Appia *vs.* Cactus: Comparing Protocol Composition Frameworks*

Sergio Mena† sergio.mena@epfl.ch

Xavier Cuvellier‡ xcuvelli@info.fundp.ac.be

Christophe Grégoire‡ cgregoir@info.fundp.ac.be

André Schiper† andre.schiper@epfl.ch

† Ecole Polytechnique Fédérale de Lausanne (EPFL) 1015 Lausanne, Switzerland

‡ Computer Science Department Univ. of Namur 5000 Namur, Belgium

Abstract

The paper presents and compares Appia and Cactus, two frameworks for protocol composition. The comparison is based on the experience gained in implementing a fault-tolerant Atomic Broadcast service. The paper also gives preliminary performance results, and concludes with a discussion of the most interesting features of the two frameworks, and suggestions for an improved framework.

1. Introduction

A protocol composition framework is the infrastructure that allows a programmer to build a complex protocol (or service) out of a set of off-the-shelf building blocks. For the external user, the composition is perceived as a whole (large) middleware providing the expected (complex) service. In the same way as the goal of a middleware is to make the development of distributed applications easier, the goal of a protocol composition framework is to make the development of a complex middleware easier. As distributed applications become more and more complex, they require stronger guarantees from the underlying middleware.

Group communication middlewares are not an exception. At the beginning, group communication middlewares like ISIS [5, 4], Transis [7], Phoenix [15], etc. were monolithic: their protocols were tightly coupled and had plenty of hidden dependencies, even though they were based on well defined building blocks. The fact that they were monolithic did not allow these building blocks from being reused elsewhere. Later on, group communication middlewares started to be modular: Consul [19], and Horus [24, 21] pioneered modular group communication. Finally, state-of-the-art group communication toolkits like Ensemble [10], JavaGroups [1], Appia's group communication toolkit [20], or the group membership service described in [12] are all modular: they are built from smaller off-the-shelf building blocks called *protocol modules*. Protocol composition frameworks yield a number of advantages over monolithic approaches, namely configurability, reusability, extensibility and ease of maintenance (see [2], Section 3.1 for a detailed discussion). However, they may yield plenty of drawbacks if they are not properly used (e.g., bad protocol module design, hidden dependencies left over, etc.), and might render a complex middleware protocol even harder to maintain than in the case of monolithic approaches.

The protocol composition frameworks for the development of modular group communication middlewares that we are aware of can be classified into two families. The first family consists of the x-kernel [13], and its successors Coyote and Cactus [2, 3, 12]. The second family, is composed of Horus and its successors Ensemble, Appia [17, 18, 16] and JavaGroups. The most representative frameworks in each of these two families are Cactus and Appia.

Cactus is an evolution of the x-kernel composition framework. It inherits and extends the x-kernel composition and concurrency model. The composition can be done at a finer granularity than in x-kernel, though it is also possible to compose legacy x-kernel protocols with newer Cactus-based protocols.

Appia is a re-engineering of Ensemble, which is itself a reimplementation of Horus using a dialect of the ML programming language. Appia inherits all features present in Ensemble, but its composition model has been extended and is more flexible. In Appia, as in Ensemble, protocol modules are composed on top of each other to form a stack. The main difference is the possibility, in Appia, to have more than one protocol module at the same level in the stack.

The Appia project has started after Cactus, but is already mature enough to be tested and compared with Cactus. In this paper we analyse and compare these two frameworks. We base our comparison on the experience gained while implementing a prototype group communication middleware (see Sect. 3) using both frameworks. The outcome of our comparison is that each framework has some unique interesting features, but none of them is clearly better than the other.

The rest of the paper is structured as follows. Section 2 gives an overview of Appia and Cactus. In Section 3 we de-

*Research funded by the EPFL grant "Semantics-Guided Design and Implementation of Group Communication Middleware" and by the Swiss National Science Foundation under grant number 21-67715.02.

scribe the prototype group communication service that has been implemented both in Appia and Cactus. In Section 4, we compare these two frameworks. Section 5 contains suggestions for better frameworks. Finally, Section 6 concludes the paper.

2. The Appia and Cactus protocol composition frameworks

In this section we present Appia and Cactus. Both frameworks provide mechanisms to solve problems that arise when composing protocol modules. We classify these mechanisms into four models.

Composition model. Specifies how protocol modules are arranged when they are composed. It can be hierarchical (protocol modules form stacks) or cooperative (no hierarchy). More complex models (e.g., a hybrid approach) are also possible.

Interaction model. Defines the way protocol modules may interact and exchange information. It may be event-driven or it may additionally allow data sharing. If it is event-driven, several *event classes* (or simply events) must be defined. Once defined, an event may be triggered at a certain moment. When this happens, a new instance of that event, called *occurrence*, is created. An event class specifies the type of information carried by all its occurrences.

Concurrency model. Describes whether and how concurrency is allowed in the framework. If some concurrency is allowed, the model should also specify how to synchronise concurrent threads. An interesting property that the concurrency model may provide is the so-called *FIFO order* of events: if two event occurrences are received by two protocol modules, then both will receive them in the same order.

Interface with the environment. Describes how the composition interacts with the outside world, i.e., application, network and system resources (such as timers, etc.).

In the remaining of this paper, we use these four models to describe and compare Appia and Cactus.

2.1. Appia

Composition Model. In Appia, a protocol module is defined as a pair *layer-session*. The layer defines three sets of events, namely, events *accepted* by the protocol module, events *provided*, and events *required* for proper operation. The session contains a private state and the protocol code (structured as a set of event handlers).

At composition time, a *Quality of Service* (QoS) is defined as a sequence of layer instances (see dashed squares in Fig. 1(a)), and the framework carries out a correction check: a QoS is rejected if some layer instance declares an event as required, but no layer declares it as provided. Once a QoS has passed the correction check, it is used for session instantiation: for each layer instance in the QoS, the corresponding session is instantiated. This yields a sequence of session instances called *channel*. Finally, an Appia stack contains one or several of these channels. Two different channels may share a session at some level (see for instance sessions p and s in Fig. 1(a)).

Interaction Model. The interaction model among protocol sessions is event-driven. Events are triggered by instantiating the event class, and providing three parameters: the channel, the *source* session (i.e., the session that is triggering the event), and a *direction* (either upwards on downwards). These three parameters define the route of the event (i.e., the sequence of sessions). If a protocol layer did not declare a given event class as "accepted", its companion session will not be put into the event's route. Thus, all occurrences of that event will bypass this session. A session forwards an event occurrence to the next session in the event's route by calling the event's method *go()*. Event occurrences convey data inside. Data sharing among sessions is not possible.

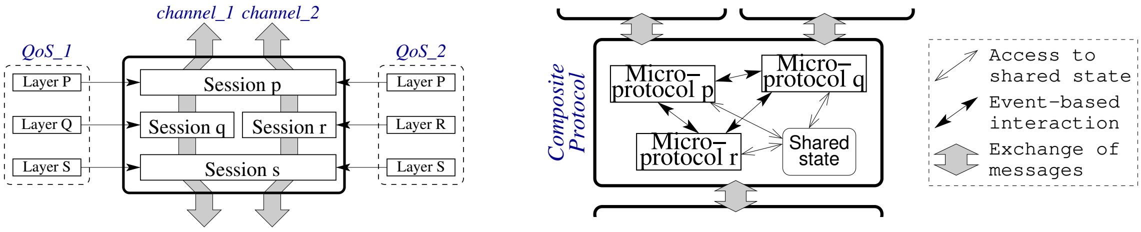
The framework has adopted a so-called *open event model* [16], which means that new event classes may be defined, in contrast to the *closed event model* of Ensemble. New event classes are defined by inheritance: the root event is class *Event*, and any other event class inherits from it or from one of its descendants. The advantage of the open event model is that legacy protocol modules can deal with new event classes, regarding them as an ancestor class (inheritance-controlled polymorphism).

Event occurrences are unable to get out of the channel they are in: when they end their channel route, they are destroyed. There is one exception: *SendableEvent* (and its descendants). Sendable events contain a *message*: at the bottom-most protocol module, the message is marshalled and transmitted to its destination; at the top-most protocol module, the extraction of the message is treated in an ad-hoc way. Messages have a simple interface consisting of *pushing* and *popping* headers.

Concurrency model. All event occurrences in all channels in the stack are processed by one single thread: the event scheduler thread. The main advantage of this single-threaded model is the absence of racing conditions inside sessions' code, i.e., protocol developers never have to worry about thread synchronisation. All event occurrences are put into the event scheduler's queue. In each step, the event scheduler pops the first event occurrence e from the event queue, looks up the next session that handles e, and executes the corresponding event handler. If the session creates a new event occurrence e' in the same direction as e, e' is inserted in the event queue immediately after e.

Interface with the environment. The outside world can insert event occurrences into a channel to notify about external happenings (e.g., a packet has come from the network, the application has sent a message). This is the only way for the application and the network to interact with an Appia stack.

As a minor interfacing detail, Appia defines a special event class called *TimerEvent* that sessions can use to set up timers.



(a) Appia example stack

(b) Cactus example composite protocol

Figure 1. Composition in Appia and Cactus. (a) Appia stack with two channels. Both channels share sessions p and s. (b) Composite protocol in Cactus

2.2. Cactus

Composition Model. Cactus defines a *two-level* composition model: coarse grain and fine grain. The coarse grain level is inherited from its ancestor, the x-kernel. The coarse grain protocols, called *composite protocols*, are composed by defining a hierarchical graph. In this graph, several composite protocols may be placed at the same level.

The composite protocols enclose a set of finer grain *micro-protocols* arranged in a cooperative way (no hierarchy) and interacting via event triggering and data sharing. Micro-protocols cannot exist on their own in the hierarchy graph: they need to be within a composite protocol.

Interaction Model: Composite Protocols. Composite protocols are linked by edges that define an acyclic oriented graph. Only composite protocols that are linked by an edge in this graph can directly interact. Composite protocols are instantiated at run-time yielding *sessions*. Each composite protocol session maintains its own state and its own micro-protocols. A session can *open* a new session of a composite protocol to which it is linked. The communication between two sessions is by *message* passing. In Cactus, a message is a data structure consisting of a set of *attributes*. Attributes are tuples of the form *(tag, scope, value)*. *Tags* are used to retrieve the attributes' *values*, which are the actual data. The *scope* restrains the attribute's visibility. A session may send a message up or down to the next session in the graph. When sending a message up, if there are several sessions that may receive the message, a *demux* function must be provided in order to decide which is the receiving session (this decision is usually based on some attribute conveyed in the message).

Interaction Model: Micro-Protocols. Micro-protocol execution is event-driven: micro-protocols are structured as a set of event handlers and contain private state. Event handlers' code, may modify this state and trigger other events. Micro-protocols can also share state (see Figure 1(b)).

At start-up time, a micro-protocol *binds* its handlers to events. So, when an event occurrence is triggered in some composite protocol C, the framework will execute all handlers in all micro-protocols of C that are bound to this event class. A priority is chosen at binding time to set the execution order of all handlers bound to the same event within a composite protocol. Event occurrences do not cross session boundaries: they are only seen by micro-protocols within the session in which the event was triggered. An important feature of Cactus is the ability to bind/unbind event handlers to events on-the-fly, which may change the behaviour of a micro-protocol at run-time. Moreover, a micro-protocol may be activated or disabled on-the-fly (by binding/unbinding all its handlers).

Concurrency model. A composite protocol, i.e., all the micro-protocols it contains, are idle as long as no message arrives from above/below. Message arrivals start out the activity in a composite protocol. At that point, the framework triggers an occurrence of the predefined event called *"message has arrived"*: all handlers of micro-protocols bound to this event will be executed one after the other.

The handler code may read/modify the private state in the micro-protocol as well as the shared state in the composite protocol, and may trigger other events. There are two ways to trigger an event: *raising* and *invoking*. *Raising* is asynchronous (a thread is spawn or reused from a thread pool), while *invoking* is synchronous (function call semantics). In the *raising* scheme, the private state of a micro-protocol may be exposed to racing conditions. So, it is necessary to synchronise the access to this private state (as well as to the micro-protocols' shared state). Such a synchronisation is not enforced by Cactus; it is the responsibility of the protocol programmer to enforce synchronisation using standard mechanisms (mutex, monitors, semaphores). Yet, some versions of Cactus (e.g., C versions) enforce, by default, some basic synchronisation properties such as atomic execution of handlers.

Cactus defines a special way to trigger events: *coordinated raising*. This is done using the *register* and *signal* methods. When several micro-protocols mp execute *register(mp,e)*, where e is an event class, then the event occurrence will only be triggered once *all* micro-protocols mp have executed *signal(mp,e)*.

Interface with the environment. There are no restrictions for micro-protocols to use OS resources as timers, sockets, etc. For the case of timers there is the possibility of raising an event after some delay. As for the network, the micro-protocols at the lowest level may open sockets themselves or can be placed on top of an x-kernel transport stack. The application, which is usually placed on top of the hierarchy graph, has no special interface: it has to use an interface as though it was just another composite protocol.

3. Service implemented

3.1. Description

In order to compare Appia and Cactus, we have implemented the same service in both frameworks. This service provides Fault-tolerant Atomic Broadcast,[1] and is implemented by the composition of several protocol modules (see Fig. 2, where arrows correspond to events):

- *Reliable Channel*. This protocol module provides reliable message communication between two processes located on different machines. The interface consists of the events *send* and *receive*, depicted in Figure 2. The protocol module also receives information about the group membership change (event *join_remove_list* in Fig. 2). The current implementation is based on TCP.[2]
- *Failure Detector*. This protocol module implements a failure detector based on ping messages.
- *Consensus*. This protocol module implements an algorithm for solving consensus [6]. It interacts with the Failure Detector module by means of *start_stop_monitor* and *suspect* events (see Fig. 2). It also interacts with the Reliable Channel module.
- *Atomic Broadcast*. This module implements Atomic Broadcast [6]. It interacts with the Consensus module (via the *run* and *decide* events) and the Reliable Channel module.

3.2. Implementation decisions

While carrying out the implementation in both frameworks, we have made a number of decisions. The most relevant ones are described below.

3.2.1 Common protocol code for both frameworks

All the protocol modules of Figure 2 have an internal private state and consist of a set of event handlers that may read and/or modify the private state and trigger events. Each of these protocol modules has been implemented as a Java class in which the event handlers are public methods and the state are private attributes. These *protocol classes* are instantiated in both composition frameworks without changing a single line of code (see circles in Fig. 3). As a result, the actual implementation of each protocol module is exactly the same in both frameworks.

[1] *Atomic Broadcast* ensures that all messages are delivered in the same order at all destination processes [9].

[2] We use Robust TCP connections [8], which properly mask link failures.

3.2.2 Event routing using framework facilities

We composed these protocol objects (i.e., these instances of protocol classes) using Cactus' and Appia's infrastructure:

- **Cactus:** As we can see in Figure 2, almost every protocol interacts with every other protocol. In Cactus event occurrences are unable to cross composite protocol boundaries. For this reason, we have used in Cactus one single composite protocol for the whole service (see Fig. 3). Every protocol object has been wrapped into a micro-protocol, which binds handlers in order to capture events from the environment and routes them to the appropriate handler. The wrapping micro-protocol also intercepts events triggered within the protocol object, and translates them into a Cactus event invocation.
- **Appia:** In Appia, every protocol object has been wrapped within a protocol session. The companion layer declares all events that the protocol object provides, requires and accepts. The session code has the same role as micro-protocols in the Cactus version: event handling only consists of (1) calling the appropriate handler inside the protocol object, and (2) intercepting all events triggered inside the common protocol code and translating them into Appia events, which are routed along the channel by calling method *go()*.

3.2.3 Concurrency model

We have chosen a similar concurrency model in both frameworks. In Appia there was only one choice, since the event scheduler is single-threaded. In Cactus, we have protected the composite protocol with a *mutex* to avoid concurrent execution. Moreover, we use (synchronous) event invocation to prevent spawning of new threads inside the composite protocol. This choice avoids to some extent the cost of context switches, and allows a fair performance comparison between Cactus and Appia.

4. Comparison

This section is devoted to a comparison between the two frameworks used to implement the service described in Section 3. Even though the models adopted by Appia and Cactus are very different, they share some common features. These features are pointed out in Section 4.1. Then, we discuss all the issues that make these two frameworks so different (Sect. 4.2). Finally, a performance comparison concludes this section.

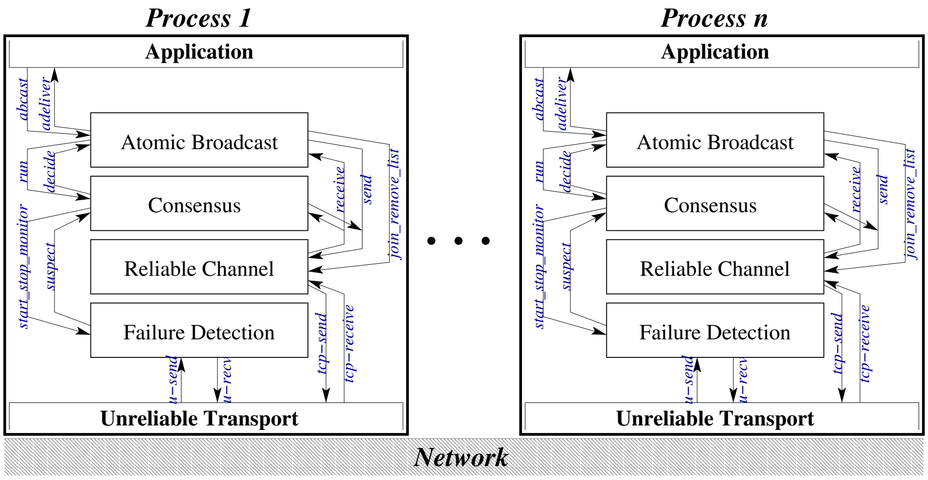


Figure 2. Protocol modules for Atomic Broadcast.

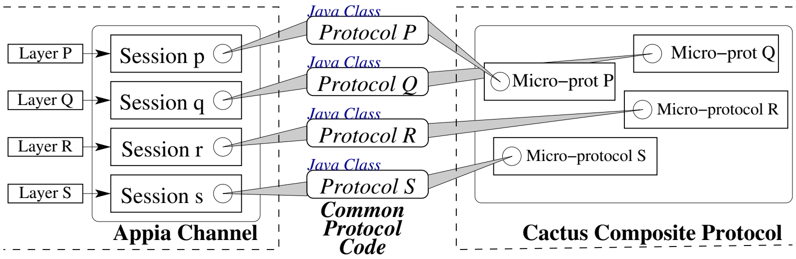


Figure 3. Protocol code shared by Cactus and Appia.

4.1. Similarities

4.1.1 Common features

Internal structure of protocol modules. The internal structure of Cactus' micro-protocols and Appia's sessions is the same: a set of event handlers (with an internal state), which react to occurrences of events. These event handlers play three main roles: (1) managing messages, (2) modifying the internal state, and (3) triggering events that may cause handler execution in other protocol modules. The fact that Appia and Cactus implement the same internal structure for protocol modules is the most important common feature for us, since without this resemblance it would have been impossible to share the protocol code in the implementation of our *Atomic Broadcast* service.

Message abstraction. Both frameworks have the concept of message. Messages are the only information able to reach the network. Although message structure differs, the concept of *headers* is present in both frameworks.[3] If a given protocol module pushes a header into a message, only its corresponding modules at the other endpoints are able to access and strip off this header. As a message flows down to the network, protocol modules see upper modules' headers as application data. Likewise, as a message flows upwards, each protocol module strips its header off the message. Our common protocol code takes profit from this similarity between both frameworks, treating messages in an homogeneous way.

[3] Cactus does not define headers, but attributes with *peer* scope behave exactly as headers.

4.1.2 Common lacks

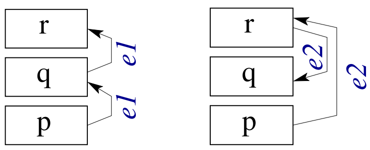
Flow control. Neither Appia nor Cactus have any form of flow control that would bound the workload (events in Appia, messages in Cactus) produced by the environment (network, application). In Appia, the absence of flow control may lead the event queue to overflow if flooded with incoming events (see Sect. 4.3). However, there is an ongoing effort to incorporate flow control to Appia.

In Cactus, the problem is prevented in our implementation thanks to the *mutex* that protects execution in the composite protocol.

4.2. Differences

4.2.1 Composition model

Appia's channels *vs.* Cactus cooperative composition. Protocol composition is quite different in each of the frameworks. In Appia, the concept of channel yields a strictly hierarchical composition, even though channels are allowed to share sessions: events flow up and down following a dedicated channel. This restricts in Appia the possible interaction patterns. Consider for instance Figure 4(a), where event $e1$ goes from protocol module p to protocol module q, and then to r. Appia's channels are optimised for this case. Figure 4(b) shows a different itinerary for event e_2: first from p to r and then from r to q. This is much trickier to express in Appia, since this order clashes with the order defined by the channel.[4] Appia has a workaround consisting of using a special event called *EchoEvent*, which conveys an event and travels up to the top (or down to the bottom) of the channel. At the end of the channel the conveyed event is extracted and reinjected into the same channel, but in the opposite direction. In Figure 4(b), protocol p would put an event $e2$ inside an *EchoEvent*, and send the latter up.



(a) (b)

Figure 4. Two different patterns of event flow. Appia channels only support the event path depicted in (a). Figure (b) is only possible in Appia with the help of *EchoEvent*.

This problem does not exist in the non-hierarchical composition of Cactus, where any micro-protocol can send an event directly to any other micro-protocol. Handler priority in Cactus allows to express either the event flow in Figure 4(a) or in Figure 4(b).

Correctness check. Cactus does not offer any correctness check. Appia does a composition correctness check at QoS creation time (see Sect. 2.1, Composition model). This correctness check rejects some incorrect compositions, but does not aim at being complete: an incorrect composition may pass this check.

4.2.2 Interaction model

Multiplexing occurrences. Some compositions may have complex event handling constraints. Two different protocol modules may accept the same event class, but each occurrence of this event is for only one of the two modules. Consider for example Figure 2, where *Reliable Channel* triggers *receive* occurrences. Every *receive* occurrence is handled either by *Consensus* or by *Atomic Broadcast*, but never by both. The *receive* event must be handled by *Consensus* if the event is the result of a *send* event triggered by *Consensus* in the corresponding remote endpoint; the *receive* event must be handled by *Atomic Broadcast* if the event is the result of a *send* event triggered by *Atomic Broadcast* in the corresponding remote endpoint. In Appia, these two occurrences travel through the same channel,[5] i.e., there is no way to directly reach *Atomic Broadcast*. In other words, all up-going *receive* events will first be handled by *Consensus*, even if they are not destined to it.

In Cactus, micro-protocols can directly interact, but the problem is similar. Both *Atomic Broadcast* and *Consensus* bind a handler to event *receive*. Therefore, upon triggering of an occurrence of this event (by *Reliable Channel*) both micro-protocols will handle the event, which is incorrect. We describe a solution to this problem in Section 5.2.

As a conclusion, any interaction based on Appia's notion of channel can straightforwardly be implemented in Cactus: the micro-protocol composition in Cactus is a generalisation of Appia's channel-based composition.

Shared data. In Appia, protocol interaction is only possible by means of events. Cactus offers data sharing among micro-protocols, in addition to event triggering. However, in order to reuse the same Java protocol classes in Appia and in Cactus, data sharing was not used in our Cactus implementation of the *Atomic Broadcast* service. In this specific example, we do not think that data sharing would have been very useful.

FIFO guarantee. The *adeliver* events triggered by the *Atomic Broadcast* protocol module in Figure 2 have to be handled by the application (or by any other protocol module that may be composed on top of *Atomic Broadcast*) in FIFO order, in order to preserve the total order provided by the service. This is ensured by Appia, which enforces FIFO order among event occurrences (see Sect. 2.1, Concurrency model). Cactus does not provide such a property. Nevertheless, in our implementation, the absence of concurrency within the composite protocol indirectly ensures the FIFO guarantee.

4.2.3 Concurrency model

As already said in Section 2.1, Appia follows a single-threaded model, which prevents race conditions. In Cactus, the programmer needs to properly synchronise concurrent access to shared data within a micro-protocol, and within a composite protocol (if data is shared). A simpler solution is to consider a setting in which only one thread is allowed to execute in a composite protocol at a time (see Sect. 3.2.3). Such a design yields a concurrency model much closer to Appia in the sense that there is no need to include thread synchronisation in the micro-protocol code.

4.2.4 Interface with the environment

In Cactus, both the application and the network have to adopt the interface of a composite protocol, i.e., an x-kernel interface. This is appropriate for the network, since *send* and *receive* is what one usually uses in this context (*u-send* and *u-recv* in Figure 2). This message-based interface is however too restrictive for the interface with the application. Consider for example an application sending (1) messages that need to be totally ordered, and (2) point-to-point reliable messages. In Cactus, this would require to add some *type* information in

[4] Using two channels does not solve the problem, since a given event occurrence is bound to only one channel.

[5] A two-channel solution may work, but with such a solution, Consensus and Atomic Broadcast *are* aware of each other, which is not desirable.

the message in order for the protocol to be able to distinguish between the two types of messages.

In Appia, the interaction with the application and network is done by so-called asynchronous events (see Sect. 2.1). This is a much more flexible interface, allowing interactions more complex than just pushing/popping messages. Considering the above example, the application could trigger (1) *abcast* events and (2) *send* events. The only problem is that this works only from the application to the channel, but not the opposite. In the other direction, i.e., from the channel to the application, Appia does not provide any solution: the interface has to be implemented in an *ad-hoc* way. In our service, *adeliver* has been implemented using a producer-consumer queue.

To summarise, Appia's application interface is more flexible, but it would be desirable to have the same interface from the channels to the environment as from the environment to the channels.

4.3. Performance comparison

We measured the performance of Cactus and Appia on the *Reliable Channel* protocol module between two processes, using the IBM SockPerf benchmark suite, version 1.2 [14]. These benchmarks are designed to measure socket performance. In order to run these benchmarks, we implemented a special protocol module that interacts with the benchmarks: the SockPerf Proxy (see Fig. 5). Sockperf Proxy interacts with Reliable Channel to send/receive messages. The Reliable Channel protocol module is the same as in Section 3.

The hardware used for the measurements was (1) a 100 Base-TX Ethernet, with no third-party traffic, (2) two PCs running Red Hat Linux 7.2 (kernel version 2.4.18-19). The PCs have a Pentium III 766Mhz processor and 128 MB of RAM. The Java Virtual Machine was Sun's JDK 1.4.0.

4.3.1 Throughput

The first benchmark measures throughput with respect to message size. The benchmark does the following. First, the client connects to the server using event *connect* (Fig. 5), which causes a *join_remove_list* event to be triggered. This opens a socket at the TCP level. Once the socket is open, the client sends messages of a given size using event *send_bytes*. The handling of *send_bytes* results in triggering of *send*, which in turn triggers *tcp-send*. The client sends messages at maximum speed, and the throughput is measured.

In our Cactus implementation, as discussed in Section 3.2.3, the client benchmark's thread executes all the code in the composite protocol, including the *tcp-send* call on the socket. The TCP flow control blocks the calling thread if the sending buffer is full. In Appia, the client thread just inserts events *send_bytes* in the scheduler queue: execution of the event is carried by another thread. Because of the absence of flow control in Appia, our preliminary tests caused a memory overflow (overflow of the scheduler event queue). To prevent this, the designers of Appia added a preliminary flow control in the system, by bounding the size of the scheduler event queue: a caller wanting to add an event occurrence to the queue is blocked if the queue is full.

We have also measured the throughput directly using TCP sockets. The results are shown in Figure 6(a). The first observation is that the throughput grows linearly approximately up to messages of 2000 bytes. The growth slows down as we get closer to the network bandwidth limit (12.5 MB/s). This is not surprising: for large messages (e.g., 16384 bytes) the overhead of the framework becomes small compared to the network transmission time. Figure 6(a) shows also that for large messages the throughput obtained with Cactus is four times the throughput obtained with Appia. We come back to this issue later. The measurements also show that the overhead of both Cactus and Appia with respect to TCP is particularly significant for small messages.

4.3.2 Request-Reply

The Request-Reply benchmark measures the round-trip time of a message. The client first connects to the server in the same way as it does for the throughput benchmark. Once the socket is open, the client sends a message to the server and waits. The server receives the message and immediately resends it to the client. The time elapsed at the client between the message sending (*send_bytes*) and reception (*receive_bytes*) is measured (averaged over several request-reply interactions).

We have also performed the same measurements directly using TCP. The results are shown in Figure 6(b), where we can see that Cactus performs again better than Appia: the factor varies approximately from 2 for short messages to 5 for large messages. Compared to TCP, the overhead of both Cactus and Appia is again quite significant, especially for small messages.

4.3.3 Execution profiling

In order to understand the results of our experiments, we have used a profiler on the client side for the Request-Reply benchmark, with 16384 byte long messages. We have classified the methods executed into five groups: (1) waiting on the socket, (2) benchmark-related methods, (3) framework-related methods (including protocol execution), (4) thread synchronisation methods, and (5) message transmission and marshalling methods.

The results, are depicted in Figure 7. The time spent waiting on the socket does not appear, since it is not relevant to locate bottlenecks. For both frameworks, we see that most of the time is spent treating messages (84% in Appia, 66% in Cactus).[6] The percentage is particularly high for the case of Appia. We believe that the reason is because Appia implements a library (class *ObjectsMessage*) to serialise objects, which itself uses standard Java marshalling. In Cactus, Java marshalling is used without any intermediate library.

In any case, the results show that message marshalling represents the bottleneck of both Cactus and Appia, and thus probably explains the results that appear in Figure 6. Serialisation needs certainly to be done in a more efficient way. Standard Java class serialisation is too heavy-weight for simple data types such as *byte[]* (used by Java sockets).

[6] A similar result was obtained when profiling the execution on the server side.

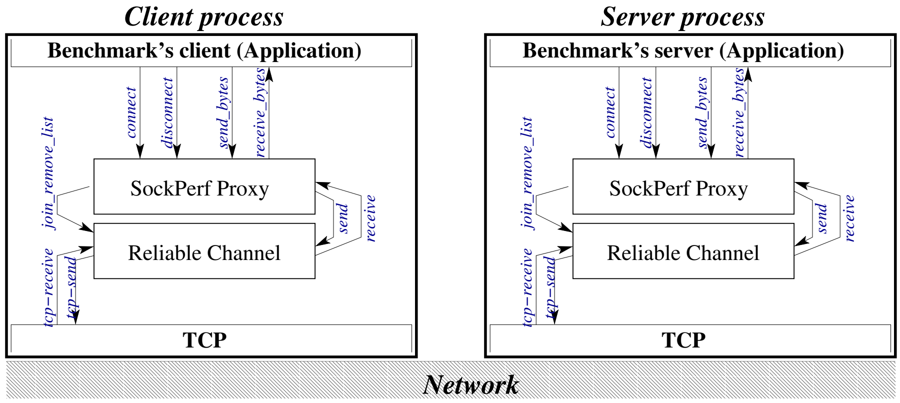
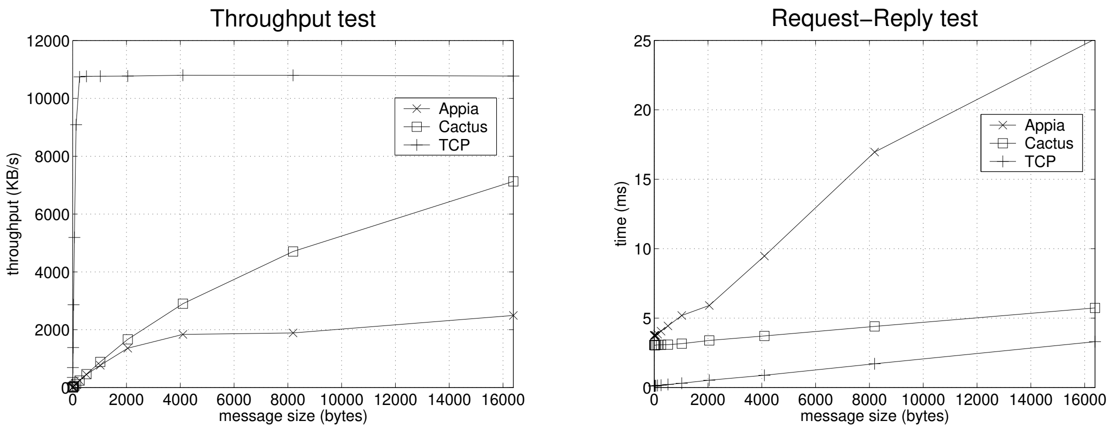


Figure 5. Benchmark configuration



(a) Throughput

(b) Request-reply

Figure 6. Performance comparison

5. Proposals for better frameworks

After having analysed the strengths and weaknesses of Appia and Cactus, we discuss in this section the most interesting features of the two frameworks, and propose improvements that could lead to better frameworks.

5.1. Composition model

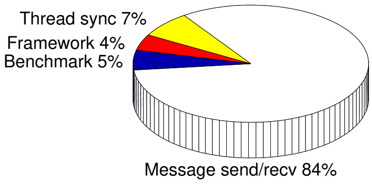
As we have seen in Sect 4.2, the best and most general composition model is non-hierarchical, allowing protocol modules to directly communicate with each other. More restrictive compositions, e.g., the Appia channel, enforce an order that is sometimes not the right one.

5.2. Interaction model

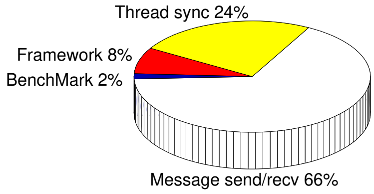
Point-to-point events. Our experience in building the Atomic Broadcast service shows that the interaction models proposed by Appia and Cactus do not fit well the situation encountered most of the time. Indeed, the most frequent situation was a protocol module triggering an event, and one single protocol module handling it. We call such events *point-to-point events*. Appia's channels are not very well adapted for point-to-point events.

We propose an additional way to compose protocol modules, in which there is a point-to-point binding between the event triggered in one protocol and the incoming event in the other protocol module. It is a simplified version of the Cactus event binding scheme, where only one handler can be bound to an event class. The Appia correction check could easily be extended to include this new binding rule.

Connectors. In Section 4.2.2, we have described the event multiplexing problem. With point-to-point events, the multiplexing problem can be solved with a special module called *connector*, which could be provided by the framework. A connector would allow a finer event routing. This is shown in Figure 8, which illustrates one particular type of connector called *multiplexor-demultiplexor*. If module m (respt. m') in



(a) Appia



(b) Cactus

Figure 7. Profiling of the Request-Reply benchmark with messages of 16384 bytes (client side).

Figure 8 triggers event $in1$ (respt. $in2$) at some process p, the multiplexor at the remote process q will trigger $out1$ (respt. $out2$). Note that this connector is different from the Cactus *demux* function (Sect. 2.2, Interaction model), which deals with sessions, while the connector deals with micro-protocols. Cactus' coordinated event raising (Sect. 2.2) could easily be implemented as a connector.

5.3. Concurrency model

To be general, the framework should allow multiple concurrency models in order to easily adapt to a changing environment (e.g., from mono-processor systems to multiprocessor systems). One of these models could be the Appia single-threaded model, which does not waste time in context switches. Another — more concurrent model — could be the *"thread per protocol module"* model, in which each protocol module has (1) its own thread, and (2) a (FIFO) queue of event occurrences, waiting for the protocol module thread to handle them. All the concurrency models should at least provide the following guarantees:

- *Default synchronisation:* Protocol programmers should be freed from the burden of expressing synchronisation. This implies, for instance, that atomic handler execution should be guaranteed by default.
- *FIFO event handling:* FIFO order event handling is desirable in concurrency models in which the property makes sense. This includes the single-threaded model, and the "thread per protocol module" model.
- *Transparency:* Changing the concurrency model should be transparent for protocol modules.

5.4. Interface with the environment

The interface with the environment can be divided into two issues: application/network interface, and interface to system resources.

In our opinion, a good interface from the application/network to the framework should be close to Appia (where the application/network can insert event occurrences into the framework). The interface from the framework to the application is trickier, since it depends on the application model. One solution for the framework is to provide various interfaces, which fit most common application models. A good solution for interaction with the network would be to support standards like APEX [22].

The manipulation of system resources (typically timers) should be well integrated in the framework event model (which is the case with Appia and Cactus).

A crucial aspect for performance is serialisation. We have noticed that the standard serialisation provided by Java is not adapted to short messages. The framework should provide efficient serialisation for primitive data types, and not just standard Java object marshalling (actually, a non-documented feature of Appia provides something similar).

6. Conclusion

In this paper, we have described Appia and Cactus, two frameworks for protocol composition. We have compared the two frameworks based on the experience gained in implementing an Atomic Broadcast service. Our conclusion is that the optimal framework would inherit features from both frameworks: a composition and interaction model close to Cactus, several concurrency models that would include Appia's model, and an extended version of Appia's interface with the environment. It would also include new features like point-to-point events and a repository of connectors.

We also feel that there is a lack of semantic foundations of protocol frameworks ([25] is a preliminary work in this direction). Such work is needed to prove the correctness of a complex protocol built using such a framework. Although there has been some work on formalisation of module composition [23], and correctness proofs of individual protocol modules [11], this work usually abstracts away from composition frameworks, focusing only on protocol modules. More needs definitely to be done.

Acknowledgements. We would like to thank Jean Vaucher for the implementation of the Atomic Broadcast module and for his help in designing a connector for Appia. We also thank Paweł Wojciechowski for his useful remarks.

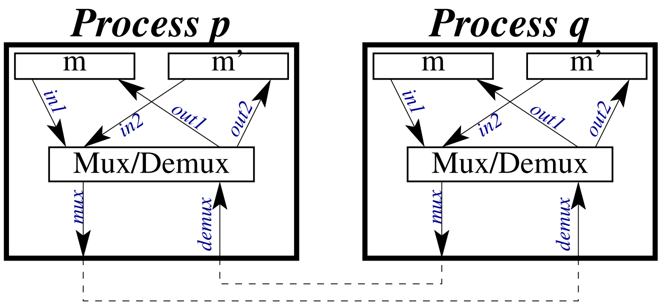


Figure 8. Multiplexor-demultiplexor connector.

References

[1] B. Ban. *JavaGroups 2.0 User's Guide*, Nov 2002.

[2] N. T. Bhatti. *A system for constructing configurable high-level protocols*. PhD thesis, University of Arizona, 1996.

[3] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4):321–366, Nov. 1998.

[4] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):37–53, December 1993.

[5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

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

[7] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[8] R. Ekwall, P. Urbán, and A. Schiper. Robust TCP connections for fault tolerant computing. In *Proc. 9th Int'l Conf. on Parallel and Distributed Systems (ICPADS)*, Chung-li, Taiwan, Dec. 2002.

[9] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. Technical Report 94-1425, Department of Computer Science, Cornell University, May 1994.

[10] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan. 8, 1998.

[11] J. Hickey, N. Lynch, and R. van Renesse. Specifications and Proofs for Ensemble layers. In R. Cleaveland, editor, *5th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 119–133. Springer Verlag, March 1999.

[12] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.

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

[14] IBM Corporation. *SockPerf: A Peer-to-Peer Socket Benchmark Used for Comparing and Measuring Java Socket Performance*, 2000.

[15] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996.

[16] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st Int'l Conf. on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, Apr.16–19 2001. IEEE Computer Society.

[17] H. Miranda and L. Rodrigues. Communication support for multiple QoS requirements. In *3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, Portugal, Apr. 1999.

[18] H. Miranda and L. Rodrigues. Flexible communication support for CSCW applications. In *5th Int'l Workshop on Groupware - CRIWG'99*, pages 338–342, Cacún, México, Sept. 1999. IEEE.

[19] S. Mishra, L. Peterson, and R. D. Schlichting. Consul: a communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1:87–103, 1993.

[20] A. Pinto. Appia group communication manual, Feb. 2001.

[21] R. V. Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Department of Computer Science, Cornell University, Apr 1996.

[22] M. Rose, G. Klyne, and D. Crocker. The application exchange core. RFC 3340, The Internet Society (IETF), 2002.

[23] P. Sinha and N. Suri. On simplifying modular specification and verification of distributed protocols. In *HASE'01*, pages 173–181, Oct. 2001.

[24] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.

[25] P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In *5th Int'l Conf. on Coordination Models and Languages*, volume 2315 of *Lecture Notes in Computer Science*, pages 389–404. Springer, April 2002.