The design of *RT-Appia* *

João Rodrigues
Instituto Nacional de Engenharia e Tecnologia Industrial (INETI)
Departamento de Electrónica
Paço do Lumiar, Lisboa, Portugal
joao.carlos@mail.ineti.pt

Hugo Miranda, João Ventura, Luís Rodrigues
Universidade de Lisboa (FCUL), Campo Grande, 1749-016 Lisboa, Portugal
hmiranda@di.fc.ul.pt
jcv@ieee.org
ler@di.fc.ul.pt

December 14, 2000

Abstract

This paper presents the design of *RT-Appia*, a framework for the development and implementation of configurable real-time protocol stacks. The goal of *RT-Appia* is to allow the construction of specialized protocols through the composition of pre-defined micro-protocols. *RT-Appia* attempts to balance the flexibility and efficiency of micro-protocols with the predictability requirements of real-time applications.

1 Introduction

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 models and 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

* Parts of this report will be published in the Proceedings of the Sixth IEEE International Workshop on Object-oriented Real-Time Dependable Systems, Rome, 8-10 January 2001. This work has been partially supported by the PRAXIS/P/EEI/14187/1998 project, DEAR-COTS. J. Ventura has been supported by the PRAXIS XXI Programme under grant PRAXIS XXI/BM/20729/99.

number of auxiliary services (such as memory management for message buffers and timer management).

The x-Kernel [7] is an early and influential work on protocol composition. A version of x-Kernel adapted to real-time operation has been developed in the scope of the CORDS project [15]. Following the initial work with x-Kernel, many other protocol kernels have been designed with enriched functionality. Notable examples are Ensemble [5], Coyote [2]/Cactus [6] and *Appia* [8]. Although these systems offer more flexible infrastructures to the development, composition and execution of micro-protocols, most of them were not designed with the goal of supporting real-time operation. This paper describes the design of *RT-Appia*, a real-time extension of the *Appia* protocol kernel. *Appia* has been developed in the Java programming language and *RT-Appia* is designed to use the facilities introduced by the Real Time Specification for Java [3].

RT-Appia retains the flexibility of its non real-time counterpart: complex stacks can be composed of different channels and each channel may use a different quality of service. If needed, channels may share state at more common protocol layers and this facilitates the implementation of coordination policies among channels.

One common goal in the design of real-time systems is that they must behave in a way that is predictable with respect to timing requirements. This means that it must be possible to show, demonstrate or prove that these requirements are meet for the lifetime of the system [10]. Real-time systems must support flexibility, however too much flexibility can destroy predictability [11]. *Appia* relies on a single thread to manage all communication stacks, does not take priorities into account while scheduling events, and makes intensive use of the garbage collector. On the other hand, *RT-Appia* allocates memory and computing resources to each channel to achieve predictable behavior.

The paper is organized as follows. Section 2 motivates *RT-Appia* and surveys the related work. Section 3 briefly describes the *Appia* system and Section 4 describes the extensions that we designed to support real-time operation. Section 5 shows preliminary schedulability analysis results. Finally, Section 6 concludes the paper.

2 Motivation and Related work

The design of a protocol kernel must take into consideration several distinct aspects of protocol development and execution. From the point of view of protocol design the kernel must provide a framework to support the clean composition of micro-protocols. This encourages the re-use of protocol components and allows the applications to configure protocol stacks exactly tailored to their needs. This

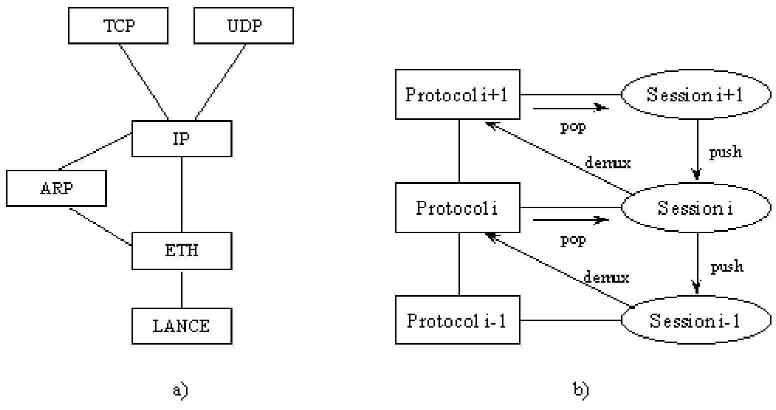

Figure 1: x-kernel.a) A protocol graph. b) Using push to pass a message down a stack and demux and pop to pass a message up a stack.

aspect is particularly relevant in the context of real-time applications where, due to memory and power consumption constraints, it is interesting to execute in each component just the protocol layers required to support the intended functionality.

The x-Kernel [7] introduced a model in which protocol stacks are building as a graph of protocols organized hierarchically. This simple model as three primitive communication objects to support it: *protocols*, *sessions* and *messages*. Protocol objects represent conventional protocols and serve to major functions: they create session objects and they demultiplex messages received upward. Session objects are dynamically created and can be seen as instances of protocols. Loosely speaking, protocol objects export operations for opening channels, resulting in the creation of a session object, and session objects export operations for sending and receiving messages. Messages objects move through sessions via PUSh and POP session operations. Session push operation is invoked to pass a message downward. While flowing upward, a message alternatively visits a protocol via its DEMUX operation and then a session via its pop operation. The protocol demux operation makes the decision to which session the message should be routed. Figure 1 presents these relationships.

So x-Kernel supported composition by restricting the communication between adjacent layers to the exchange of data PUSH, data POP, and control events. More recent systems, such as Ensemble [5], Coyote [2] and *Appia* [8], offer a more flexible structure by allowing the exchange of a richer set of events.

From the point of view of support for protocol execution, the protocol kernel implements the mechanisms that support the exchange of events among layers. 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 as 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 [5] and Coyote [2] events are data structures that are routed from one layer to the next by a specialized event scheduler. In real-time applications, the event scheduler must allow priorities to be assigned to different events and communication channels. It must also ensure that the priorities of each communication channel (with respect to other channels and the remaining activities in the system) are respected.

Additionally to scheduling of events, a protocol kernel usually also provides auxiliary libraries that simplify protocol implementation, such as efficient interfaces to message buffers (offering primitives to add and remove headers to messages) and timer management. A real-time communication kernel must support resource reservation to ensure that the resources required by a channel, including buffer requirements, are available during the protocol execution.

CORDS [15] and Cactus [6] are framework that systematically addresses the problem of supporting protocol development in real-time environments. CORDS extended the original x-Kernel system to include resource reservation mechanisms. These resources include memory allocation, CPU cycles (the ability to select the number of threads and their scheduling attributes) and network bandwidth. The "unit" of resource reservation is called a PATH. CORDS follows the x-Kernel "thread per message" model and allows the application designer to select the number of threads and their scheduling attributes (policy and priority) for a PATH. This work is supported by the real-time kernel OSF MK (derived from the Mach kernel [1]). The kernel Integrated Time-Driven Scheduler (ITDS) allows the user to define several scheduling policies for a set of processors. Thread synchronization mechanisms provide a priority inheritance protocol to avoid priority inversion problems. This kernel also provides a memory management mechanism to avoid unpredictable delays in paging faults [14].

Cactus extends the Coyote system to support the construction of configurable real-time services as *composite protocols*. A real-time service is implemented by combining finer-grain micro-protocol modules with the Cactus runtime system. A micro-protocol interact with other micro-protocols using event propagation. A micro-protocol is the basic building block in Cactus and is a software module that implements a well-defined property of the desired service. A micro-protocol, in turn, is structured as a collection of event handlers, which are procedure-like segments of code bound to be executed when a specified event occurs. When such event occurs all event handlers are stored in an ordered event handler queue (EHQ) and are executed serially by a dispatcher thread. Execution of handlers are atomic

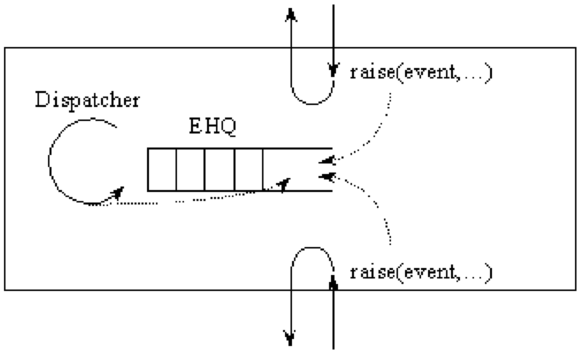

Figure 2: Composite protocol.

i.e. each handler is executed to completion without interruption. Figure 2 illustrates this design. In the figure EHQ is the event handler queue. Solid arrows represent threads of execution, including the dispatcher thread associated with EHQ. The other threads represent the threads invoking the x-kernel push and pop operations from the protocols above and below, respectively. The dashed arrows represent the handlers being inserted into EHQ as a result of raising an event through the RAISE procedure call in the composite protocol.

Cactus uses CORDS as its underlying communication subsystem, included in the OpenGroup/RI MK 7.3 Mach operating system. In Cactus each composite protocol is a protocol in a CORDS protocol graph. Thus composite protocols interact through the x-kernel procedure call interface push and pop operations. An application may open multiple logical connections called sessions with its own quality of services requirements (QoSs). A session may have one or more CORDS paths assigned to each QoS. Event handlers associated with different QoSs within a composite protocol are assigned to corresponding CORDS paths.

Cactus supports implementing real-time channels with several shapes as a composite protocol and uses system resources allocated as CORDS PATHS. Resource allocation has been divided into the channel control module (CCM) and admission control module (ACM). The CCM translates channel arguments into general resource requirements as a set of micro-protocols, paths and CPU cycles that satisfy these requirements. The ACM is a process that mantains information about available resources, and based on this information, either grants or denies requested resources.

The CactusRT system uses the concept of micro-protocols and a good execution model. However this system is dependent on x-kernel and if we want a stack of composite protocols we must call the push/pop operations and miss the flexibility

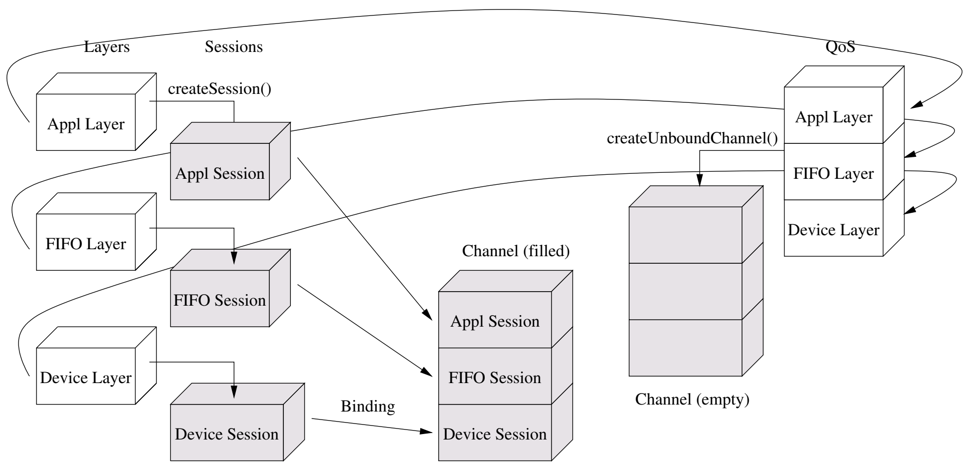

Figure 3: Relation among QoS, layers and session in *Appia*.

of event propagation. This system seems to have one thread per composite protocol (dispatcher) and a thread per message (inherited from CORDS). This leads to two context switches per composite protocol. Furthermore the non-preemptive scheduling of CORDS results in a priority inversion between traffic types with different deadlines and with a worst case duration being the maximum across all handler execution times.

In this paper we also follow the approach of extending an existing protocol kernel, the *Appia* system [8], with the mechanisms required to support real-time operation. Since *Appia* offers a more flexible structure than x-Kernel, with this effort we attempt to balance the flexibility and efficiency of micro-protocols with the predictability requirements of real-time applications.

3 *Appia*

The *Appia* system is an object oriented framework developed in Java providing a set of classes to support definition and implementation of protocols stacks [8]. In *Appia* each stack is composed of one or more *channels*. Each channel is an ordered sequence of *sessions*, instances of a specific *protocol layer*. The session maintains state that is used by the layer to process events. A layer that implements an ordering protocol may maintain a sequence number or a vector clock as part of the session state. In connection oriented protocols, the session also maintains information about the endpoints of the connection. The sequence of layers associated with a given channel defines the quality of service implemented by the channel. An important aspect of an *Appia* stack is that different channels may share session at one or more layers. This mechanism supports the implementation of inter-channel coordination policies. The relation among QoS, layers and session in *Appia* is illustrated in Figure 3

Communication between layers is made by exchange of *events*. Events are object oriented data structures, all descendant of the EVENT main class. New events can be created by deriving from a previously defined event class. In order to allow future event refinement, event type tests are always performed on the weakest class satisfying the desired requisites. The goal is to support event specialization using inheritance. This way, legacy protocols, unaware of the new event attributes, will continue to execute correctly. At quality-of-service definition time, each layer declares the set of events the layer produces and that the layer is interested in subscribing. Using this information, *Appia* constructs event routes that maintain the exact set of sessions that need to be visited by each event, optimizing the time needed for the event to traverse the channel.

The routing of events is implemented by an EVENTSCHEDULER, a passive component responsible for selecting the next event to be processed. It is possible to assign different schedulers to different set of stacks but all schedulers are activated by a single thread, avoiding the need to implement concurrency control on every session. A discussion of the comparative advantages of this strategy against the "thread per message" model can be found in [9]. However, *Appia* is not single threaded. Additional threads are used to manage timeouts and to interface the network.

Some of the advantages of *Appia* are the use of an open event model that allows the protocol designer to define the set of events more appropriate to a target application area, the fine grain configuration possibilities and the efficient mechanisms for event routing. Thus it provides an excellent framework to develop specialized and efficient protocol stacks to be used in real-time applications.

4 *RT-Appia*

The *RT-Appia* adds to the original *Appia* system a number of mechanisms to ensure the predictability of execution of real-time communication stacks.

A real-time channel (RTChannel) is a schedulable object and has a priority attribute to distinguish channels with different traffic priorities. Figure 4 presents a RTChannel. Channel is an instance of QoS and is a concept imported from *Appia*. Events represents the events supported by RTChannel. EventScheduler is responsible for scheduling and executing events in the channel.

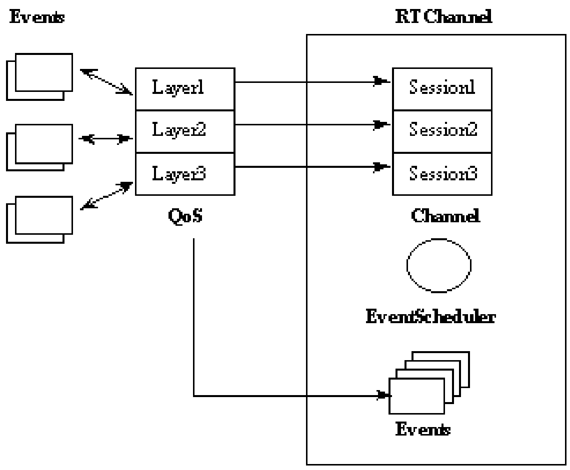

Figure 4: RTChannel.

Appia was implemented in Java language. One of strongest advantages of Java is its platform independence through the adoption of a virtual machine, a feature advertised as "write once run everywhere". However, the basic Java platform and language does not provide adequate support for development of real-time applications, which the unpredictable latency introduced by the garbage collector is an example. The Real-Time for Java Experts Group (RTJEG) is currently polishing the Real-Time Specification for Java (RTSJ) [3, 4]. The goal of RTJEG is to provide a real-time software development platform, suitable for a wide range of applications, with backward compatibility with Java. RTSJ defined seven main areas to achieve this goal: scheduling, memory management, synchronization, asynchronous event handling, asynchronous control transfer, asynchronous thread termination and access to physical memory. The mechanisms added to *Appia* rely of the basic mechanisms provided by the RTSJ. Namely, *RT-Appia* introduces the following features:

- Preallocation at initialization time of memory to allocate events and message buffers.
- A REALTIMETHREAD is associated with each *RT-Appia* channel. To each channel is allocated a private set of resources, including memory resources allocated from a SCOPEDMEMORY area. There is also prioritized event scheduling within each channel.
- Timer management using the TIMER mechanism.

- Channels interface with the user and with the network using asynchronous event handlers.

Each of these features is discussed with more detail in the subsequent sessions.

4.1 Memory allocation

In *RT-Appia*, each channel owns an individual set of resources including a permanent memory area. The memory allocation overhead is pushed into the channel instantiation time in *RT-Appia*. In channels, one can find permanent and transient objects. Permanent objects are those that will exist as long as the channel instance exists. Examples of these objects are SESSIONS and EVENTROUTES[1]. Permanent objects are allocated from a memory are reserved for the *RT-Appia* kernel. The *RT-Appia* kernel is responsible for allocating and releasing these resources.The quantification of the necessary memory must be performed prior to channel instantiation. The memory required by each session can be obtained from the layers composing the Quality of Service of the channel. Additional objects like the CHANNEL and the EVENTSCHEDULER have constant memory requirements that can be added to those of the channels.

Transient object examples are EVENTS, MESSAGES, and channel specific session state. These objects will be allocated from a SCOPEDMEMORY area reserved for each channel. Since the amount of memory required for each channel depends on the specific needs of the chosen layers, at channel startup (when handling the CHANNELINIT event) each SESSION is requested to preallocate all the required data structures and all instances of events that it may generate. Thus, each session will have several pools of predefined events (one pool for each class of event) that it generates. In run-time, events are taken from the pool by each session and returned to the pool by the kernel (an event may be returned to the pool after being processed by the last session in its route). Pools of events must implement the OBJECTPOLL interface defined by the *RT-Appia*.

Data structures associated with protocol messages are managed by the MESSAGE class and must be associated with events (there is a particular subclass of EVENT, the SENDABLEEVENT, that owns reference to a data message). While in the basic *Appia*, the size of the data buffers associated with each message grows and shrinks dynamically while headers are added or removed, in *RT-Appia* we have opted to maintain the size of messages fixed once allocated. This avoids the need to maintain a message heap for each channel. *Appia* already owns the introspection mechanisms that allows a layer to dynamically obtain the size of the headers required by the lower layers.

[1]Each Event Route keeps the sequence of sessions to be visited by each event type.

4.2 Scheduling

The original *Appia* system uses a single thread to schedule all the events. As we have noted before, this strategy tries to minimize the overhead associated with concurrency control and context switching. We have opted to retain this approach in *RT-Appia* but opted to assign a different REALTIMETHREAD for each protocol channel. Since each channel owns private resources, this allows the preemption of a channel by another channel of higher priority.

Within each channel, the programmer may also associate a different priority to each event. Event priorities may be used by any specialization of the basic *Appia* EVENTSCHEDULER that takes priorities into account. It should be noted however, that all events of the same channel are processed by a single REALTIMETHREAD.

The scheduler maintains all events that flow in the channel in a data structure that we call the SCHEDULABLE EVENT SET. The internal structure of the set takes into consideration the priorities of the events but also the requirement to preserve the FIFO order of events of the same priority exchanged between any pair of layers. When invoked, the event scheduler selects the SCHEDULABLE EVENT of highest priority and delivers it to the session that it should visit in its route. The processing of the event by a session is performed by the HANDLE method of that session. During the processing of the HANDLE method, the session may insert new events, possibly of higher priority, in the set of schedulable events. However, the invocation of the method HANDLE is not preempted by the insertion of other events in the set. In particular, if the session decides to propagate the current event in the channel it must re-insert the event being processed to the schedulable set. Only when HANDLE returns, the scheduler picks a new (or the same) event to be consumed. Naturally, the processing of events of a channel can always be preempted by an event of a higher priority channel.

4.2.1 Multi-channel sessions

Multi-channel sessions are those being shared by more than one channel. The possibility of having one session being used in more than one channel is an important feature of *Appia* as it allows inter-channel coordination. Multi-channel sessions can easily suffer priority inversion problems due to: i) concurrency control in the access to the shared data structures; ii) inter-channel coordination features that are specific of each session.

Since *Appia* tries to avoid unnecessary synchronization costs, calls to a given session are not synchronized by default. On the other hand, in *RT-Appia* we have opted to assign a different thread to each channel, thus, sessions shared by different session require the use of synchronization. Sharable sessions in *RT-Appia* are

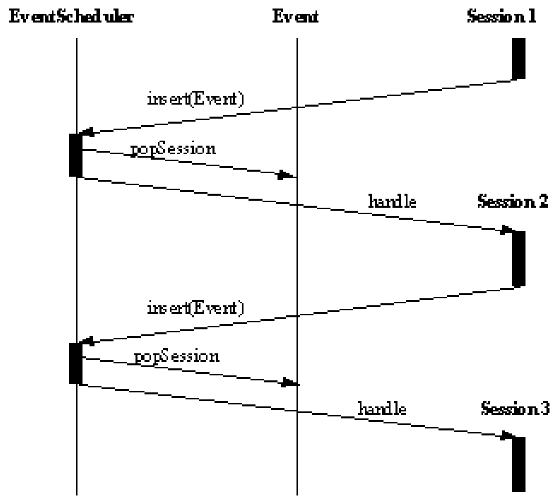

Figure 5: Example of an event execution in a channel.

required to implement a pair of PREHANDLESYNC and POSTHANDLESYNC synchronization methods where session specific synchronization is implemented. The PREHANDLESYNC should be invoked *before* the HANDLE method and POSTHANDLESYNC invoked *after* the HANDLE method returns. These method are automatically called by the *RT-Appia* run-time only in the case the session is being shared by different channels. Typically, PREHANDLESYNC will lock the session resources and POSTHANDLESYNC will unlock the same resources, although more spohisticated fine-grain concurrency control mechanims can be implemented (readers-writers, etc). Needless to say, multi-channel sessions must be used with care.

Figure 5 presents an example for exchange of messages between objects in RTChannel, for an event who visit two sessions in a channel. Session1 produce a event and invoke the method *insert* of the Event object. The event is inserted in the SCHEDULABLE EVENT SET. To find the next session to visit, EventScheduler invoke *popSession* in the Event object. Finally EventScheduler executes the protocol, by invoke *handle* in the Session2 object.

4.3 Timer management

In *RT-Appia* the timer management is very similar to its implementation in the basic *Appia* system. Timeouts are triggered using special events that are captured by the *Appia* kernel. When the timer expires the kernel re-injects the event in the channel's schedulable event set. While in *Appia*, the list of pending timers is managed by a dedicated thread, in *RT-Appia* the asynchronous TIMER mechanism is used. When a timer expires the event must be asynchronously inserted in the schedulable set. Thus the methods that perform insertion and removal of events from this set must by synchronized.

4.4 Interfacing the channel

A communication channel interfaces the network and the application at well defined layers. Typically, every channel has an application layer and a network layer. These layers are responsible for inserting events in the channel in response to user stimuli or messages arriving from the network. In *RT-Appia* these layers are responsible for defining the asynchronous events and asynchronous event handlers associated with those stimuli. The handlers use the same SCOPEDMEMORY of the channel and should allocate the events from the pool of the associated interface layers. Like the timers described above, these events are inserted asynchronously in the channel's schedulable events set.

5 Schedulability analysis

To determine if all the objects used by *RT-Appia* are schedulable or not, it is necessary to perform an analysis of the complete system. This includes not only the computation time used by the *RT-Appia* objects, but also the time taken by message transmission. The proposed method is based on the work of [12, 13] and is the subject of an ongoing study [16]. The mathematical model used assumes all events are periodic. Sporadic events can be handled through the use of a periodic task that acts as a server for this type of events, called the sporadic server.

To perform schedulability analysis of the *RT-Appia* objects, it is necessary to derive the Worst Case Execution Time (WCET) of any given channel activation. We have previously noted that the *RT-Appia* stores an event route for each event. The event route defines the set of layers visited by a given event. From the event route and the WCET of the HANDLE methods it is possible to derive the worst case computation time of any given event. However, the processing of an event at one session may generate other events. To perform the schedulability analysis it is also necessary to capture the chain of events with the longest computation time. Note that a chain of events must have a finite number of events. A chain of events terminates when a channel has no further events to process and must remain idle waiting for stimuli from the application, network or timers. The WCET of a channel activation is derived from the computation time of the worst case chain of that channel and from the WCET of other higher priority channels.

From the WCET of a channel activation and the periods of the relevant events that activate the channel it is possible to derive a conservative figure for the schedulability analysis of a given channel (the analysis of channels that share sessions with other channels is more complex). Note that this figure is conservative, since not all channel activations process the longest chain of events. Additionally, a chain of events may contain events of different priorities and it would be interesting to obtain WCET figures for each specific event. Techniques to exploit the finer detail given by the theoretical model are still under study. For instance, since the theoretical model described in [16] takes the distributed computation into account, it would be interesting to schedule additional channel activations in order to process response events as soon as they arrive. This is possible because the distributed model computes the offsets introduced by the network and remote endpoint delays.

6 Conclusion

RT-Appia is a protocol kernel that supports the development and execution of real-time protocol stacks that can be constructed through the composition of micro-protocols. The approach allows the application to create protocol stacks that exactly match their requirements. *RT-Appia* has been designed to run on the real-time Java environment and is currently being implemented. The reference implementation of the RTSJ, scheduled for early 2001 will be used to evaluate the architecture.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for Unix development. In *Proceedings of the Summer Usenix*, July 1986.

[2] 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, Nov. 1998.

[3] G. Bollella and J. Gosling. The real-time specification for java. *IEEE Computer*, 33(6):47–54, June 2000.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. URL: `www.javaseries.com/rtj.pdf`.

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

[6] M. A. Hiltunen, X. Han, and R. D. Schlichting. Real-time issues in cactus. Technical Report AZ85721, Department of Computer Science, University of Arizona, Tucson, 1996.

[7] N. C. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engeneering*, 17(1):64–76, Jan. 1991.

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

[9] 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, Mar. 1998. IEEE Computer Society.

[10] J. A. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *The Journal of Real-Time Systems*, 2:247–254, 1990.

[11] J. A. Stankovic and K. Ramamritham. A reflective architecture for real-time operating systems. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 23–38. Prentice-Hall, Inc, 1995.

[12] K. W. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Technical Report YCS197, Department of Computer Science, University of York, Apr. 1993.

[13] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS221, Department of Computer Science, University of York, Jan. 1994.

[14] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In USENIX, editor, *Mach Workshop Conference Proceedings, October 4–5, 1990. Burlington, VT*, pages 73–82, Berkeley, CA, USA, Oct. 1990. USENIX.

[15] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the 2nd IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Laguna Beach, CA, Feb. 1996.

[16] J. C. Ventura and L. Rodrigues. Timing analysis of object-oriented communication protocols. Di/fcul tr, Departament of Computer Science, University of Lisbon, Sept. 2000.