process. Communication between processes is much slower and often unreliable, and thus concurrency issues require different solutions. Communication between processes appears in our model as special microprotocols that do communication over the network.

3. Related work

Microprotocol frameworks. We first list the microprotocol frameworks we investigated for their features related to concurrency. We concentrated on frameworks that are still in use.

- Neko 0.9 [15, 16]. The Neko framework is capable of both simulating and executing protocols, using the same source code. It comes with an extensive set of protocols.
- Cactus/C 2.2 (C version), Cactus/J 2.0 (Java version) and a later, non-public Java version (received 2004/04) [6, 13]. We consider the Cactus framework because of its maturity (incorporates 10 years of experience) and elegant, minimal design.

  Cactus has a two-level hierarchy of components, and the components interact very differently within each level. For this reason, we often present Cactus as two frameworks: *Cactus/μp* and *Cactus/cp* refer to how things work at the lower and the higher level, respectively (μp stands for microprotocol and cp stands for composite protocol, the Cactus terms for components at the two levels). Cactus/cp is rather similar to an influential early framework, the x-kernel [7].

  The concurrency features of the C and Java version of Cactus/μp are rather different. We will refer to them as Cactus/μp/C and Cactus/μp/J whenever the distinction is necessary.
- Samoa [17]. The Samoa framework is unique in its advanced support for handling concurrency in a way that is transparent to microprotocols.
- Eva (the version used in the Eden group communication toolkit, received 2004/04) [3]. This framework handles events and hierarchies of microprotocols in an elegant manner.
- Appia (full distribution, versions kernel-1.9-2 and protos-0.11-2) [11]. The Appia framework extensively uses object-oriented features, such as class hierarchies. It is distributed with a lot of protocols.
- Fortika (received 2005/11) [10, 14]. Fortika is a group communication toolkit whose microprotocols can be composed using various frameworks (currently Cactus/J, Appia, and Samoa). This is possible thanks to a set of conventions and interfaces to write framework independent code. It is interesting for us because the conventions it follows are a synthesis of the features of three frameworks. In the sequel, when we cite Fortika, we refer to these conventions.
- JGroup 2.0 [12]. The JGroup framework is included because it is most similar to generic Java component frameworks: components usually hold references to other components and use method calls to communicate.
- JGroups 2.2.7 [1]. This framework is included because it is a successor of the influential Horus and Ensemble frameworks, and because it is rather visible in the broader Java developer community. In this paper, JGroups appears by its former name, JavaGroups, to avoid confusion with JGroup.[2]

Other related work. In [14], the authors define four sets of criteria, called models, to compare the Appia and Cactus frameworks.

The concurrency model defines whether and how concurrency is allowed in the framework. The interface to the environment defines how the application interacts with the outside world (network, application, etc.). The present paper focuses on the concurrency model and some implications of the concurrency model to the interface to the environment, in much more detail and involving more frameworks than [14].

The other two models describe how microprotocols compose and interact. We only summarized this in Section 2; a detailed discussion is out of scope for this paper.

4. Concurrency models

As the consequences of multiple threads running concurrently are difficult to foresee, one needs to coordinate how threads access shared resources, such as the internal state of microprotocols, or the shared state of multiple microprotocols. Frameworks differ in how they solve this problem. Some allow only a single thread for running microprotocols, and others are more permissive. In this section, we describe the characteristics of each group of frameworks.

Multi-threaded model. A lot of frameworks (Neko, JavaGroups, Cactus/μp, Cactus/cp, Eva, and JGroup) permit the use of multiple threads. These frameworks provide special operations, microprotocol skeletons, or microprotocols that launch new threads, or allow the microprotocol programmer to use standard features of the programming language (`Thread` class in Java, POSIX threads in C, etc.) to launch new threads. These frameworks do not offer extensive support for restricting multi-threaded behavior; microprotocol programmers and composers have to manage concurrency on their own.

Cactus/μp is unique among the frameworks surveyed in that the microprotocol programmer must choose the concurrency of handling an event at the moment of sending that event: the event can be handled by the sending thread, by a new thread, or by a thread from a thread pool.

Cactus/μp/C provides mechanisms to guarantee that the execution of a handler is not interrupted by the execution of another handler in the same higher-level microprotocol, even in the presence of concurrency. In contrast, Cactus/μp/J does not restrict concurrency at all. We will return to Cactus/μp/C in Section 5.3.

2 JavaGroups provides both a gateway to the Ensemble toolkit and a Java re-implementation of Ensemble protocols. We analyze the latter.

Single-threaded model. Two frameworks, Appia and Fortika (when used with Appia), have a dedicated thread that executes microprotocols. No other thread is allowed to execute microprotocols. Hence no concurrency is possible inside the composition. The underlying philosophy is that multi-threading introduces a lot of complexity, and the possible gains in performance and readability are not worth this additional complexity.

The single-threaded model requires that microprotocols follow a set of strict rules:

no new thread launched Microprotocols cannot start new threads and owning private threads.

non-blocking handlers Handling an event should not take a long time, and should never block the dedicated thread waiting for some condition. Otherwise, the handling of subsequent events may be blocked indefinitely, and the whole composition may have problems of liveness.

no external interaction Microprotocols should not interact with the outside world. This includes using the network, or peripherals. The reason is that such interactions may block, or may call the code of the microprotocol asynchronously, thus introducing concurrency.

The rules can be summarized the following way: (1) reactive microprotocols only send events while handling an event; (2) handling an event always finishes within a short time. Microprotocols that follow these rules are called *reactive* microprotocols (they only react to their environment), and other microprotocols are called *active* microprotocols (e.g., they can launch new threads). The single-threaded model thus requires that all microprotocols be reactive. In contrast, both active and reactive microprotocols are allowed in the multi-threaded model.

Of course, frameworks that follow the single-threaded model also need functionality that involves external interactions. As reactive microprotocols cannot implement such functionality, it is implemented by code that is not part of microprotocols, and is therefore *outside* the composition. The thread that executes microprotocols may communicate with code outside the composition by queues, for instance.

Finally, note that using a dedicated thread to execute the composition is not the only possibility to implement the single-threaded model. Another possibility is embedding the composition in a monitor that guarantees that only one thread can access microprotocols at any given time (this thread is not necessarily the same thread all the time). Fortika (when used with Cactus) uses this solution.

Model with transparent concurrency. The Samoa framework is unique in its approach to concurrency issues. It features a scheduler capable of allowing multiple threads to execute handlers concurrently, but, unlike in the multi-threaded model, this concurrency is *transparent* to the microprotocols: all microprotocols are reactive, just like microprotocols in the single-threaded model.

We have just seen that frameworks implementing the single-threaded model do not allow concurrency inside the composition. Therefore, if the environment sends several events into the composition at the same time (e.g., several messages from the network), such frameworks handle these events one after the other. In contrast, Samoa's scheduler allows such events to progress into the composition unless they conflict (see [17] for the exact definition of conflicts). In the best case, the events produced by external interactions can be handled concurrently, and otherwise, some events are blocked while other events are handled. The scheduler enforces an isolation property similar to the isolation property of transactions in databases. The kind of scheduler used is called a pessimistic (i.e., rollback-free) scheduler in that context.

To support the detection of conflicts, the composer has to provide additional information about the microprotocols that may and may not be executed for each type of external interaction. Such information is not necessary in either the single- or multi-threaded model.

Overall, this new approach is promising, but there remain open questions. One is how much concurrency can be introduced into the composition. For example, given compositions where most of external interactions execute most of microprotocols, the scheduler will hardly allow more than one event to progress concurrently. Other compositions probably fare better. Further research, involving measurements, is needed to answer this question.

5. Comparing the concurrency models

In this section, we investigate the tradeoffs between single- and multi-threaded frameworks, and propose a combination. We then contrast this combination with the model with transparent concurrency, which can also be seen as a combination of the single- and multi-threaded models.

5.1. The single-threaded model is too restrictive

Multi-threading is useful in a variety of scenarios. In particular, multi-threading may simplify the protocol code; it is necessary for certain tasks, e.g., interfacing to system libraries that are multi-threaded; and it is useful to improve the performance of a service with slow operations. We first present examples for these scenarios. We then investigate how the scenarios are implemented in the single-threaded model, and highlight the drawbacks of these implementations.

Scenarios. There are numerous examples of multi-threading simplifying the protocol code; in fact, many pro-

tocols in the literature (see, e.g., [5]) are described as a set of interacting tasks, and the simplest implementation often uses a thread to implement each task.

There are activities that are only conveniently implemented with multi-threaded code. Protocols nearly always interact with the network, and networking libraries (such as the Java standard library) are often multi-threaded themselves. Scheduling activities for execution at some future time is another example.

Even tasks that do not involve external interaction may take a long time. An example is cryptographic computations in security protocols. An optimization that may improve the throughput of this task significantly is parallelizing the computations. Another, more common example is a web server. Web servers typically handle requests in a new thread because the pages should be available to requests arriving concurrently.

Implementing the scenarios. We have shown scenarios in which multi-threading is beneficial. As the single-threaded model does not allow multi-threading within the composition, part of the code for these scenarios must be implemented outside the composition, as some kind of components used by microprotocols. One problem is that one part of the code will be inside microprotocols, and another part in components that are *not* microprotocols. There is a danger that such components will offer a different interface than ordinary microprotocols, especially if the components provided by the framework are not enough, and the microprotocol programmer must implement ad-hoc components. Another problem is that such a decomposition exposes implementation details: if, for instance, the cryptographic computations mentioned above are not used by other protocols, there is no reason for exposing their interface to the framework. These problems ultimately limit the flexibility of frameworks that follow the single-threaded model, as microprotocols and components different from microprotocols cannot be interchanged easily, and there are interfaces that programmers must adhere to that are really only implementation details. In other words, the composer is exposed to concerns that should be concerns of the microprotocol programmer.

Conclusion. Our conclusion is that the single-threaded model limits microprotocols too much: programmers are forced to solve important problems or implement parts of solutions outside the composition. The reason is that *all* microprotocols are required to be reactive.

5.2. The multi-threaded model is too permissive

The multi-threaded model, in contrast to the single-threaded model, does not restrict the concurrency of microprotocols. This means that frameworks following this model provide few or no facilities for microprotocol programmers and composers to solve problems of concurrency, beyond the generic facilities offered by the programming language and its libraries.

Difficulties for microprotocol programmers. The problem here is that single-threaded programming is genuinely easier. The issue is not just the convenience of the microprotocol programmer: sometimes, simpler code can be written if the composition, or a part of the composition, is executed by a single thread. In the multi-threaded model, the programmer is confronted to non-determinism in the protocol execution. This non-determinism can even be a concern when performing usual tasks, such as preserving FIFO order of events from the sender to the handler microprotocol in a given composition. While such tasks are easy to implement in the absence of concurrency (the sender microprotocol can just send the events and the framework ensures that they are handled in FIFO order), they require complicated solutions if multiple threads are involved (the order of two events might be reversed, and thus the microprotocols need to be modified to keep FIFO order, e.g., by using sequence numbers).

In the subsequent sections, we will provide more examples of facilities that allow microprotocol programmers to write simpler code. These facilities have a small overhead, but only if some microprotocols are executed by a single thread.

Difficulties for composers. Concurrency issues, such as protecting the states of microprotocols against concurrent changes and avoiding deadlocks, cannot always be solved by the microprotocol programmers on their own: the composer must be involved. In order to do this, the composer needs to have deep knowledge of the composed microprotocols that includes details such as the cumulative state of the microprotocols to protect, or the handlers in which new threads are launched. Having to account for all these details greatly complicates the task of the composer. In contrast, the composer can ignore concurrency issues in the single-threaded model.

Conclusion. Our conclusion is that frameworks should provide facilities to make microprotocol programming and protocol composition easier, by restricting concurrency. Frameworks following the multi-threaded model provide few such facilities.

5.3. Islands of reactive microprotocols: the best of two worlds

One can combine the advantages of the single- and multi-threaded models in the following way: let the programmer and the composer define sets of microprotocols that will be executed by a single thread at a time. We call such sets *islands*. Within islands, microprotocol program-

Concurrency model	single-threaded	multi-threaded with reactive islands	multi-threaded
Microprotocols	reactive, simple code		reactive or active, complex code
Active code	outside the composition	inside the composition	
Composition	simple		complicated

Table 1. Comparing the single- and multi-threaded models of concurrency and their combination.

mers can write simpler code by taking advantage of the kind of guarantees offered in the single-threaded model: if all its microprotocols are reactive, the island itself is reactive as well. However, outside the islands, programmers are free to use multi-threading without restrictions, just as in the multi-threaded model, and thus implement any required functionality as microprotocols, within the composition. Table 1 summarizes the main characteristics of the single- and multi-threaded concurrency models (left and right columns, respectively) and points out how the multi-threaded model with reactive islands (center column) combines the advantages of both: microprotocol code and the task of composition is usually simpler, yet multi-threaded code can reside inside the composition.

Out of all the frameworks, only Cactus/μp/C follows a model similar to the multi-threaded model with reactive islands. This means that the framework allows active microprotocols, but its event scheduler provides guarantees for certain sets of microprotocols if they consist of reactive microprotocols only. The sets are the higher-level microprotocols of Cactus. The difference is that the boundaries of these sets cannot be chosen arbitrarily, as microprotocols and higher-level microprotocols work very differently in Cactus. Moreover, microprotocol programmers should not use certain features, e.g., sending events in a new thread, in reactive islands.

Reactive islands and their execution. We next elaborate on what we mean by reactive islands, and how they interface to other microprotocols.

The key property of islands is the following: if each of the microprotocols of an island is reactive, then the whole island behaves like a reactive microprotocol, as well, and we speak of a *reactive island*. This means that reactive islands never send events if they are not handling any event; and if handling an event starts, it always finishes, and does so within a short time (with suitable and straightforward definitions for sets of microprotocols sending and handling events). Note that the single-threaded model uses exactly one reactive island that includes all microprotocols, whereas the multi-threaded model uses no reactive islands, or at least, the framework is not aware of reactive islands.

There are at least two solutions for executing an island that consists of reactive protocols only, such that it will be reactive.

- One solution uses a dedicated thread. It is similar to how single-threaded frameworks handle events. There are two queues. Incoming events are put in the first queue, called inbound queue. A dedicated thread then repeats the following: it reads an event from the inbound queue, handles the event and any events sent to a microprotocol in the island while handling the event, in a recursive manner. Outgoing events are put in the second queue, called outbound queue, from which other threads will read the events and handle them.
- The other solution is that the island forms a monitor. Any thread can enter to handle an event, but in mutual exclusion with other threads. Inside the monitor, the thread should handle the event and any events sent to a microprotocol within the island, in a recursive manner, but not any outgoing event. It may handle outgoing events only after exiting the monitor.

Using islands of microprotocols. The question remains what parts of the protocol stack or distributed application can and should be implemented as an island of reactive microprotocols, and which ones should be active (i.e., multi-threaded).

Previous work with protocol composition (e.g. [9, 14]) reveals that, in usual compositions, most of processor time is spent on the serialization and deserialization of events for transmission over the network. Moreover, if the framework implements a single-threaded model, (de)serialization becomes a prominent performance bottleneck.

In multiprocessor platforms, serialization workload produced in a typical protocol composition is big enough to fully utilize all processors available at each node. Besides, parallelizing serialization is extremely easy since the (de)serialization of two different events is completely independent.

This leads us to the conclusion that most of the code of the distributed application can be implemented as a single reactive island of microprotocols, whereas serialization-related microprotocols should be executed by multiple threads so that the high workload they generate is distributed over all processors of the execution platform. Note that if the application has different performance characteristics (e.g., serialization is no more the bottleneck), it is often straightforward to shrink the reactive island's boundary to introduce concurrency, by adding active microprotocols or replacing some of the reactive microprotocols by active versions.

Concurrency model	single-threaded	transparent	multi-threaded
Microprotocols	reactive, simpler code		reactive or active, complex code
Active code	outside the composition	outside the composition but concurrency in the composition	inside the composition
Composition	simple	complicated	

Table 2. Comparing the single-threaded, multi-threaded and transparent models of concurrency.

5.4. Comparing with transparent concurrency

Just like the multi-threaded model with reactive islands, the model with transparent concurrency can be considered as a combination of the single- and multi-threaded concurrency models. Table 2—whose structure parallels that of Table 1—points out how the model with transparent concurrency (center column) combines aspects of the single- and multi-threaded concurrency models (left and right columns, respectively): concurrency is possible within the composition, just like in the multi-threaded model, and the price is that the composer's task becomes difficult, as the composer has to provide rather detailed information about possible executions. Other aspects are similar to the single-threaded model.

Let us now contrast the multi-threaded model with reactive islands and the model with transparent concurrency. With both models, (most) microprotocols are reactive. However, concurrency is introduced in a different way. The model with reactive islands allows active microprotocols, and the composer can compose these with reactive microprotocols. In contrast, the model with transparent concurrency puts all the burden of introducing concurrency on the composer, and the composer's task is much more difficult.

Note that the model with transparent concurrency still requires that external interactions are performed outside the composition. In other words, this model only aims at solving one problem that active code is used for: that of increasing performance on multiprocessor platforms. For this reason, we view the two approaches as complementary, and in fact, they could be combined. The combination is a multi-threaded model with reactive and active microprotocols, in which some reactive microprotocols form reactive islands as presented above, and some others form islands that are executed by a scheduler offering transparent concurrency. The latter islands are like the entire composition in the model with transparent concurrency.

6. Overlapping executions of handlers

Consider an execution that involves handlers h_1, h_2, and h_3. Handler h_1 sends an event that is handled by handler h_2, and h_2 sends another event handled by h_3. Finally, let h_1 and h_3 be handlers of the same microprotocol $\mu protocol_1$. Figure 1 depicts this execution.

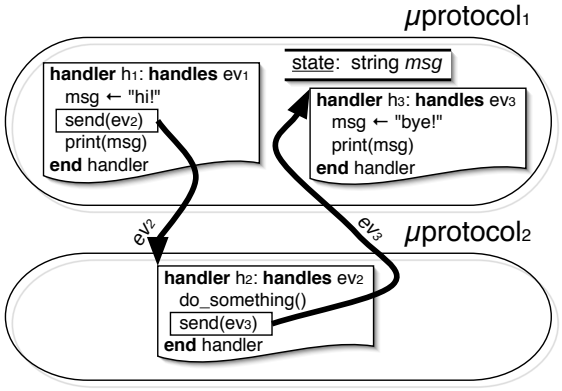


Figure 1. Example of possibly overlapping executions of handlers. What are the contents of msg when handler h_1 prints it?

Similar chains of events, ones that start and end at the same microprotocol, may give rise to consistency problems. In the example, h_1 and h_3 are part of the same microprotocol ($\mu protocol_1$), and thus they both modify the state of this microprotocol (the string msg). The consistency problem is that the output depends on how the statements of h_1 and h_3 are executed: if the first line of h_3 executes before the last line of h_1, h_1 will print the wrong message ("bye!").

Note that the root of the problem is that the executions of handlers h_1 and h_3 may overlap in time. Note also that the microprotocol programmer cannot use standard mechanisms (*synchronized* keyword in Java, *mutex* variables in C) to guard against the consistency problems. The reason is that such mechanisms only protect against modifications by *different* threads, and h_1 and h_3 might be executed by the *same* thread.

This section describes and discusses possible solutions to the problems caused by overlapping executions of handlers.

6.1. Anticipating consistency problems.

The first possible solution is requiring that the microprotocol programmer anticipates how consistency problems may occur: at least, how consistency problems may occur when a single thread executes several handlers of the microprotocol such that they overlap. In order to do this, the programmer would have to know the details of the composition (e.g., $\mu protocol_1$ and $\mu protocol_2$ in Fig. 1). This goes against the main property of microprotocol frameworks: microprotocols should be written in (relative) isolation and composed in a (relatively) unrestricted manner. Hence we do not discuss this solution any further.

6.2. Introducing concurrency.

The second possible solution is to avoid that the same thread executes handlers of the same microprotocol whose execution overlaps. This means that at least one of the handlers involved (e.g., h_2 in Fig. 1) should be executed by a different thread. There are different ways of achieving this, depending on who introduces concurrency:

- Both the microprotocol programmer and the composer might introduce concurrency in an ad-hoc manner. This is possible in all frameworks that follow the multithreaded model.
- The default composition might introduce concurrency. Neko uses this solution.
- The runtime system may use a different thread for executing each handler. Cactus/cp uses this solution.

6.3. Non-overlapping handler executions.

The third possible solution is to disallow overlapping handler execution, by requiring that a handler h finishes before the handling of any event sent by h. If overlapping handler executions are not allowed, the consistency problem described above cannot occur: the execution of handlers h_1 and h_3 in Fig. 1 will never overlap.

We next present two possible implementations of non-overlapping execution.

Scheduler with an event queue. Non-overlapping execution of handlers can be implemented by putting sent events into a queue, and handling the events in this queue only after the sending handler finishes. Appia and Cactus/μp/C use this implementation. The advantage of this implementation is that it is implemented in the runtime system; microprotocol programmers do not have to worry about ensuring non-overlapping executions.

Conventions for writing handlers. Non-overlapping execution of handlers can also be implemented by convention. Sending an event triggers handling the event immediately: the method implementing the handler is called directly from the sending handler. Normally, this would result in overlapping handler executions. This can be avoided if microprotocol programmers follow rules that ensure that the resulting execution is equivalent to a non-overlapping execution. Fortika microprotocols and most of the Neko microprotocols follow such rules.

We now present two examples of such rules. The first rule requires that sending events be the last actions of a handler: if events are generated before the end of the handler, they must be stored in local variables and sent at the end of the handler. The second rule states that handlers copy all the data necessary for their execution into local variables before any events are sent. If this rule is followed, the execution of a handler is not affected if any of the sent events causes the execution of a handler of the same microprotocol.

The example in Fig. 1 clearly does not follow either of these rules. To make the microprotocols follow the rules, one needs to move `send(`ev_2`)` in h_1 to the end of the handler.

6.4. Discussion

Introducing concurrency vs. *non-overlapping handler executions.* Of the two classes of solutions, non-overlapping executions and introducing concurrency, we argue that non-overlapping executions are preferable.

One reason is that we would like to provide reactive islands of microprotocols, and such islands are executed by a single thread at any time (see the extensive discussion of advantages and possible implementations in Section 5). The solution that introduces concurrency would limit the size of reactive islands, whereas non-overlapping executions do not have such an effect.

The other reason is performance: introducing concurrency is costly, as it involves launching and synchronizing threads. In contrast, non-overlapping executions do not require the use of multiple threads. Moreover, when introducing concurrency, the number of threads and/or the memory used for buffering events that wait for a handling thread becomes difficult to control.

Nevertheless, non-overlapping executions have a small impact on performance. They require that the execution of handlers or parts of handlers is delayed, as handling some events may only start once the sender handler finishes. However, the majority of handlers (those in reactive islands of microprotocols) will execute quickly in any case, and thus this delay will not be significant.[3]

Conventions for writing microprotocols vs. *scheduler with an event queue.* We next contrast the two implementations of non-overlapping handler executions: conventions for writing microprotocols, and the scheduler with an event queue.

Conventions for writing microprotocols restrict microprotocol programmers. However, the advantage is that a single Java method call (or C function call) implements sending and handling an event. This yields good performance, as synchronous calls are the primary means of interaction between different parts of the code in all popular programming languages and are thus significantly faster than any other communication mechanism. Moreover, method/function calls allow the compiler to check the types of the data transmitted in an event, whereas other communication mechanisms are not as type-safe.

To summarize, conventions for writing microprotocols offer better performance and type-safety, but they are slightly inconvenient for microprotocol programmers. Our conclusion is that none of the implementation is conclusively better than the other, and thus we consider that either implementation yields good microprotocol frameworks.

7. Ordering events

A microprotocol's effect on other microprotocols and the environment is not just determined by the set of outgoing events that the microprotocol sends. Often, the order in which the outgoing events are sent is important as well. An everyday example is requiring first-in-first-out (FIFO) guarantees from a communication channel, such as a TCP connection. Implementing protocols such as HTTP on top of TCP is much easier than it would be to implement it on top of a reliable datagram service without FIFO guarantees. Note that providing FIFO guarantees between the two communicating processes is not enough. FIFO ordering is needed within each process, between the code that implements HTTP and the code that implements TCP, as well.

3 Note also that the designers of Java were confronted to an analogous problem, and they also chose the solution that corresponds to non-overlapping executions. Consider a thread t_1 that calls the notify method on an object o, in order to wake up another thread t_2, blocked in a call to the wait method on o. Both the call to notify and the call to wait are in a synchronized block (synchronized on o). The analogy is the following. Both synchronized blocks in Java and handlers in our problem are blocks of code. The call to notify corresponds to sending an event in our problem, as both triggers the asynchronous execution of another block of code. The exact execution of blocks of code is out of the control of the programmers of the blocks of code, because it involves other blocks of code, people composing blocks of code, and the non-deterministic Java scheduler. For these reasons, the solution is also analogous: according to the Java specification, t_2 may resume executing only after t_1 has finished executing the synchronized block surrounding the call to notify.

Systems	message passing distributed systems	microprotocol frameworks
Comm. entities	processes	microprotocols
Unit of comm.	messages	events
Start of comm.	sending	sending
End of comm.	receiving	start of handling
Comm. steps	events	actions

Table 3. Conceptual mapping between communication in message passing distributed systems and microprotocol frameworks.

A variety of ordering guarantees have been introduced in the field of message passing distributed systems, starting with early papers [8]. Events are an asynchronous form of communication, just like messages in distributed systems, and microprotocols communicate by events and may be executed in parallel, by different threads; they are like processes in distributed systems that communicate by messages and execute in parallel. The two kinds of systems are contrasted in Table 3. For this reason, some of the ordering guarantees for distributed systems might be useful in microprotocol frameworks. This section investigates ordering guarantees and the feasibility of providing them in microprotocol frameworks.

7.1. Feasibility of ordering

A kind of ordering, already mentioned, is FIFO order. Informally, FIFO order means the following: if two events are sent from the same microprotocol to the same handling microprotocol, then the order of handling is the order of sending. We also consider extensions of FIFO order, such as causal order, well-known in distributed systems [2, 8], as well as an extension to causal order.

These kinds of ordering are motivated, defined and discussed later, in Section 7.2. We first investigate how microprotocol frameworks should support ordering events. Knowing only about FIFO order suffices for understanding this discussion.

Ordering events between all microprotocols. We argue that the framework should not guarantee order between events in the general case, if active microprotocols are involved. The reason is that the framework must observe the order of actions of microprotocols, i.e., the order in which microprotocols send and handle events, in order to provide ordering. Observing the order of these actions requires that the threads involved are synchronized frequently. However, a major motivation for using active microprotocols is to fully exploit multiprocessor systems, and this is only achieved if threads are synchronized infrequently. In other words, or-

dering events if active microprotocols are involved defeats the reason for using active microprotocols.

Note that ordering can be implemented in the general case if the need arises: the microprotocol programmer can implement ordering within microprotocols, or the composer can add microprotocols to the composition that help ordering events. We have only argued against the framework ordering *all* events. Microprotocols can implement ordering by using algorithms from distributed systems. For example, microprotocols can keep FIFO order by using sequence numbers in events, and handling incoming events in the order of their sequence numbers. Keeping other kinds of ordering requires more complex algorithms; e.g., see [2] for an algorithm that provides causal order.

Ordering events within a reactive island of microprotocols. We argue that the framework should offer a limited support for keeping order: events within a reactive island of microprotocols should be subject to ordering.

The reason is that keeping even the most complicated kind of ordering (discussed and motivated in Section 7.2) is cheap in this context: no thread synchronization is involved, as all events sent or handled in a reactive island are processed by a single thread at any time. We present algorithms for ordering within reactive islands in Section 7.3.

7.2. Definitions of ordering

In this section, we review the kinds of ordering we consider, provide formal definitions, and present related research results, also from other contexts.

We introduce the following notation: $\mathrm{send}(e)$ denotes the action of sending some event e, $\mathrm{handle}(e)$ the action of starting the handling of event e, and $\mathrm{handler}(e)$ the microprotocol that handles e.

We also introduce precedence relations on actions. Actions are either the sending or the handling of an event. The first precedence relation orders actions that take place on the same microprotocol.

Definition 1 (Local precedence) *Consider two actions A and B that take place on the same microprotocol. A locally precedes B ($A \rightarrow_l B$) if (1) A and B are executed by the same thread and A is executed first, or (2) A and B are ordered by a synchronization primitive of the programming language and A is executed first. Local precedence is the transitive closure of these relations.*

For example, A and B are ordered by a synchronization primitive of the programming language if A and B are part of synchronized blocks of a Java program that synchronizes on the same object, even if they are executed by different threads. Note that not all actions of a microprotocol are ordered by local precedence. For example, actions executed by different threads on a microprotocol that does not use synchronization facilities are not ordered.

The following order is defined in terms of local precedence:

Definition 2 (FIFO order) *Let e and e' be two events such that $\mathrm{handler}(e) = \mathrm{handler}(e')$. If $\mathrm{send}(e) \rightarrow_l \mathrm{send}(e')$ then $\mathrm{handle}(e) \rightarrow_l \mathrm{handle}(e')$.*

Informally, FIFO order means the following: if two events are sent by the same microprotocol to the same microprotocol, then the order of handling is the order of sending.

Other kinds of ordering involve a different kind of precedence.

Definition 3 (Causal precedence) *Consider two actions A and B. A causally precedes B ($A \rightarrow B$) if $A = \mathrm{send}(e)$ and $B = \mathrm{handle}(e)$ for the same event e. Causal precedence is the transitive closure of this relation and local precedence.*

Causal precedence is used to define the following order:

Definition 4 (Causal order) *Let e and e' be two events such that $\mathrm{handler}(e) = \mathrm{handler}(e')$. If $\mathrm{send}(e) \rightarrow \mathrm{send}(e')$ then $\mathrm{handle}(e) \rightarrow_l \mathrm{handle}(e')$.*

Causal order is a generalization of FIFO order. FIFO order concerns sending events from a *single* microprotocol; in contrast, causal order concerns sending events from a computation that spans causally ordered actions on *multiple* microprotocols. It ensures that if events are sent by such a computation, then the order of sending is the order of handling.

We now propose a different extension to FIFO order. This extension reflects that communication may involve multiple events. In contrast, causal order reflects that computations may involve multiple microprotocols. The extension implies causal order, not just FIFO order.

Definition 5 (Extended causal order) *Let e_s, e'_s, e_h and e'_h be events such that $\mathrm{send}(e_s) \rightarrow \mathrm{handle}(e_h)$, $\mathrm{send}(e'_s) \rightarrow \mathrm{handle}(e'_h)$, $\mathrm{send}(e'_s) \not\rightarrow \mathrm{handle}(e_h)$, and $\mathrm{handler}(e_h) = \mathrm{handler}(e'_h)$. If $\mathrm{send}(e_s) \rightarrow_l \mathrm{send}(e'_s)$ then $\mathrm{handle}(e_h) \rightarrow_l \mathrm{handle}(e'_h)$.*

Causal order involves sending events, and handling the same events. In contrast, extended causal order involves the sending of events that causally precede the handling of some other events.

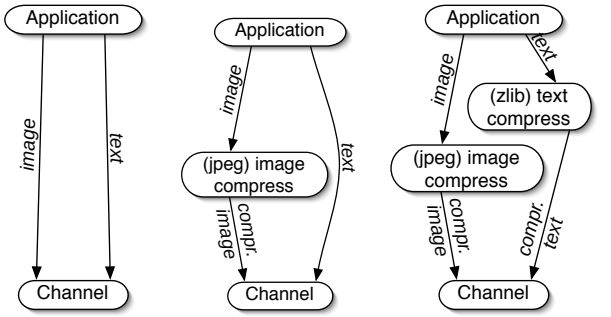
Example to illustrate extended causal order. We next present an example that highlights why extended causal order is useful in microprotocol frameworks.

Consider the two microprotocols illustrated in Fig. 2(a). The application microprotocol sends a message with both image data and text. The order is important: the image is

sent before the text, hence a FIFO communication channel, implemented by the other microprotocol, is used. The communication between the two microprotocols must be FIFO.

Now consider a variant of this scenario, shown in Fig. 2(b). Again, the application microprotocol sends the image event before the text event. The difference is that the image data is now compressed by a third microprotocol. In this variant, one needs causal order. With FIFO order, it is possible that the communication channel handles the compressed image *after* the text. If this happens, the message would be garbled.

Finally, consider another variant of the scenario, shown in Fig. 2(c). The difference is that the text is compressed by a fourth microprotocol (the third microprotocol cannot be reused, as image data and text are usually compressed with different algorithms). In this variant, FIFO and causal order cannot guarantee that the communication channel handles the compressed image first and the compressed text second. One needs extended causal order to avoid garbled messages.



(a) no compression (b) image compressed (c) both image and text compressed

Figure 2. Example to illustrate FIFO, causal and extended causal order.

Related work. Causal precedence corresponds to Lamport's precedence relation in message passing distributed systems [8]. The difference is that in [8], all events on a process are ordered, whereas our local precedence is a partial order.

Local and causal precedence are similar to relations introduced in the context of distributed object environments [4]. The main difference is that our model is simpler: besides asynchronous communication, [4] considers synchronous communication and read/write operations on shared variables, as well.

Our definition for causal order is analogous to causal order in message passing distributed systems [2] and distributed object environments [4].

[18] defines extended causal order. Our definition is a greatly simplified version of that definition.

Real-time order. Besides the order of events, the time of sending and handling events may also be important, e.g., for real-time applications. We do not expect that the majority of applications of a protocol framework would be real-time, hence we do not think that the cost of automatic timestamping can be justified. Moreover, timestamping is a relatively expensive operation, as it often involves a system call.

Nevertheless, if microprotocols need timestamps they can be provided easily. Either microprotocols can explicitly put a timestamp into the events they send, or the composer can add timestamping microprotocols to the composition.[4]

7.3. Implementations of ordering

The conclusion of Section 7.1 was that microprotocol frameworks should offer ordering within reactive islands of microprotocols, and Section 7.2 defined useful kinds of ordering. This section discusses how the most general kind of ordering, extended causal order, could be implemented, and how (and whether) it is implemented in existing frameworks.

In Section 6.3, we presented two scheduler implementations: one with direct method calls and another with event queues. The implementation with direct method calls keeps extended causal order. The implementation with event queues also keeps extended causal order if all newly sent events are inserted at the front of the event queue in the order in which they were sent.

All frameworks but Appia and Cactus/μp/C use a scheduler with method calls and thus keep extended causal order, in executions that involve a single thread.

Cactus/μp/C uses event queues. Recall that Cactus has one kind of send operation that has the event handled in the sending thread, and other kinds that have the event handled in a different thread. If all send operations are of the first kind (i.e., the execution involves a single thread), Cactus/μp/C provides extended causal order. Otherwise, it provides only casual order. The reason is that newly sent events are inserted at the end (rather than the front) of the event queue. No guarantees are provided beyond the boundary of the higher-level microprotocol, though.

Appia, the other framework that sends event queues, does not keep causal order or extended causal order within reactive islands.[5] We next explain how their scheduler works. Appia organizes microprotocols in stacks, and

4 Real-time scheduling of handler executions, if needed, is a more complex problem.

5 To be more precise: Appia uses *one* reactive island that includes all microprotocols.

events are associated with a direction: up or down, depending on the position of the handler microprotocol in the stack with respect to the position of the sender microprotocol. When an event going up is handled, events sent by the handler are placed at the front of the event queue if they go up, and at the end of the event queue if they go down. This breaks causal order for the scenario shown in Fig. 2(b) if the scenario is triggered by an event going down (not shown in the figure), and the image compress microprotocol is on the top, the application microprotocol in the middle and the channel microprotocol on the bottom.

It is unclear why the Appia team made this design decision, and changing Appia so that it keeps causal and extended causal order would be rather easy. The same goes for Cactus/μp/C keeping extended causal order. In any case, the example of Appia and Cactus/μp/C illustrates that the idea that frameworks should provide extended causal order is non-trivial.

8. Conclusion

In this paper, we focused on microprotocol frameworks, frameworks that are specialized for implementing protocols and more complex distributed applications. Multi-threading is often useful in this context. We investigated what support for multi-threaded programming such frameworks provided and should provide for programmers.

Along with a survey and detailed discussions of the features of existing frameworks, we proposed a set of features that can be offered without significant changes in programs, and that have a negligible performance hit. This includes the following: (1) islands of microprotocols with reactive behavior that can coexist with active microprotocols, thus taking the best of two worlds; (2) non-overlapping execution of microprotocols involved in a chain of events, to avoid inconsistencies; and (3) providing ordering guarantees for events sent among microprotocols, including first-in-first-out (FIFO), causal order, and extended causal order. To the best of our knowledge, our definition of extended causal order is the simplest so far.

A preliminary implementation of these ideas has been carried out in the Neko microprotocol framework [16], with promising results.

Acknowledgments

We would like to thank Olivier Rütti, Rick Schlichting, Matthias Wiesmann and Paweł Wojciechowski for their insightful comments about protocol composition during various discussions.

This research is supported by the Japan Society for the Promotion of Science, a Grant-in-Aid for JSPS Fellows, the program "Fostering Talent in Emergent Research Fields" in Special Coordination Fund for Promoting Science and Technology by the Japanese Ministry of Education, Culture, Sports, Science and Technology, and the Swiss National Science Foundation.

References

[1] B. Ban. Design and implementation of a reliable group communication toolkit for Java. http://www.cs.cornell.edu/home/bba/Coots.ps.gz, http://www.jgroups.org, Sept. 1998.

[2] K. P. Birman and T. A. Joseph. Reliable communication in presence of failures. *ACM Trans. on Computer Systems*, 5(1):47–76, Feb. 1987.

[3] F. Brasileiro, F. Greve, F. Tronel, M. Hurfin, and J.-P. Le Narzul. Eva: An event-based framework for developing specialized communication protocols. In *Proc. IEEE Int'l Symp. on Network Computing and Applications (NCA'01)*, pages 108–119, Cambridge, MA, USA, Oct. 2001.

[4] L. Duchien, G. Florin, and L. Seinturier. Partial order relations in distributed object environments. *SIGOPS Oper. Syst. Rev.*, 34(4):56–75, 2000.

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

[6] M. A. Hiltunen and R. D. Schlichting. The Cactus approach to building configurable middleware services. In *Proc. Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany, Oct. 2000.

[7] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Software Engineering*, 17(1):64–76, Jan. 1991.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[9] S. Mena, C. Basile, Z. Kalbarczyk, A. Schiper, and R. Iyer. Assessing the crash-failure assumption of group communication protocols. In *Proc. 16th IEEE Int'l Symp. On Software Reliability Engineering (ISSRE)*, Chicago, IL, USA, Nov. 2005.

[10] S. Mena, A. Schiper, and P. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Int'l Middleware Conference*, pages 414–432. Springer, June 2003.

[11] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 707–710, Phoenix, AZ, USA, Apr. 2001.

[12] A. Montresor, R. Davoli, and Ö. Babaoğlu. Middleware for dependable network services in partitionable distributed systems. *Operating Systems Review*, 35(1):73–84, Jan. 2001.

[13] R. D. Schlichting and M. A. Hiltunen. The Cactus project, 1999.

[14] Sergio Mena, Xavier Cuvellier, Christophe Grégoire, and André Schiper. Appia *vs.* Cactus: Comparing protocol composition frameworks. In *22nd Symposium on Reliable Distributed Systems. Florence, Italy*, Oct. 2003.

[15] P. Urbán. *The Neko project*. École Polytechnique Fédérale de Lausanne, Switzerland, 2000.

[16] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, Nov. 2002.

[17] P. Wojciechowski, O. Rütti, and A. Schiper. Samoa: A framework for a synchronisation-augmented microprotocol approach. In *Proc. of IPDPS '04 (18th International Parallel and Distributed Processing Symposium)*, Apr. 2004.

[18] T. Yoshida. Message ordering based on the strength of causal relation. In *15th Int'l Conf. on Information Networking (ICOIN)*, pages 915–920, Beppu, Japan, Feb. 2001.