Appia, a flexible protocol kernel supporting multiple coordinated channels[*]

Hugo Miranda Alexandre Pinto Luís Rodrigues
Universidade de Lisboa, FCUL
Campo Grande 1749-016 Lisboa Portugal
{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.

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

[*]This work was partially supported by Praxis/C/EEI/12202/1998, TOPCOM. Selected portions of this report were published in the Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21) April 16-19, 2001. Phoenix, Arizona, USA (Poster session).

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.

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. Section 6 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 [5] is an early and influential work on protocol composition. Protocols interact through generic push and pop primitives that allow layers to be implemented with independence from each other. After x-Kernel, several systems have proposed different models of binding layers to stacks and of supporting event propagation. In Ensemble [4] and Coyote [1] events

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

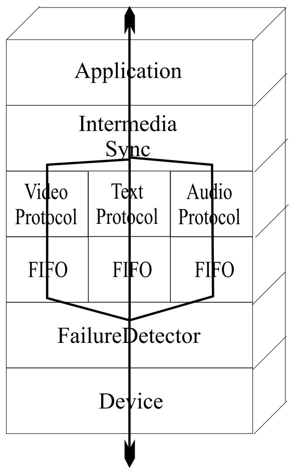

Figure 1: A multimedia stack with *diamond* shape

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 [8] and Ensemble [4] 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 [7]. 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) [6] and Maestro [2]. CCTL uses a control channel to coordinate data channels that may have weak reliability and ordering 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. We now present an example of coordination requirements that are not adequately addressed by previous protocol kernels.

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 text). 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 [3]. Another problem of using different stacks is that failures can be detected in a non-consistent way by the protocols in each stack.

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.

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 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 [5]. 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.

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 thus defining a quality of service by enumerating the properties it will provide. 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 or UDP).

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 example, a single session of the INTERMEDIASYNC 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.

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 one.

Events in *Appia* are object oriented data structures. New events can be created by deriving from a previously defined event class (in particular, directly from the main event class). 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 [4], sessions only interact with the environment by events. 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. For sanity, it is expected that for every set $\Phi_l, \Psi_l \subseteq \Phi_l$.

Although this is not a complete stack validation tool, the model improves previous frameworks in this respect. By using the functional features of the ML language to validate the correct execution of a stack, Ensemble [4] takes a stronger approach to the problem. *Appia* simply performs a run-time check of the composition, assuming: 1) that the protocols were coded correctly and 2) the semantic of the events is well known.

3.5 Efficient event routing

Under normal execution, only a few protocols add valuable information to messages [1, 8, 4]. 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 Implementation

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.

Events Events make extensive use of inheritance. 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.

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 channel event subclass.

A session may gather information about the remaining sessions in the stack by using `EchoEvents`. `EchoEvents` are a subclass of `ChannelEvent` that 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 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.

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.

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

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 sessions. 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.

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

5 Performance

In this section we present the overall performance of *Appia* on a network composed of Pentium III processors at 500MHz 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. Besides the two hosts used on the tests, the network included 3 other hosts. All of them connected by a single hub.

A simple stack was defined to measure the overall performance of the prototype. The stack was composed of four layers providing UDP sockets access, FIFO reliable delivery, fragmentation[3] and user interface. Messages exchanged over the network have a length of 90 bytes. The round trip delay was averaged from bunches of 100 operations. In the above conditions, the measured round trip delay of *Appia* was $2.962ms$.

To evaluate the overhead of using *Appia*, one control program also writen in Java measured the round trip delay of messages using the UDP protocol. UDP datagrams took $0.542ms$. For crossing 4 times the stack, *Appia* delays each message by $2.42ms$.

[3]Due to the sze of the exchanged messages, the fragmentation layer did not execute usefull work

	Event delay (μs)				
# sessions	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 1: Delay of `EchoEvents` with GHOST and TRANSPARENT protocols

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 them 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 1 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 come from external events on the workstation hosting the tests and can be ignored. The results of the GHOST protocol however, shows that presenting unwanted events to protocols can be an adverse factor on performance. Each GHOST protocol session adds between $10,1\mu s$ and $12,0\mu s$ to the return of the event.

6 Conclusions and future work

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.

Two improvement directions are currently being taken: extend the available protocol set and improve the performance results. A group communication protocol set is almost concluded. This will be an important step that will extend the *Appia* applications one step further from its current prototype status.

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, 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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