Appia, a flexible protocol kernel supporting multiple coordinated channels*

Hugo Miranda Alexandre Pinto

Luís Rodrigues

Universidade de Lisboa†

{hmiranda,apinto,ler}@di.fc.ul.pt

Abstract

Distributed applications are becoming increasingly complex, often requiring the simultaneous use of several communication channels with different qualities-of-service. This paper presents the *Appia* system, a protocol kernel that supports applications requiring multiple coordinated channels. *Appia* offers a clean and elegant way for the application to express inter-channel constraints, such as, for instance, that all channels should provide consistent information about the failures of remote nodes. These constraints can be implemented as protocol layers that can be dynamically combined with other protocol layers.

*This work was partially supported by Praxis/ C/ EEI/ 12202/ 1998, TOPCOM.

†*Postal Address: FCUL, Campo Grande 1749-016 Lisboa, Portugal* Phone: +351 217500613 Fax: +351 217500084

1 Introduction

Distributed applications are becoming increasingly complex, offering rich and powerful services to their users. In order to offer satisfactory performance, these applications are also becoming increasingly demanding in terms of communication support. It is easy to find applications that require the simultaneous use of several communication channels, such as virtual environments, distributed simulation and computer supported collaborative work (CSCW).

One particularity of these applications is the need to exchange and disseminate several kinds of data, each with different quality-of-service requirements. Text messages or blocks in a file transfer, for example, are expected to be reliably delivered in FIFO order while in video streams some packets can be lost. As a result, these applications tend to rely on the use of multiple communication channels. It is easy to find communication substrates that support the use of several independent channels. However, in multi-channel applications there are often inter-channel constraints that need to be preserved to simplify the application logic. For instance, one may need to enforce FIFO, causal or total order constraints across channels, to cipher data in different channels using the same session key, or to ensure that consistent failure detection information is provided to all channels. If the protocol kernel does not provide any support for coordination among channels, this burden is left to the application designer.

A promising approach to tackle the complexity of these systems is to rely on configurable communication architectures that are able to support component re-utilization and composition. Recent protocol kernels, such as Ensemble [6] and Coyote [1] offer an environment where "off-the-shelf" micro-protocols can be combined to obtain different QoSs. However, few systems provide support for the coordination of multiple chan-

nels. Maestro [3] and CCTL [10] support coordination only at a limited set of properties such as membership or ordering. Inter-channels constraints that were not anticipated by these kernels still have to be implemented at the application level.

This paper presents *Appia* [1], a protocol kernel that offers a clean and elegant way for the application to express inter-channel constraints. This feature is obtained as an extension to the functionality provided by current systems. Thus, *Appia* retains the flexible and modular design that has previously proven to be advantageous, allowing communication stacks to be composed and reconfigured in run-time.

The paper is organized as follows. Section 2 presents several examples that motivate our work. Section 3 describes the innovative aspects of *Appia*. Section 4 describes the *Appia* implementation and Section 5 presents early performance results from a prototype written in Java. Our work is related with the current state of the art in Section 6. Section 7 concludes the paper.

2 Motivation

A protocol kernel is a software package that supports the composition and execution of micro-protocols. In terms of protocol design, the protocol kernel provides the tools that allow the application designer to compose stacks of protocols according to the application needs. In run-time the protocol kernel supports the exchange of data and control information between layers and provides a number of auxiliary services such as timer management and memory management for message buffers.

The *x*-Kernel [7] is an early and influential work on protocol composition. Protocols interact through generic push and pop primitives that allow layers to be im-

[1] *Via Appia* was one of the most important roads of the Roman Empire.

plemented with independence from each other. In run-time, x-Kernel supports the thread-per-message model, where a thread is assigned to process each message. Thus, the push/pop interaction between adjacent layers is implemented using function calls that are executed in the context of the message's thread. After x-Kernel, several systems have proposed different models of binding layers to stacks and of supporting event propagation. In Ensemble [6] and Coyote [1] events are data structures that are routed from one layer to the next by a specialized event scheduler.

Much emphasis has been put on the flexibility of protocol composition and on the efficiency of the event propagation mechanisms. Horus [13] and Ensemble [6] are protocol kernels designed to support group communication. They propose a strictly vertical stack composition where events must cross all the layers from the stack. Both frameworks predefine a fixed set of events whose semantics is well known. For efficiency reasons, Horus and Ensemble define particular exceptions to the strictly vertical model. For particular sets of events and stack configurations, Horus allows some layers to be bypassed using the FAST protocol [12]. On the other hand, in Coyote [1] micro-protocols are able to register the events they are interested in. This allows composing protocols in a manner that is not strictly vertical and also prevents layers from processing unnecessary events.

All these frameworks allow the application to define and use different communication channels, but they do not provide explicit support to enforce inter-channel constraints. The need for coordination among different channels used by the same application has been recognized in at least two existing frameworks: the Collaborative Computing Transport Layer (CCTL) [10] and Maestro [3]. CCTL uses a control channel where strong properties are enforced (total order, FIFO, Failure detection and Name Service) to coordinate data channels that may have weaker reliability and order-

ing constraints. Maestro is a group manager for protocol stacks. The base of Maestro is a core Ensemble stack that handles membership procedures for the data stacks. Maestro's data stacks can be created using a wide set of components, including UDP sockets, and CCTL or other Ensemble stacks.

Both frameworks are tied to specific inter-channel constraints, such as membership synchronization across several channels, and do not provide the appropriate interface to allow programmers to express alternative forms of coordination. In the following section we give several examples of coordination requirements that are not adequately addressed by previous protocol kernels.

2.1 Examples

Synchronized streams with different QoS To support interaction, a multimedia application opens different communication channels, one for each type of media (namely audio, video and control). These streams require different transport protocols, thus different communication stacks are used. However, all streams need to be synchronized. To achieve this goal, an inter-stream synchronization layer can be used as proposed in [4].

Another problem of using different stacks is that failures can be detected in a non-consistent way by the protocols in each stack. The Maestro [3] system tackles this issue by creating a "core" stack in charge of coordinating the others. The core stack coordinates by sharing joins, leaves and failure detectors.

An elegant solution based on protocol composition could use the stack of Figure 1. This approach consists of having a common failure detection layer at the bottom of each stream and a synchronization layer in the upper layers. The combination of the several "paths" creates a "diamond" structure.

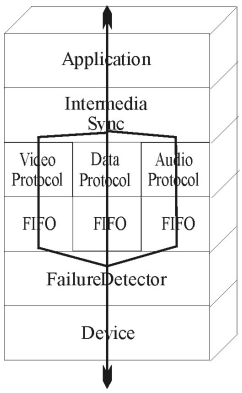

Figure 1: A multimedia stack with *diamond* shape

When events are inserted in *Appia*, they are tagged with a channel reference, defining the set of protocol instances they will traverse. From the event and protocol implementation view, this model becomes quite similar to strictly vertical layered frameworks like Horus [13]. However, *Appia* does not enforce that a one-to-one relation exists between protocol instances and stacks: one protocol instance may share its state over several channels. Some protocols can even be oblivious to the number of channels that traverse a session.

Light Weight Groups The second example is derived from the implementation of Light Weight Groups [11], a micro-protocol that maps several user-level groups in a single virtual synchrony group (Heavy Weight Group). When two or more groups have similar membership, utilization of Light Weight Groups (LWGs) promotes resource sharing and improves performance. The LWG layer hides the existence of user-level groups from the layers bellow it and different micro-protocol combinations can be multiplexed on the same Heavy Weight Group by the LWG layer. The model of LWGs is presented in Figure 2. The mapping between user level groups and virtual synchrony groups is dynamically established by the LWG protocol. This mapping may change in

run-time for performance reasons.

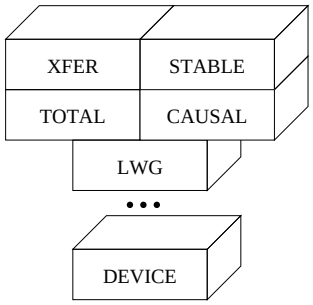

Figure 2: Light Weight Groups model

Although the main goal of *Appia* is to provide the mechanisms that simplify the task of expressing and implementing inter-channel constraints, the system retains the flexibility and performance characteristics of systems such as Ensemble and Coyote. In the following section we describe the *Appia* system.

3 The *Appia* System

Appia clearly separates the static and dynamic aspects of protocol compositions. Static aspects are used to model Qualities of Service and dynamic aspects are related with the implementation of these QoSs. Both aspects are present when new protocols are created and when protocols are combined to implement communication channels. This section explains how the above concepts interact to support inter-channel coordination.

We define a *layer* as the representative of a micro-protocol. Micro-protocols exchange information using events. All protocols implement the same *event interface*. The format and semantics of these events will be presented latter in this paper.

We define a *session* as an instance of a micro-protocol [7]. The session maintains

state variables used to process events. A session implementing an ordering protocol may maintain a sequence number or a vector clock as part of its state. In connection-oriented protocols, the session also maintains information about the endpoints of the connection.

Layers and sessions can be combined to satisfy inter-channel coordination requirements as follows. A *QoS* is defined as a stack of layers. The QoS specifies which protocols must act on the messages and the order they must be traversed. A *channel* is an instantiation of a QoS and is characterized by a stack of sessions of the corresponding layers. Sessions may be shared by more than one channel. Events exchanged between two sessions are delivered respecting FIFO order. A stack interfaces a given communication media through a DEVICE protocol. This is just an abstraction of any protocol outside the control of our communication kernel (for instance, TCP, UDP or IP Multicast).

3.1 Model

The model honors the distinction between the properties of a stack, captured by the QoS concept and each specific instance of a given QoS, the channel. Data flows through specific channels. In many systems, message processing requires every layer to perform a local demultiplexing operation to retrieve the appropriate session context. In *Appia*, like in Ensemble, the demultiplexing is performed only once, when messages enter the system and the target channel is retrieved. However, Ensemble uses kernel functions to retrieve messages from the network and to find the appropriate channel. *Appia* makes the model more flexible by delegating on protocols both operations.

Upon creation, a channel is as an array of "typed empty slots". Each of these slots must be filled with a session of the layer specified in the QoS for that position. Sessions

can be bound to the slots explicitly or implicitly by other sessions (automatic binding). By default, new sessions will be bound to the remaining slots.

Using explicit binding it is possible to associate specific sessions to specific channels. These sessions may either be already in use by other channels or may be intentionally created for the new channel. Explicit binding enables the user to have fine control over the channel configuration. For instance, on our Light Weight Groups example a single session of the LWG layer should be explicitly bound to all channels.

Using automatic binding it is possible to delegate on already bound sessions the task of specifying the remaining sessions for the channel. Typically, a mixture of explicit and automatic binding is used. Again, in the Light Weight Group example, the application should delegate on the LWG session the task of managing the remaining underlying sessions.

In *Appia*, inter-channel coordination can be achieved by letting different channels share one or more common sessions.

3.2 Configuration capabilities

A protocol is defined as *channel-aware* if its algorithm recognizes and acts differently upon reception of events flowing on different channels. As noted before, it is desirable to have the Failure Detector protocol to be channel-unaware, while Intermedia Sync is, by definition, channel-aware (it selects the desired Quality of Service for message sending). Protocol reusability would be limited in *Appia* if channel-awareness was mandatory.

However, it is possible to create protocols that are oblivious to the number of channels that traverse their sessions. A channel identifier *cid* is presented to sessions

on every event delivery. This value can be considered opaque by the session. If a new event is generated in response to a given incoming event (for instance a reply), the session should propagate the opaque *cid* value associated with the original event. Many of the sessions that can be found in existing stacks are channel-unaware.

Channel-awareness allows greater configurability possibilities. Sessions can use the channel information to learn the available QoSs and then choose which channel to use for their own events.

3.3 Events

Some frameworks support only a pre-defined set of events. The knowledge of the semantic of each event is then used to implement event-specific optimizations. We say that these frameworks support a *closed* event model. Closed models are very difficult to apply in different contexts since they only support a fixed set of interactions: the pre-defined set of events may not be enough to express, in an efficient manner, interactions required in other protocols. A framework that uses an *open* model, allows new events to be defined. Naturally, it is harder to implement optimizations in event routing when the set of events is unknown *a priori*. The model presented here tries to merge the advantages of a closed model with the flexibility of an open model.

Events in *Appia* are object oriented data structures. The `EVENT` class has two fundamental attributes: `channel` and `direction`. `channel` is a reference to the channel instance where the event will flow. `direction` can have only the `UP` or `DOWN` value to indicate in which direction the event is flowing along the stack. Like in most communication architectures, the lower layer makes the interface with the network and the upper layer makes the interface with the application. New events can be created by deriving from a previously defined event class (in particular, directly from `EVENT`). In

order to allow future event refinement, tests on the event type are always performed on the weakest class satisfying the desired requisites. The goal is to enforce event specialization using inheritance. This way, legacy protocols, unaware of the new event attributes, will continue to execute correctly.

As in Ensemble [6], sessions only interact with the environment by events. The basic `Event` class is extended by the kernel in two different branches: events to be sent to the network (class `SendableEvent`) and events containing requests to the kernel (class `ChannelEvent`) such as timers and notifications of channel initialization. Conceptually, the channel is positioned bellow the lowest session on the stack and above the upper one.

3.4 Run-time compatibility check

At QoS definition time, layers are requested to declare three event-related sets: Ψ_l containing the events that layer-l requires to provide a correct execution; Φ_l containing the events that layer-l is willing to receive and Γ_l containing the events that layer-l will generate.

A **correct stack** is one having every element of Ψ in Γ (Ψ and Γ are respectively the union of all Ψ_l and Γ_l sets on the stack). QoS definitions not respecting the above constraints will return an error and will not be created. Although this is not a complete stack validation tool, the model improves previous frameworks in this respect. For sanity, it is expected that for every set $\Phi_l, \Psi_l \subseteq \Phi_l$.

3.5 Efficient event routing

Under normal execution, only a few protocols add valuable information to messages [1, 13, 6]. For example, in a group communication stack not every protocol is interested in receiving view change information. Horus's FAST protocol bypasses a predefined set of layers under specific conditions. Our approach allows layers to explicitly state the events their protocols are interested in.

The event sets specified at QoS definition time are used for optimizing execution in run-time in the following way. For each event $e \in \Gamma$, an ordered set of layers will be constructed containing those that mentioned event e in their Φ set. Upon channel creation, this event routing tables will be ported to the channel. This operation will map QoS layers on instantiated sessions.

Event routing is static for each channel. On event arrival, the event table defined for its type at the target channel is associated with the event. Thus, an event is able to find the next session to be visited by simply keeping a pointer to an array. In *Appia*, most of event routing overhead is clearly pushed to QoS definition time, which is expected to happen at application startup. Using this approach, the introduced flexibility does not compromise run-time performance.

4 Architecture

Appia is being developed in Java.[2] Thus, inheritance can be extensively used to refine and specialize the main *Appia* components. The following classes represent the main components of the framework.

[2] `http://appia.di.fc.ul.pt`

Events Events make extensive use of inheritance. Programmers are free to define their own events, as long as they descend from the main `EVENT` class or one of its descendants. As we have mentioned before, the *Appia* kernel already defines two specializations of the `EVENT` class: `CHANNELEVENT` and `SENDABLEEVENT`.

Channel Events Every `CHANNELEVENT` is qualified with one of three values: `NOTIFY`, `ON` or `OFF`. `ON` and `OFF` qualifiers are typically used to activate/de-activate specific features such as the creation of debug logs. Notifications are triggered by the channel in response to relevant events such as timer expiration or stack initialization. The exact meaning of each qualifier may be refined for each event.

Timers *Appia* offers a timer management service. Sessions may start a timer with a specific timeout value and be notified of timer expiration (periodic timers are also available). Timer activation and notification is done through events of the class `TIMER`, a subclass of `CHANNELEVENT`. Qualifiers are used to optimize timer management. A timer is started using an event with the qualifier `ON`; the same object instance is used to provide the associated notification when the timer expires just by changing the event qualifier and direction in the stack, saving unnecessary data copies. Additionally, using inheritance, sessions can create specializations of `TIMER` that include all the context information for handling the timeout.[3]

Echo events A session may gather information about the remaining sessions in the stack by using `EchoEvents`. `EchoEvents` carry another event inside them; when a echo event reaches one side of the stack, the channel extracts the event being carried and forwards it in the opposite direction. On its way back, the returning event may

[3] *Appia*'s implementation of FIFO includes in the timer request the message to be retransmitted.

gather information about the traversed layers. For instance, a protocol attempting to temporarily prevent a stack from sending messages, may use this feature to receive the approval from the remaining sessions.

Sendable events The `SendableEvent` class defines a common interface to all events containing data to be sent or received from the network. Messages are expected to be serialized over an array of bytes, represented by the `MESSAGE` class. `SendableEvents` have a source and destination attribute that are untyped objects. It is up to the protocols to agree on the appropriate data format for these attributes. Source and destination can change while the event is traversing the stack: a Membership protocol, for instance, may convert a group name to an IP Multicast address.

The Message class The `MESSAGE`'s class interface has similarities with the message interface supported by the *x*-Kernel [9]. Both interfaces have the same goal: avoid unecessary memory copies (the difficulties of achieving this goal under the restricted memory model of Java are discussed in Section 5.2). The class assumes that messages are constructed using a stack model: headers are "pushed" by sessions on the sender endpoint and "popped" on the receiver. Memory is allocated in chunks. Whenever a `push` invocation exceeds the available memory in the first chunk of the stack, a new one is appended in a linked list fashion. Prior to being sent to the network, messages must be serialized in a flat array of bytes.

Event serialization *Appia* default behavior regarding event serialization follows the classic approach of making each session responsible for explicitly packing (and unpacking) the information to be sent to the network on the `Message` data structure. The session that interface concrete networks, simply extract the `source` and `destination`

attributes from the base SendableEvent class and obtain the payload from the message attribute. Given the use of the Java programming language, a different alternative would be to rely on the language serialization mechanisms to send an event (and all its attributes) to the network. There are however two advantages of *Appia*'s default behavior:

Portability Java serializes objects in a language specific byte array. By having each session to pack the message header in any desired format we ensure that it is possible to build a Java stack that interoperates with a corresponding stack written in a different language.

Over serialization Java default serialization procedures places the object and all its references in a stream. An event contains references to other *Appia* structures such as the channel and the event scheduler. Serializing an event in a straightforward way would result in the transfer of irrelevant information to the recipients.

Note that the default behavior can be specialized by a family of protocols that agree to use Java serialization as the standard message format. This can be easily obtained by defining an extension to the Message class that allows arbitrary Java objects to be pushed and popped.

QoS definition In *Appia*, QoSs are defined by passing one array of layers to the QoS class constructor. At QoS definition time, a QoSEventRoute object is created for each event e in the Γ sets of the QoS' layers. Each QoSEventRoute lists the layers that have declared e in their Φ set.

Channel definition Channels are created by issuing requests to the proper QoS. Channels are expected to be created by the application programmer or by other ses-

sions. Upon creation, channels have no bound sessions, which have to be created issuing request to the respective layers.

Channel cursors are provided to perform the configuration of the channel. Using a channel cursor it is possible to assign sessions to the channel in a safe way, since the cursor validates the session type against the layer type defined for the corresponding position in the channel. As previously stated, sessions can be bound to channels explicitly, automatically or by default. Binds occur in the following order:

1. Explicitly binding is performed immediately after the channel creation. When finished, the channel creator handles the control to the new channel.

2. The channel will then perform the automatic binding by inviting the sessions already bounded to fulfill the still empty positions. Whether the above sessions where explicitly created for this channel or already participate on other channels is completely transparent to *Appia*. This proves to be a powerful feature of *Appia* as it gives the chance to advanced protocols requiring channel introspection capabilities (like Light Weight Groups [11]) to tailor the channel in a optimal way.

3. Binding by default happens at the end of the process: the channel request to the layers of the empty slots the creation of a session for each. The channel is then initialized with a `ChannelInit` event.

At channel definition time, each `QoSEventRoute` is mapped into a `ChannelEventRoute`, that lists the specific sessions that a given event type has to traverse.

Protocol execution Sessions are the execution unit of protocols in *Appia*. They interact with the environment by sending and receiving events. Events are generated

by invoking the class constructor. The session creating the event is responsible for providing values for the main EVENT class attributes: `channel`, `direction` and `generator` (the session responsible for the event instance creation). The values in these attributes are used in the following way: the event's type and the `channel` attribute are used to retrieve the appropriate CHANNELEVENTROUTE; `generator` is used to locate the event on the CHANNELEVENTROUTE list of sessions and `direction` is used to indicate the next session to visit. Events are delivered to sessions through the invocation of the `handle` method. Sessions have access to all event attributes. Events are forwarded by invoking its `go` method. This method takes no arguments. Using the CHANNELEVENTROUTE information, the `direction` attribute and local state, the event knows the next session to visit.

Event scheduling EVENTSCHEDULER is the class responsible for queueing and delivering events to sessions. Currently, *Appia* provides a default implementation of the EVENTSCHEDULER. However, the architecture allows the scheduler to be specialized to fit specific classes of applications. By default, each new channel instantiates a private event scheduler, establishing a relation of one-to-one between channels and event schedulers. Programmers are free to establish relations of one event scheduler to many channels. The event scheduler interface was conceived to allow different scheduling models, possibly concurrent on the same *Appia* process. The default event scheduler is passive and schedules events in response to an invocation to a `consumeEvent` method.

One of the tasks of the APPIA class is to multiplex event schedulers. In the first prototype, we have opted for a single threaded execution of all event schedulers (Section 3 discusses this option). Thus, the APPIA class is a static class that executes an infinite loop in the `run` method, and cyclicly calls the `consumeEvent` method of the registered event schedulers. The APPIA class does not need to be aware of internal

scheduling policies of the registered event schedulers.

Threads Three reasons have been reported in the literature against the use of multi-threading scheduling of events in protocol kernels: performance, programming complexity and portability [8, 2]. However, the "thread per message" model was successfully used in at least two frameworks: x-Kernel [7] and Horus [13].

Java threads do not suffer from the portability problem of threads mentioned in [8]. Nevertheless, to avoid the context switching and synchronization overhead we have opted to assume that event scheduling is single-threaded. Sessions handle and pop events in the conxtext of the *Appia* scheduler thread. This does not prevent the *Appia* kernel from using additional threads for other tasks. For instance, a dedicated thread manages timers.

On the other hand, implementing a strict event-driven model in *Appia* is not possible because Java API does not provide a system call capable of blocking on several descriptors for a certain time period.[4] Thus, *Appia* uses a non-strict event driven model, close to the one implemented by Ensemble [6]: sessions are free to use internal threads to implement their own functionality. To allow concurrent threads to spontaneously generate events (outside the context of the `handle` invocation), channels provide a thread-safe `async` method. When this method is invoked, the channel introduces an `Async` event in the event scheduler.

The asynchronous call is particularly usefull for sessions handling external events: those that need internal threads. Two protocols were implemented using this model: a protocol that interfaces an UDP socket and a protocol that interface the user. Both have a internal blocking thread that, when unblocked (on datagram arrival or user

[4]That is, no equivalent to the C's `select` system call is available.

input,) calls the `async` method of the channel.

5 Evaluation

This section evaluates the Java implementation in two separated aspects. Firstly, the multi-channel coordination support is illustrated. Secondly, performance figures obtained with the current prototype are given.

5.1 Example of multi-channel coordination

The code in figure 3 shows the steps executed in the QoS and channel definition for the implementation of the multimedia synchronization example in section 2.1. For brevity only fragments of the code are presented.

The first step in the definition of *Appia* stacks is layer instantiation (line 2). Layers will be passed as arguments to the QoS constructor (line 10) and (if necessary) used to instantiate sessions (line 17). Lines 5 to 11 show the steps necessary for the creation of a QoS: first the definition of an array with references to the layers and finally the instantiation of a QoS object having the layers array as the argument to the class constructor. Line 14 shows that a channel is created with the `createUnboundChannel` method of the QoS. Forcing the same session instance to be present in several channels is performed in lines 20 to 29: after the session creation, channel cursors are used to bind the session to the desired channels. On this particular case the procedure should be repeated for sessions of the Application, Intermedia Sync and Device protocols. The binding of the remaining sessions on the channels (FIFO and the media protocols) will be performed by default. Line 30 deliveries the remaining channel configuration to the channel. Automatic and default binding is performed on the `start` method.

```
1   // Layer instantiation
    Layer fdLayer=new FDLayer();
    // ...

5   // QoS definitions
    Layer[] audioLayers=new Layer[6];
    audioLayers[0]=deviceLayer;
    audioLayers[1]=fdLayer;
    // ...
10  dataQoS=new QoS(dataLayers);
    // ...

    // Channel instantiation
    Channel dataChannel=dataQoS.createUnboundChannel();
15
    // Retrieve sessions from layers
    FailureDetectorSession fdSession=FailureDetectorLayer.createSession();
    // ..

20  // Binding one single failure detector session for the three
    // channels   (channel cursor handling ommited)
    dataCCursor.setSession(fdSession);
    videoCCursor.setSession(fdSession);
    audioCCursor.setSession(fdSession);
25
    // The same procedure should be done for Application, Intermedia
    // Sync and Device sessions. Remaining session can be left empty.

    // Start the channels
30  dataChannel.start();
    // ...
```

Figure 3: Implementation of the intermedia sync example

The above example assumes that the INTERMEDIA SYNC session was not designed to query QoSs and perform automatic bindings. An alternative implementation could rely on this protocol to bind the shared sessions when requested by the channel. In this case, the only bounded session would be the Intermedia Sync.

5.2 Performance

In this section we present the overall performance of *Appia* on a network composed of Pentium processors running Windows NT. The framework was running over a Java 1.2.2 virtual machine. The workstations were connected with a lightly loaded 10Mbps Ethernet.

Message class The Message class design mainly concern was to optimize the most expected usage patterns. Our initial work tried to apply to Java the approach followed by *x*-Kernel [9] (that was written in the C programming language). The results were disappointing. We believe that the reason for the low performance was the impossibility to directly access memory addresses in Java. Keeping in mind the goal of minimizing data copies, we performed a number of adaptations to the *x*-Kernel model.

Table 1 present the average results of the pushing and popping tests.

The push tests take a initially empty message and executes the number of operations necessary to achieve a 10000 bytes message. The Message class is configured to allocate 1000 bytes per message list node; so 10 memory allocations are always performed.

The pop test takes a message with 10000 bytes and empties it by performing the necessary number of operations. The message is in a contiguous array of bytes. This is the expected behavior in normal execution: the byte arrays received from sockets are passed to the message constructor and no message list nodes are generated at the receiver.

Overall performance A simple stack was defined to measure the overall performance of the prototype. The stack was composed of four layers providing UDP sockets

Test	**Result** (μs)		
	10 bytes	100 bytes	1000 bytes
Push	2.864	4.278	14.683
Pop	2.599	2.605	2.802

Table 1: Message performance results

access, FIFO reliable delivery, fragmentation and user interface. Messages exchanged over the network have a length of 90 bytes. Table 2 shows the round trip delay of *Appia* using the above stack.

Round trip (ms)	2.962

Table 2: Performance measures of Appia

Message processing avoidance One of the interesting features of *Appia* is that session only process the events they have subscribed. To measure the impact of this facility, the TRANSPARENT and the GHOST protocols were created. The TRANSPARENT protocol does not accept any events. The GHOST protocol handles all events but simply forwards every event to the next session. Channels with different number of these protocols were created. Each channel had a TEST protocol on top. The TEST protocol sent sequences of `EchoEvents` and measured the time until receiving the echoed event.

Table 3 shows that the TRANSPARENT protocol doesn't affect the kernel execution maintaining the event round trip delay on the $19\mu s$. The first column is the control value: the result of the execution having only the test protocol on the channel. The differences in the results between 0 and 10 TRANSPARENT protocol sessions result from events on the workstation hosting the tests and can be ignored. The results of the

# sessions	Event round trip delay (μs)				
	0	1	2	3	10
TRANSPARENT	19.61	19.63	19.51	19.64	19.54
GHOST	19.61	29.75	39.95	52.13	140.38

Table 3: Round trip delay of `EchoEvents` with GHOST and TRANSPARENT protocols

GHOST protocol however, shows that presenting unwanted events to protocols can be an adverse factor on performance. Each GHOST protocol session adds between $101\mu s$ and $120\mu s$ (depending on the number of instances) to the round-trip of the EchoEvent.

6 Related Work

x-Kernel [7] was pioneer on the separation of protocols from sessions, which is the base model for our work. However, x-Kernel provides no support for flexible event exchange; this has been perceived has a constraining factor in the design of micro-protocols [1, 13]. Furthermore, in x-Kernel, message routing within the stack is determined on a per-layer basis. Each layer has to demultiplex each message explicitly and to keep state to ensure that the message is forwarded to the appropriate next layer.

Coyote [1] offers no hierarchical composition of layers. Events on this framework are intended to be handled in parallel by all layers and have no syntactic or semantic restriction. Concepts as "session" and "channel" have no representation. While *Appia* allows generic protocol to process *classes* of events, in Coyote each protocol only accepts a set of concrete event types.

In Bast [5] the same layer may be implemented by different strategies. It is possible to adapt the strategies in run-time but QoSs are statically defined at coding time.

Efficient event handling was studied in the context of the Horus [13] system, but with strong restrictions. The FAST (Protocol Accelerator) layer [12] allows optimizations that are not currently supported by the *Appia* model: events are processed by layers depending on the session state. On the other hand FAST is not generic and can only be used in a very limited fashion: the set of protocols to be bypassed is well defined. Coordination of channels was implemented on some specific layers (such as the Light Weight Group Layer) [11], but no systematic support was given to allow the designer to express which sessions should be shared by different channels. Ensemble [6] was strongly focused on formal protocol validation. In order to accomplish stack correction guarantees, strong restrictions were made. Although it is a follow up from Horus, Ensemble can't handle in an efficient manner the coordination of concurrent channels.

7 Conclusions

This paper introduces *Appia*, a protocol kernel that tries to balance the flexibility in protocol composition with run-time efficiency. With *Appia*, protocol stack designers specify the events produced and subscribed by each layer. In run time, the application may construct the sequence of protocols layers that is needed to enforce the desired semantics. Specialized event dispatchers for each QoS ensure the efficient routing of events in the kernel. Instances of QoS, called channels, may share sessions. This allows the construction of complex stacks, where different channels may share common properties.

References

[1] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, November 1998.

[2] K. Birman. *Building secure and reliable network applications.* Number ISBN 1-884777-29-5. Manning Publications Co., 1996.

[3] K. Birman, R. Friedman, and M. Hayden. The maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Cornell University, Ithaca, USA, February 1997.

[4] M. Correia and P. Pinto. Low-level multimedia synchronization algorithms on broadband networks. In *The Third ACM Intl. Multimedia Conference and Exhibition (MULTIMEDIA '95)*, pages 423–434, San Francisco, November 1995. ACM Press.

[5] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proc. of the 18th Intl. Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[6] M. Hayden. *The Ensemble System.* PhD thesis, Cornell University, Computer Science Department, 1998.

[7] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Trans. on Software Engineering*, 17(1):64–76, January 1991.

[8] S. Mishra and R. Yang. Thread-based vs. event-based implementation of a group communication service. In *Proc. of the 1st Merged Intl. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 398–402, Orlando, Florida, USA, March 1998. IEEE Computer Society.

[9] D. Mosberger. *Message Library Design Notes*, January 1996.

[10] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed collaboration systems. In *Proc. of the 17th Intl. Conf. on Distributed Computing Systems*, pages 43–50, Balitmore, Maryland, USA, May 1997. IEEE.

[11] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Veríssimo, and K. Birman. A transparent light-weight group service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 130–139, Niagara-on-the-Lake, Canada, October 1996.

[12] R. van Renesse. Masking the overhead of protocol layering. In *Proceedings of the 1996 ACM Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 96–104, Palo Alto, CA USA, August 28–30 1996.

[13] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76–83, April 1996.