Real-Time Dependable Channels: Customizing QoS Attributes for Distributed Systems

Matti A. Hiltunen   Richard D. Schlichting
Xiaonan Han   Melvin M. Cardozo   Rajsekhar Das

TR 98-06

Real-Time Dependable Channels: Customizing QoS Attributes for Distributed Systems [1]

Matti A. Hiltunen Richard D. Schlichting Xiaonan Han Melvin M. Cardozo
Rajsekhar Das

TR 98-06

Abstract

Communication services that provide enhanced Quality of Service (QoS) guarantees related to dependability and real time are important for many applications in distributed systems. This paper presents real-time dependable (RTD) channels, a communication-oriented abstraction that can be configured to meet the QoS requirements of a variety of distributed applications. This customization ability is based on using CactusRT, a system that supports the construction of middleware services out of software modules called micro-protocols. Each micro-protocol implements a different semantic property or property variant, and interacts with other micro-protocols using an event-driven model supported by the CactusRT runtime system. In addition to RTD channels, CactusRT and its implementation are described. This prototype executes on a cluster of Pentium PCs running the OpenGroup/RI MK 7.3 Mach real-time operating system and CORDS, a system for building network protocols based on the *x*-kernel.

July 9, 1998

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grant N00014-96-0207 and the Defense Advanced Research Projects Agency under grant N66001-97-C-8518.

1 Introduction

An increasing number of distributed applications must provide enhanced Quality of Service (QoS) guarantees related to dependability and real time. Implementing these requirements individually can be difficult enough, but having to deal with the combination can make the problem virtually intractable. One problem is that the two types of requirements often conflict, resulting in a tradeoff situation. For example, increasing the reliability of a communication service through retransmissions may cause a delay that results in a missed deadline. Another problem is that the actual properties to be enforced can have many variations, which can force developers to program each application from scratch to realize the exact combination required. For example, a collection of clients communicating with a replicated set of servers may have particular requirements related to atomicity of message delivery, consistent ordering of messages at the servers, how responses are collated, and how timely the response must be.

This paper describes *real-time dependable (RTD) channels*, a communication-oriented abstraction that is designed to simplify the development of distributed applications with both dependability and real-time requirements. RTD channels are implemented as middleware mechanisms between the operating system and application, thereby providing a virtual machine with enhanced QoS guarantees on which applications can be built. Moreover, RTD channels are highly configurable with respect to the properties they enforce, which allows the same supporting software to be used for a variety of applications and for a variety of execution environments.

The configurability of RTD channels is based on using CactusRT as the implementation vehicle. CactusRT is a system that supports the construction of middleware services with real-time constraints as *composite protocols* by combining finer-grain *micro-protocol* modules together with the CactusRT runtime system. Services built using CactusRT execute on a cluster of Pentium PCs running the Mach MK 7.3 real-time operating system from OpenGroup/RI [Rey95]. The emphasis on integrating a range of QoS attributes and on providing a high degree of flexibility distinguishes CactusRT from other configurable systems that support a degree of customization [Her94, RBM96, SBS93, SVK93, TMR96]. Thus, in a larger context, RTD channels illustrate the feasibility of constructing customized abstractions that merge dependability and real time using CactusRT.

This paper has several goals. One is to present RTD channels as a useful abstraction for distributed applications based on their ability to support a variety of customized QoS attributes. A second goal is to describe how RTD channels can be implemented using CactusRT; this implementation is currently nearing completion. Finally, a third goal is to describe CactusRT itself, including its event-driven programming model and features that enable it to be used for highly configurable middleware that can meet timeliness requirements.

2 RTD Channels

A channel is an abstraction for communicating between two or more application-level processes on different sites in a distributed system. A dependable channel provides guarantees related to the reliability of message transmission, while a real-time channel provides timeliness guarantees. A real-time dependable (RTD) channel has a combination of dependability and timeliness guarantees.

The system model consists of multiple machines connected by a network. As noted, our focus is on middleware services for distributed real-time systems, especially those related to interprocess communication. Thus, we model the software at each site as a layer between the application and the operating system (OS); the application makes requests to the middleware layer (e.g., message transmission) and receives information as appropriate (e.g., message reception). The middleware in turn uses the facilities provided by the underlying OS and network software to implement the guarantees desired by the application.

2.1 Channel Shapes

Different types of RTD channels can be defined based on whether the traffic is unidirectional (UD) or bidirectional (BD), and how many processes the channel connects and in what manner. A point-to-point channel (PP) connects two processes, a multi-target (MT) channel connects one source to many targets, and a multi-source channel (MS) connects multiple sources to a single target. MT and MS channels are equivalent if the channels are bidirectional, so we use BDM to refer to either. Finally, a multi-source, multi-target channel (MST) connects multiple sources to multiple targets. A special case of an MST channel is a group multicast (GM) channel, where the sources and targets are identical. The different channel shapes are illustrated in figure 1.

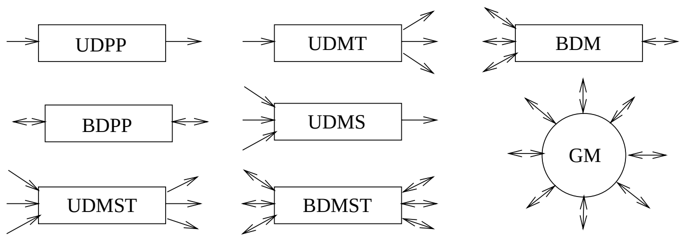

Figure 1: Channel shapes

Each channel shape has application areas for which it is the most suitable choice. For example, UDPP is good for multimedia transmission, BDPP for implementing video phones, UDMS for transmitting sensor input from replicated sensors to a controller, and GM for video conferencing and distributed multiplayer games.

2.2 Channel Properties

A large number of properties can be defined for channels, but for brevity, we describe only a representative set here. In particular, we consider *real-time*—i.e., whether each message sent on a channel will be delivered to its destinations by a deadline, *reliability*—i.e., whether each message sent on a channel reaches its destinations, and *message ordering*—i.e., in what order messages are delivered when they arrive. Other properties not considered here include bounded jitter, atomicity, stability [PBS89], security, and properties related to changes in the set of processes using the channel, such as virtual synchrony [BSS91] and safety [MMSA$^+$96]. All of these properties can be implemented in CactusRT using the same design principles as described in section 4.

Real-time properties. Real-time services often specify not only deadlines but also the importance of the service meeting its timeliness constraints, usually stated using the phrases “hard real-time” or “soft real-time”. We use a more general metric that captures this constraint by defining for each channel a probability P_D called the *deadline probability*, which is the required probability that a given message reaches its destination by the specified deadline. Various techniques such as admission control, scheduling, congestion control, and retransmissions are used to provide the desired P_D, although the requested deadline and the characteristics of the underlying system naturally constrain the range of feasible values. For example, if the network between a sender and receiver has suffered a permanent failure, no messages can reach their destination by the deadline. Thus, P_D in this case can be at most the same as the probability of no permanent network failure.

Reliability properties. We specify reliability using a similar probability metric P_R called the *reliability probability*, which is the required probability that a given message sent on a channel eventually reaches its destination, either before or after its deadline. Note that P_D and P_R are related in the sense that establishing a given deadline probability necessarily fixes a minimum reliability level. The reliability of a channel can be improved using redundancy, either in the form of redundant physical communication links (e.g., [KKG$^+$90, CASD95]) or message retransmissions. Naturally, the number of retransmissions or redundant links depends on the failure rates of the underlying networks.

No fixed number of retransmissions or redundant communication links can guarantee a reliability of 1.0 in all cases. However, if the retransmissions are allowed to continue until the destination receives the message, a reliability of 1.0 can be achieved. Thus, we can identify two different types of reliability guarantees: *bounded reliability* and *absolute reliability*, where bounded reliability is based on a fixed maximum number of retransmissions and absolute reliability may require an arbitrary number of retransmissions. Typically, bounded reliability would be implemented by transmitting the message a fixed number of times with fixed small time intervals between transmissions, whereas absolute reliability would use feedback from the receiver in the form of a positive or negative acknowledgment to determine if and when a message should be retransmitted.

These reliability variants have implications on the real-time properties of the channel. As noted above, P_R is the upper bound for P_D and thus, to reach the required deadline probability, the reliability of the channel may need to be increased using reliability techniques. Note, however, that a real-time guarantee can be made only for messages that reach their destination after a fixed number of retransmissions. Therefore, only the bounded reliability techniques will help to increase P_D, whereas both naturally increase P_R. For a channel with some timeliness requirements and an absolute reliability requirement, both of the reliability techniques can be used together, i.e., a message is first transmitted a fixed number of times and if it still does not reach its destination, additional retransmissions are done using a positive or negative acknowledgment scheme.

Finally, different applications have different requirements for P_D and P_R. For example, multimedia applications can often tolerate the loss of a few packets or missed deadlines, so P_D and P_R can be relatively low—on the order of 0.9 to 0.99 for audio and a minimum of 0.999 for video [MHCN96]. On the other hand, distributed financial systems such as automatic teller machines would likely require a value of P_R very close to 1.0, but could tolerate a somewhat smaller value of P_D.

Ordering properties. Message ordering properties define constraints on the relative order in which messages are delivered to the application at the target sites. An *unordered channel* is the weakest and imposes no constraints, while a *FIFO channel* delivers messages from any given process to the application in the same order as they were sent. In a *causally* ordered channel, if message m_i (potentially) causes a process to send message m_j, then any site that delivers both m_i and m_j delivers m_i before m_j. An application receiving message m_i before sending m_j is typically used as the basis for causality. Finally, in a *totally ordered channel*, all sites receiving any two messages m_i and m_j deliver them to the application in the same order.

Numerous other ordering properties could also be defined, including timestamp-based temporal order [Ver94], semantic order [MPS89], and synchronous order [MG95]. Note also that these definitions are orthogonal to real-time and reliability. This means, for example, that the only guarantee for FIFO channels with $P_D < 1.0$ is that any messages delivered are in order, i.e., there may be gaps in the message delivery sequence [AS95]. We can also design and implement variations of the ordering properties that enforce strict ordering in all cases even if it means that some messages may miss their deadlines. The choice of such tradeoff—that is, what should happen when both ordering and real-time cannot be guaranteed for a message—is naturally application specific.

The set of properties that can be provided by a channel is intricately related to its shape. For example, with a UDPP channel, the only ordering guarantee that can be defined is FIFO, while total order can only be defined for multi-source multi-target channels. Conversely, the need for certain channel properties can dictate the shape required for an application. For example, from the communication point of view, an application could replace any channel with a collection of simple UDPP channels. However, in this case, the application would have to manage the potentially numerous channels and implement the properties found in more complex channels itself.

3 CactusRT

3.1 Overview

The design and implementation of RTD channels are based on CactusRT, a system that supports the modular construction of middleware services with real-time constraints. CactusRT is derived from the *x*-kernel model for building network subsystems in which the software is implemented as a graph of *protocols* (i.e., software modules) organized hierarchically [HP91]. CactusRT extends this model with a second level of composition by allowing internal structuring of *x*-kernel protocols as collections of *micro-protocols*.

CactusRT has evolved from a previous system called Coyote, which has been used to construct highly-customizable versions of communication services without real-time constraints, including group RPC [HS95, BS95], membership [HS98], and atomic multicast [GBB$^+$95]. Although CactusRT is linked with the *x*-kernel model, the concept of micro-protocols and the execution model supported by the system can be implemented using any number of different vehicles. For example, a second version of Cactus without support for real time is being constructed on Sun Solaris using C++ and CORBA [OMG98].

3.2 Event-Driven Model

The approach used in CactusRT is based on implementing different semantic properties and functional components of a service as separate modules that interact using an event-driven execution model. As mentioned, the basic building block of this model is a micro-protocol, which is a software module that implements a well-defined property of the desired service. A micro-protocol is, in turn, structured as a collection of *event handlers*, which are procedure-like segments of code that are executed when a specified *event* occurs. Events are used to signify state changes of interest, such as “message arrival from the network” (MsgFromNet). When such an event occurs, all event handlers bound to that event are executed. Events can be raised explicitly by micro-protocols or implicitly by the CactusRT runtime system. Execution of handlers is atomic with respect to concurrency, i.e., each handler is executed to completion without interruption.

Event handler binding, event detection, and invocation are implemented by a standard runtime system or *framework* that is linked with the micro-protocols to form a *composite protocol*. The framework also supports shared data (e.g., messages) that can be accessed by the micro-protocols configured into the framework. Once created, a composite protocol can be composed in a traditional hierarchical manner with other protocols to form the application’s protocol graph.

The primary event-handling operations are:

- *bid* = **bind**(*event, handler, order, static_args*):

  Specifies that *handler* is to be executed when *event* occurs. *order* is a numeric value specifying the relative order in which *handler* should be executed relative to other handlers bound to the same event. When the handler is executed, the arguments *static_args* are passed as part of the handler arguments. **bind** returns a handle to the binding.

- **unbind**(*event, handler, bid*): Removes a binding *bid* previously established by **bind**.

- **raise**(*event, dynamic_args, mode, delay, urgency*):

  Causes *event* to be raised after *delay* time units. If *delay* is 0, the event is raised immediately. The occurrence of an event causes handlers bound to the event to be executed with *dynamic_args* (and *static_args* passed in the **bind** operation) as arguments. Execution can either block the invoker until the handlers have completed execution (*mode* = SYNC) or allow the caller to continue (*mode* = ASYNC). *urgency* is a numeric value specifying the relative urgency of executing the handlers with respect to other handlers that are already queued for execution (typically determined by the message deadline and type).

Other operations are available for such things as creating and deleting events, halting event execution, and canceling a delayed event. An event raised with *delay* greater than 0 is also called a *timer event*.

The event mechanism and shared data structures provide a degree of indirection between micro-protocols that facilitates the configuration of different collections into functional services. The model also has a number of appealing features for real-time systems. One is that event handlers are short and simple, with predictable timing behavior. Thus, given the CPU time for each handler and knowledge of which events are raised by each handler and the framework, we can easily calculate the CPU time needed by the entire service for any configuration. Another is that atomic execution of handlers minimizes the need for synchronization between handlers, and thus, reduces the well-known problems of synchronization, such as priority inversion. These issues are addressed further below.

3.3 Implementation

The prototype implementation of CactusRT runs on the OpenGroup/RI MK 7.3 Mach operating system, which includes real-time support and the CORDS communication subsystem [TMR96]. CORDS is essentially identical to the *x*-kernel, but with an additional *path* abstraction to support reservation of resources such as buffers and threads. We utilize paths and the thread priorities provided by the system to assign priorities to channels. Given the application traffic model and the composite protocol execution time per message, we can calculate if there is a set of priorities for all channels in the system such that all deadlines can be satisfied.

In CORDS, the networking subsystem is constructed as a protocol graph. This graph can be divided between user and kernel space, with protocols in the latter having better predictability (e.g., no page faults) and faster performance (e.g., fewer context switches). In general, standard practice is to develop and debug protocols in user space and then move as much of the graph into the kernel as possible to gain the predictability and performance advantages.

Protection domains in CORDS are based on address spaces. In this context, this means that there are separate domains for the portion of the protocol graph in the kernel and for the portions in each application's user space. Domains are protected by binary semaphores, which are used by CORDS to ensure that at most one thread is active in each protection domain at a time. When a message enters a protection domain, a thread is taken from the thread pool to shepherd the message through the domain. No other thread can

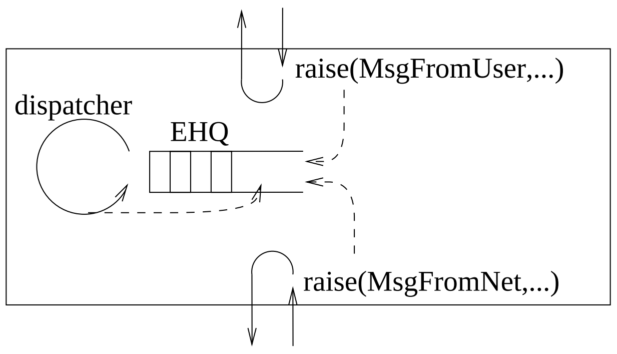

Figure 2: Framework

enter until the first thread either leaves the domain or voluntarily yields control. Thus, the scheduling within a CORDS protection domain is non-preemptive.

In our design, one or more of the protocols in that graph are composite protocols. An application may open multiple logical connections called *sessions* through the protocol graph to other application level objects. Each session typically has its own QoS guarantees. Our basic design uses one CORDS path for each session.

All threads associated with a single path have the same CORDS priority and thus, can be executed in arbitrary order by the MK OS scheduler. In many cases, however, this is not sufficient. For example, if two messages arrive from the network to a composite protocol session approximately at the same time but one was delayed longer by the network, the delayed message should often be processed first. Similarly, an acknowledgment message should often be processed as soon as possible to avoid unnecessary message retransmission.

To realize this scheduling of handlers within a composite protocol, an ordered *event handler queue* (EHQ) storing pointers to handlers awaiting execution is maintained. Handlers are added to the EHQ when an event to which they are bound occurs and removed when executed. The order of handlers in the EHQ is determined by the relative urgency of the corresponding event. Handlers are executed by a *dispatcher* thread that executes handlers serially.

Figure 2 illustrates this design. In the figure, solid arrows represent threads of execution, including the dispatcher thread associated with the EHQ. Protocols above and below the composite protocol interact with it through a procedure call interface—the *x*-kernel *push* and *pop* operations—and thus, the threads from these components operate within the composite protocol as well. The dashed arrows in the figure represent handlers being inserted into the EHQ as a result of event occurrences.

In addition to the EHQ, each composite protocol contains an *event/handler map* (EHM) that maintains information about which handlers are bound to each event. Handlers for a given event are stored in this structure according to their relative ordering requirements.

The implementation of event handling is shown in figure 3. As can be seen in this figure, the implementation of **raise** depends on the mode supplied as an argument in the call. A non-blocking raise (ASYNC) simply inserts the handlers bound for this event into the EHQ. A blocking raise (SYNC), however, is more complex since the handlers bound for the event must be executed prior to resuming the handler that invoked the raise operation. In our design, the thread executing the raise operation assumes the role of the dispatcher until all the handlers for this particular event have completed. This condition is detected by creating a counter that is initialized to the number of new handlers inserted into the EHQ and decrementing it each time a handler is executed. When this counter reaches zero, the execution of

```
raise(event, dargs, mode, delay, urgency) {
    if (delay $>$ 0) evSchedule(later,delay,event,dargs,urgency);
    else { handlers = EHM(event); addArgs(handlers, dargs); /* add dargs to handler info blocks */
        if (mode == ASYNC) insert(handlers);
        else { C$_h$ = new counter(number of handlers); for each h: handler: h$\rightarrow$C$_h$ = C$_h$;
            insert(handlers,urgency); dispatcher(); } }
}
void later(event,dargs,urgency) {
    thread$\rightarrow$priority = channel$\rightarrow$priority; raise(event,dargs,ASYNC,0,urgency);
}
void dispatcher() {
    while(true) { yield CPU to higher priority thread;
        P(S$_n$); h:HandlerInfo = head(EHQ); call h$\rightarrow$func(h$\rightarrow$args);
        if ($--$(h$\rightarrow$C$_h$) == 0) return(); /* last SYNC handler is done*/ }
}
void insert(handlers,urgency) {
    insert handlers to EHQ ordered by urgency; S$_n$ += number of handlers; }
```

Figure 3: Event-handling pseudo code

the raising handler is resumed by the dispatcher executing a return operation. Note that more than one handler may be blocked concurrently at a raise operation associated with the EHQ. To guarantee that the return operation resumes the correct handler in this case, handlers must be ordered in the EHQ so that the last handler to block is the first to resume operation (LIFO). This can be achieved simply by ensuring that the urgency of the raised handlers is always greater or equal to the urgency of the handler that executed the raise operation. Finally, raising an event with delay > 0 is implemented using the timer mechanism provided by CORDS to realize the appropriate delay, as illustrated in figure 3.

Two other aspects of the code are worth noting. First, a semaphore S_n whose value corresponds to the number of handlers in the EHQ at any given time is used to prevent execution of dispatcher threads whose EHQs are empty. Second, in the procedure `later`, the priority of the timer thread is lowered to match the current path priority. This is done because CORDS uses separate a high priority thread pool to implement timer events.

Another issue concerns priority inversion, that is, the situation where a lower priority thread is executing when a higher priority thread is pending. Such inversions should be avoided in real-time systems, since no timeliness guarantees are possible if the maximum priority inversion time is not bounded. The scheduling of threads in different CORDS protection domains is preemptive and so, will not cause priority inversions. However, as noted above, scheduling within a CORDS protection domain in non-preemptive, implying that a higher priority thread attempting to enter the domain will be blocked if a lower priority thread is executing. This priority inversion can become unbounded when interaction with threads in other protection domains is taken into account.

Unbounded priority inversion due to synchronization is a classical problem in real-time systems and we use the standard solution of priority inheritance [SRL90] to solve the problem. CORDS version 7.3 supports the Posix-compliant `pthreads` package, which provides priority inheritance semaphores. We modified CORDS to use these semaphores for synchronizing access to protection domains, thereby avoiding unbounded priority inversion.

Although priority inheritance semaphores solve the unbounded priority inversion problem, the inversion could still be relatively large if we allow a lower priority thread to execute until completion. The problem is more severe in our design than in traditional CORDS protocol graphs since we use the dispatcher thread in addition to shepherd threads. We solve this problem by having the dispatcher thread voluntarily give up its exclusive access to the protection domain if a higher priority thread is waiting to enter. This is represented in figure 3 by the "yield CPU to higher priority thread" statement prior to executing each event handler. This results in the worst case duration of a priority inversion being the maximum across all handler execution times.

The implementation of this yield operation involves determining if a higher priority thread is waiting to enter. This is done by exploiting knowledge about priority inheritance. Specifically, since priority inheritance raises the priority of the active thread, there must be a higher priority thread waiting if the priority of the current thread is higher than its original path priority. Unfortunately, the Posix standard does not include an operation that returns the current priority, so this information is obtained directly by reading a field in the `pthread` data structure.

Finally, note that the maximum execution time of each event handling operation is bounded. These bounds can be calculated since the maximum sizes of the EHQ and other data structures are determined by the set of micro-protocols included in each configuration and application characteristics such as the message transmission rate.

3.4 Estimating Execution Time

Providing real-time execution guarantees naturally requires knowledge about the execution times of different services competing for the CPU. The configurability of the event-driven model would appear to complicate this issue since the number of different configurations of a given service can be large and each configuration has its own execution time. However, in reality, relatively simple calculations based on the execution time of each event handler can be used to determine the execution time of any valid configuration of micro-protocols. The approach for doing this is outlined in subsequent paragraphs. We assume that a composite protocol is inactive until it either receives a message (or a call) from protocols below or above it in the protocol graph, or until a timer event occurs.

First, consider messages from other protocols. Arrival of such a message causes predefined events MsgFromApp and MsgFromNet to be raised. Given a specified set of micro-protocols, it is possible to determine which event handlers are executed as a result of these events, and similarly, which events these handlers raise. This process is continued until no more events are raised. We then sum up the execution times of the resulting set of event handlers to obtain the time required for processing a message.

Next, consider estimating the execution time for each occurrence of a timer event. The specifics here depend on how timers are used. In many distributed protocols, timers are used mostly to implement timeouts that cause, for example, retransmission of a message. In these cases, the timer events are directly related to message processing and the CPU time required can simply be added to the CPU time needed for message processing. In time-driven systems, however, timers are used to initiate execution at certain moments of time independent of messages or calls from outside the system. In these cases, the CPU time required for the timer event can be calculated the same way as for message processing, i.e., by determining the set of event handlers to be executed and summing up the total CPU requirement.

This approach does have restrictions, however. In particular, there must be no cycles in the chain of events raised by event handlers. That is, if handler H is bound to event A, then A must not be raised by H nor by any other handler that is executed as a result of events that H directly or indirectly raises. If this is not the case, the above calculation—and potentially system execution—is non-terminating.

4 Implementing Customizable RTD Channels

This section describes how RTD channels are implemented using CactusRT. The basic structure is presented first, followed by descriptions of the various micro-protocols that implement message ordering and reliability properties, respectively. Next, we describe how real-time properties are enforced. Finally, configuration issues are discussed.

4.1 Basic Structure

In our design, each RTD channel is implemented as a composite protocol consisting of selected micro-protocols and uses system resources allocated as CORDS paths (figure 4). Resource allocation has been divided into the *channel control module* (CCM), which is specific to channels, and the *admission control module* (ACM), which manages resources across multiple types of services. The CCM provides an API for an application to request channel creation. It passes information about the channel's traffic model and its CPU requirements to the ACM module. The ACM is a process that maintains information about available resources, and based on this information, either grants or denies requested resources. If the request is granted, the ACM also assigns path priorities. ACMs on different sites interact to ensure end-to-end guarantees.

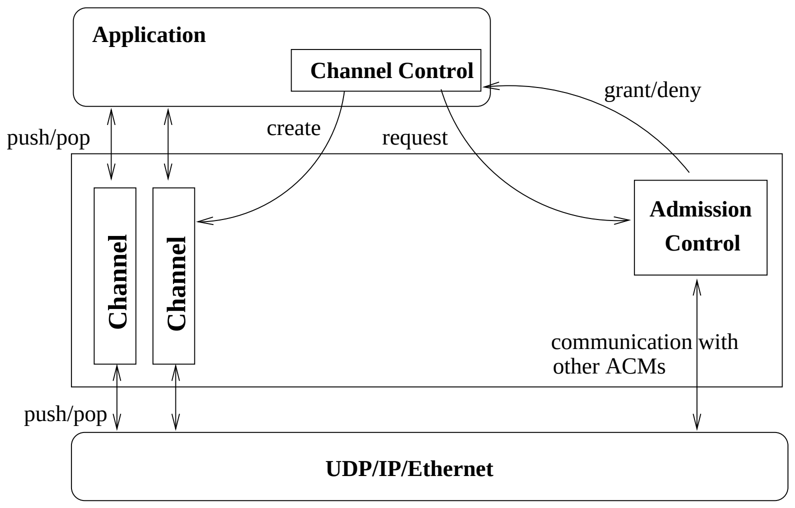

Figure 4: System components and component interactions

The core micro-protocol on which all others build is BasicChannel, which implements the abstraction of a simple channel with no reliability or ordering guarantees other than those provided by the underlying network and OS. This functionality can be augmented using different retransmission and ordering micro-protocols.

A number of different events are used in the composite protocol. Two are the system events MsgFromNet and MsgFromApp, which are raised by the framework when a message arrives from the network or application, respectively. There are also a number of user-defined events, which are raised by micro-protocols to signal completion of a task or otherwise communicate with other micro-protocols. Finally, timer events are used to implement message retransmissions.

The BasicChannel micro-protocol is shown in figure 5. This micro-protocol implements the basic function of passing a real-time message through the channel composite protocol, either from the application

```
micro-protocol BasicChannel(DropLate:bool) { /* DropLate: drop msgs that will miss deadlines? */
    procedure setReadyBits(msg,mp_id,dir) {
        msg.ready[mp_id] = true;
        if (msg.ready true for each micro-protocol)
            if (dir == UP) raise(RtMsgToApp,msg);
            else { raise(RtMsgToNet,msg,SYNC); push(msg); raise(RtMsgSentToNet,msg); }
    }
    handler HandleNetMsg(msg){
        if (msg.type == DATA) { msg.ready = DeliverOK;
            if (msg.realtime) raise(RtDataMsgFromNet,msg);
            else raise(NonRtDataMsgFromNet,msg);
            setReadyBits(msg,BasicChannel,UP); }
        else raise(MetaMsgFromNet,msg);
    }
    handler DeliverRtMsg(msg){
        if (msg passed deadline and DropLate) stopEvent(); /*drop late message */
        pop(msg); raise(RtMsgDeliveredToApp,msg);
    }
    handler HandleApplMsg(msg){ msg.ready = SendOK; setReadyBits(msg,BasicChannel,DOWN); }

    initial { SendOK[BasicChannel] = false; DeliverOK[BasicChannel] = false;
        bind(MsgFromNet,HandleNetMsg,5); bind(RtMsgToApp,DeliverRtMsg,2);
        bind(MsgFromApp,HandleApplMsg,2); }
}
```

Figure 5: BasicChannel micro-protocol pseudo code

to a lower-level protocol or vice versa. Several aspects of this micro-protocol are worth highlighting. One is how interactions with other micro-protocols in the composite protocol are handled, and in particular, how micro-protocols can influence passage of a message. This is done primarily by using specialized events such as RtDataMsgFromNet that are relevant only for the micro-protocols concerned with this type of message.

A second issue is determining when all micro-protocols are ready for a given message to exit the composite protocol. In our design, the global vectors SendOK and DeliverOK indicate whether a micro-protocol might potentially wish to delay a message. A matching vector in each message is then used to record whether a micro-protocol agrees that the message can exit. The SetReadyBits procedure is invoked by a micro-protocol for this purpose; once the last micro-protocol has given its approval, this procedure initiates the transmission of the message either up or down the protocol graph.

Finally, note that any missing event-handling arguments take on their default values. Of importance here is that when the urgency parameter is omitted, the event handlers inherit the current urgency. In this case, we assume the framework sets the urgency of the events MsgFromNet and MsgFromApp based on message deadlines. The default value for the mode parameter of **raise** is ASYNC.

4.2 Ordering Micro-protocols

Ordering micro-protocols impose relative ordering constraints on the message delivery from the channel to the target application. If no ordering micro-protocols are included, the delivery order is arbitrary. The current suite of ordering micro-protocols includes:

- FifoOrder. The sender of a message includes a sequence number in the message; the receiver delivers the messages from each sender in the sequence number order.
- CausalOrder. The sender of a message m_i includes with the message a causality vector that indicates the sequence numbers of the latest messages delivered from each of the channel sources before message m_i was sent. The receiver only delivers m_i when all messages that precede it based on the causality vector have been delivered.
- TotalTimeStamp. Uses timestamps from synchronized clocks to order message delivery at the receiver. If two messages from different senders have the same timestamp, the tie is broken by their unique sender identifiers. Message delivery is delayed until it can be guaranteed that there are no messages in transit with earlier timestamps.
- TotalSequencer. One of the receivers acts as a coordinator to order messages. Each message is sent to the coordinator, which assigns a global sequence number and then retransmits to other receivers.

FifoOrder, TotalOrder, and TotalSequencer delay messages that arrive out of order until all their predecessors in the given order have been delivered. This type of ordering may not always be satisfactory, however. First, all channels are not completely reliable ($P_R < 1.0$) and imposing such strict ordering for such channels could result in a deadlock where the message delivery is completely stopped to wait for a missing message. Second, in a real-time channel, one message that is delayed to the point of missing its deadline might cause all of its successors in the chosen order to also miss their deadlines.

To address these issues, we introduce an additional micro-protocol ForcedDelivery. If included in a configuration, this micro-protocol will deliver a message out of order if its deadline would otherwise be missed. ForcedDelivery can be used together with any of FifoOrder, CausalOrder, or TotalSequencer. TotalTimeStamp automatically provides a lossy order in the case some messages are lost or delayed since it only orders those messages that have been received by the time ordering is done.

4.3 Reliability Micro-Protocols

As indicated in section 2.2, reliability properties can be implemented using different types of redundancy, while different variants can be based on whether the message is retransmitted a fixed maximum number of times or until it is received. These two techniques correspond to the reliability variants of bounded reliability and absolute reliability, respectively, and are implemented by two retransmission micro-protocols, RetransReliable and PosAckReliable. The first transmits a message a specified number of times with small fixed intervals between transmissions. The second uses positive acknowledgments, and is thus more applicable for transmitting non-real-time messages or messages for which the deadline has already been missed.

The choice of which reliability micro-protocols to use and how they should be parameterized depends on the desired reliability P_R and timeliness P_D of the channel, as well as the characteristics of the underlying system. The reliability micro-protocols are only required if the failure probability of the underlying system—that is, the probability that it fails to deliver a message to the destination—is too high to satisfy the desired reliability and timeliness. This failure probability is used to calculate the number of retransmissions required by the RetransReliable micro-protocol. An issue that complicates the calculation is that failures of the underlying system are typically correlated. For example, if the underlying network suffers from a transient failure such as congestion, the probability of a retransmitted message being lost can be very high. However, the failure probability of such a retransmission typically decreases with time, so the probability of success can be increased by increasing the interval between retransmissions. Thus, both the number of retransmissions and the retransmission frequency for RetransReliable must be determined based on the channel requirements and the characteristics of the underlying system.

The RetransReliable micro-protocol is typically used to increase the reliability of the channel to the point the timeliness requirement P_D of the channel can be satisfied. If the reliability requirement P_R of the channel is higher than its timeliness requirement, the PosAckReliable micro-protocol is typically used to increase the reliability of the transmission further. PosAckReliable operates on a pool of messages that have not yet been acknowledged and periodically retransmits a fixed size batch of these messages.

Given such an approach, the order in which the messages are retransmitted may be important. For messages that can still meet their deadlines, the retransmission order is obviously earliest deadline first. However, for messages that have or will miss their deadlines, different policies are useful in different situations. These types of policies can be defined using a technique similar to the soft real-time *value functions* that determine the value of completing a soft real-time task after the deadline [JLT85]. In this case, a value function represents the value of delivering a message after its deadline. Typical alternatives include:

- Zero. Delivering a message after its deadline does not have any value to the application (dotted line in figure 6). Thus, a message that misses its deadline can be dropped.
- Constant. Delivering a message after its deadline has a constant value to the application, but the value is smaller than if the message had been delivered on time (dashed line in the figure). Thus, messages that have missed their deadlines can be retransmitted in any order, such as FIFO.
- Diminishing. The value of the message being delivered diminishes as time passes (solid line in the figure). Thus, depending on how quickly the value diminishes, it may be better to retransmit messages that will miss their deadlines in LIFO order rather than FIFO order.

Other options are possible, of course. In our design, the chosen value function is provided as a parameter to the PosAckReliable micro-protocol.

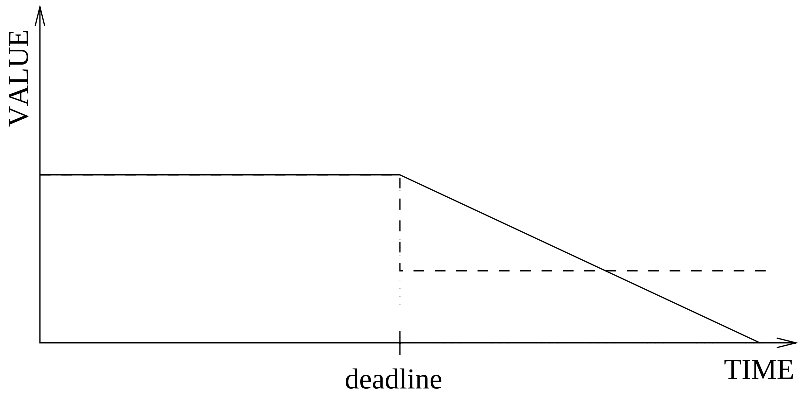

Figure 6: Message retransmission value functions.

The two reliability micro-protocols have different implications on CPU usage and message deadline guarantees. For a channel that uses RetransReliable, the CPU time required to send and receive retransmissions must be considered when doing CPU allocation. In addition, the time from the first transmission to the last retransmission must be factored into the deadline guarantee.

A channel that uses PosAckReliable is more complex, both with respect to calculating real-time guarantees for messages in the channel and determining how it affects other channels and other real-time services in the system. The main reason for this complexity is that the set of messages to be retransmitted may grow unpredictably, potentially until buffer space is exhausted. Thus, the CPU time required to retransmit these messages might similarly grow without special measures. The current design of PosAckReliable bounds the CPU time by executing the retransmission event-handler periodically at fixed intervals and having it transmit only a fixed number of messages each time it is activated.

Although this design ensures that the CPU utilization of a channel is always bounded, it has the potentially negative effect of giving priority to message retransmissions in channels with higher CORDS path priorities relative to processing in channels with lower priorities. Therefore, we are exploring an alternative design that assigns a lower priority to message retransmission relative to timely transmission in other channels. This can be accomplished by having a channel use more than one CORDS path in its implementation. In particular, a second lower priority CORDS path is used to retransmit messages, thereby ensuring that retransmissions are only done when there is no processing required in other real-time channels.

4.4 Implementing real-time properties

Real-time properties are different from ordering and reliability properties in the sense that they cannot simply be implemented as new micro-protocols. Rather, they must be realized by controlling the execution of the entire system, including the channel composite protocols and any other components competing for resources. The main aspects of this control are admission control, which determines which tasks are allowed to execute, and scheduling, which determines the order in which the CPU is allocated to different tasks.

The details of admission control are different for each real-time service type. The following discussion is specific to the channel abstraction, although many of the same principles apply to other service types. In this paper, we only address allocating the CPU, memory, and network bandwidth to different RTD channels.

4.4.1 Network assumptions

The initial version of the channel service does not implement network scheduling, but rather relies on the underlying Ethernet to provide relatively predictable message transmission times given a light enough network load. Despite this implementation restriction, the assumptions about the network are general enough that the system could be easily ported to other types of networks. In our assumptions, we have to acknowledge the realities of different network types and the restrictions they place on predictability. Generally, a network such as the Ethernet is not considered predictable enough for hard real-time systems, while networks such as the Internet or wireless networks are even worse. However, given our probabilistic approach to QoS guarantees, we aim at providing the best possible guarantees given the available network architecture. If these guarantees are not satisfactory for an application, another network architecture must then be chosen for the system.

The foundation of our approach is the probability distribution function (pdf) of the message delivery time in the given network architecture. Typically, such pdfs have an exponential distribution such as illustrated in figure 7. Note that the distribution often depends not only on the network architecture, but also on the system load. In the figure, the solid line represents the pdf when the system is lightly loaded and the dashed line one when the system is heavily loaded. This type of behavior has been documented for Ethernet networks [BMK88].

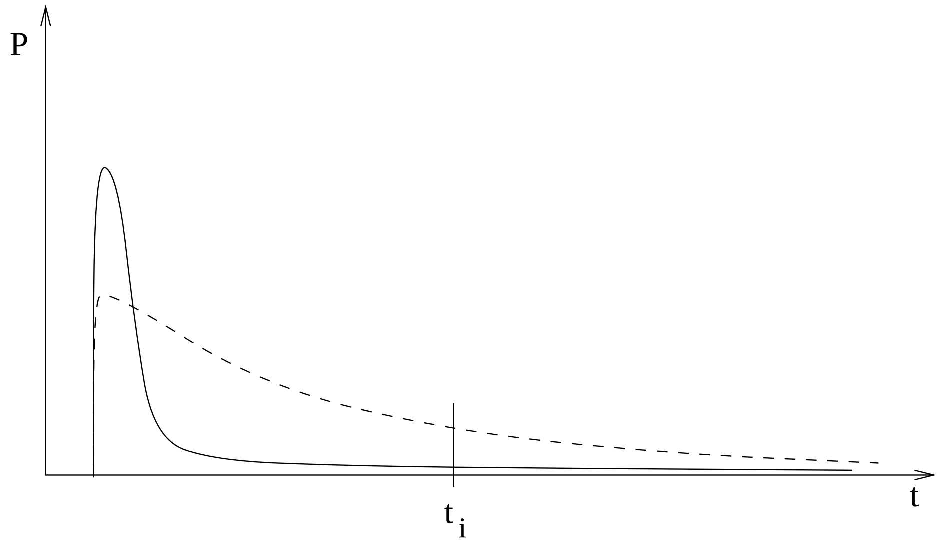

Figure 7: Message transmission time pdf

Given the pdf of the system and the current load, it is possible to determine for any message transmission time t_i the probability $P(t_i)$ that an arbitrary message will reach its destination before t_i. This probability is simply the area under the pdf from 0 to t_i. Thus, any desired deadline probability can be obtained by choosing t_i appropriately.

As noted above, the system is more predictable the lighter the load, so our design limits use of the network to a fraction of the total bandwidth. This approach limits the worst case load and thus, the worst case pdf for the network. By using this worst case pdf for admission control, channels are always able to match or exceed the expected timeliness.

Given the pdf of message transmission time and a controlled maximum network load, we can calculate the maximum network transmission time T_n for any given probability level P_n such that the probability of a message reaching its destination in less than or equal to T_n is P_n. The probability P_n naturally affects the deadline probability of the channel. For example, in the simple case where no retransmissions are used, P_D is always less or equal to P_n. This follows because even if the processing time of the message at the sender and the receiver meet their respective deadlines, the network transmission will only meet its deadline T_n with probability P_n.

4.4.2 Admission control

Admission control is divided into a *capability test* that determines if enough resources exist for the new channel, and a *schedulability test* that determines whether a feasible schedule that allows messages to meet their deadlines can be constructed. A new channel is established only if both tests are satisfied. Before these tests can be developed, however, the appropriate traffic models must be defined to characterize assumptions made about application message traffic.

Traffic models. Different types of real-time applications can have different communication patterns, so the system is designed to support different traffic models. Currently, however, we are concentrating on performing admission control using the (σ, ρ) model from [WKZL96], where σ is the burstiness factor and ρ is the average traffic rate. With this model, the size of the traffic backlog at a server never exceeds σ given that the server works at rate ρ [Cru91]. The server in our case is a channel composite protocol, ρ is the message rate R (messages/second), and σ is the burst size B (number of messages). Other traffic models to be explored include the H-BIND model [Kni96], which allows statistical guarantees to be made concerning timeliness of communication.

While a necessary starting point, this model only describes the traffic originating from application processes sending on a given channel. The traffic that a channel composite protocol actually receives from the network can be very different, however, because of variations in network transmission delay and extra messages generated by the composite protocol, such as acknowledgments and retransmissions. These factors mean that the model must be modified for the traffic received from the network. For example, if the burstiness and rate at the sending site are B and R, and the maximum network delay is T_n, then the burstiness at the receiving site will be B + $T_n \cdot$ R [Cru91]. Similarly, retransmissions increase both the message rate and the burst size. Finally, for multisource channels (including GM channels), the traffic model at the receiving site is the aggregation of the traffic models of the source sites.

Capability Tests. The three primary resources to be considered are CPU time, memory space, and network bandwidth. For the CPU capability test on a given site, we assume that a fixed fraction of the total CPU time is allocated for composite protocols, with the remainder used by the OS and the application. Given a request to create a channel C, then, the test is performed in two steps. First, the CPU usage for the composite protocol implementing C is calculated as outlined in section 3.4 based on the application and network message rates from the traffic model and the message generation behavior of the composite protocol. Second, this value is added to the sum of the usage values for the currently existing channels to determine if adding C would exceed the CPU allocation. If it would, then the request is denied; otherwise, the request passes the capability test and can continue to the schedulability test. The network bandwidth test is very similar. In this case, the bandwidth requirements for each channel is calculated based on the message rate, the maximum message size, and information about any extra messages generated by the composite protocol.

For the memory capability test, only the memory space required for storing messages needs to be considered since the size of static data structures can easily be estimated. Calculating this value requires taking two specific items into account. The first is the schedulability test (below), which dictates how long a message may have to wait until it can be transmitted. The other is the maximum retransmission time, which determines how long a message must be kept for possible retransmissions before its storage can be reclaimed. In the worst case—which occurs when the specified reliability requirement is 1.0—a message may have to be kept forever, implying that any fixed amount of memory may be insufficient. However, a probabilistic estimate of expected memory usage can be calculated even in this case given the estimated failure rate of the network and a desired level of assurance that the composite protocol will not exhaust its memory.

Schedulability Test. When a channel creation request is made, a schedulability test is needed to determine if execution deadlines can be met even if the CPU, bandwidth, and memory capability tests indicate sufficient resources. For simplicity, we only consider schedulability tests for CPU time; approaches to network scheduling can be found elsewhere (e.g., [MMM96]).

An application creating a new channel specifies a message deadline d, i.e., a message sent into the channel at time t must be delivered to all of its destinations by time $t + d$. For the schedulability test, we consider this deadline the sum of three values, $d = d_s + T_n + d_r$, where d_s is the maximum allowed delay on the sending site for the message to traverse the protocol stack down to the network, d_r is the corresponding time at the receiving site, and T_n is the maximum network transmission time as given in section 4.4.1. We assume that T_n is fixed, so the schedulability test derives the values of d_s and d_r. This is done using the *response times* for a message. In particular, let r_s denote the worst case response time for a message to traverse the composite protocol at the sending site and r_r the corresponding time at the receiving site. Then, the schedulability test determines if d_s and d_r exist such that $d_s + T_n + d_r \leq d$, $r_s \leq d_s \leq 1/R$, and $r_r \leq d_r \leq 1/R$, where R is the message rate. If not, the channel creation request is denied. Note that the delays at the sender and receiver can be adjusted to accommodate sites with different loads as long as the sum of delays is less or equal to the deadline.

The worst case response times r_s and r_r are calculated using the algorithm described in [LWF94]. First, the existing paths on a site are sorted based on their priority, where path P_i has the i^{th} highest priority. Given this set of paths, the CPU response time r_k for any channel P_k is then calculated using the formula:

$$r_k = \frac{\sum_{i=1}^{k} B_i \cdot E_i + \Delta}{1 - \sum_{i=1}^{k} R_i \cdot E_i} \tag{1}$$

Here, B_i, R_i, and E_i are the burst size, traffic rate, and maximum CPU time for processing a message in path P_i, respectively. The factor Δ is the maximum priority inversion time, which occurs in CORDS when a lower priority thread continues to be executed even though a higher priority thread becomes ready to execute. Thus, $\Delta = \max_{j>p} e_j$, where e_j is the maximum execution time of any single event handler in path P_j. Note that this formula applies only to paths with a single traffic model for arriving messages; the formula becomes more complex when a path has more than one traffic model.

This formula is used in conjunction with the D_order procedure [KSF94] to determine a priority for the new path that is allocated as a result of the channel creation request. In essence, this step calculates the highest possible priority for the path that still allows all deadlines to be met. First, the new path is assigned the highest priority and the response time of all other paths is calculated using the above formula. If their deadlines can still be satisfied, then the priority of the new path is taken to be the current value. Otherwise, the priority of the new path is decreased and the process repeated. In addition to establishing the priority, this procedure also determines the minimum worst case response time.

After calculating the worst-case response times r_s and r_r, d_s and d_r are assigned to the new path so that $d_s + T_n + d_r < d$, $r_s \leq d_s \leq 1/R$, and $r_r \leq d_r \leq 1/R$ are satisfied. As noted above, if no such assignment exists, then the path cannot be created and thus, the corresponding channel creation request is denied. Otherwise, various heuristics can be used to choose reasonable values for d_s and d_r, such as setting them in proportion to r_s and r_r. The set of micro-protocols in the composite protocol affects the decision as well. For example, if total or causal ordering are included, then a message may be delayed awaiting the arrival of other messages. In this case, d_r should be larger than d_s.

4.5 Possible configurations

An RTD channel can be customized across a wide range of QoS attributes, which allows it to satisfy applications with diverse real-time and dependability requirements. The customization of a channel typically starts by choosing the channel shape. As indicated in section 2, eight different shapes are available. After the shape is chosen, we can further define the behavior of the channel by selecting the desired ordering, reliability, and timeliness guarantees. For ordering, this is done by selecting the appropriate micro-protocol from among FifoOrder, CausalOrder, TotalTimeStamp, and TotalSequencer. Ordering behavior can be specified further by choosing ForcedDelivery if timeliness of the message delivery is more important than ordering. No ordering micro-protocol is required if the channel is not required to enforce any ordering. Given these options, then, we can configure eight different types of GM channels, for example. Note, however, that all ordering properties are not feasible for all channel shapes. For example, total ordering only applies to channels with multiple senders and receivers, while causal ordering only applies to GM channels.

The desired reliability and timeliness guarantees are set by specifying the corresponding probabilities P_R and P_D and the desired message deadline d. The probabilities, together with our assumptions about the inherent reliability and timeliness of the underlying network, determine which of the reliable communication micro-protocols RetransReliable and PosAckReliable need to be included in the configuration and the values of the arguments that further define their behavior. For example, for RetransReliable, we have to determine the number of retransmissions and the delay between each retransmission. Similarly, for PosAckReliable, we specify the value function that determines the retransmission order of messages. In the initial version of the RTD channel, these arguments are determined manually. A later version will include a configuration support tool that automates this task.

In addition to the micro-protocols discussed in this paper, we are designing additional micro-protocols that control other aspects of system behavior. These include jitter control, which reduces jitter in the stream of messages delivered, and flow control, which enforces sender compliance with the given traffic model.

5 Related Work

A number of systems support configurability and customization in distributed systems, including Adaptive [SBS93], Horus [RBM96], and the *x*-kernel [HP91]. However, only Adaptive and the configurable control system in [SVK93] address issues related to real time. Adaptive introduces an approach to building protocols that employs a collection of reusable 'building-block' protocol mechanisms that can be composed automatically based on functional specifications. The objects are tightly coupled in the sense that interactions between objects are fixed *a priori*. Furthermore, although Adaptive targets multimedia applications, its runtime system appears to be designed to maximize performance rather than ensuring deadlines.

A reconfigurable real-time software system is introduced in [SVK93]. The target application domain is sensor-based control systems, rather than real-time communication as is the case here. The port-based object model used in this system is suitable for combining existing software components, but lacks the degree of flexibility and fine-grain control found in the CactusRT approach. As such, it would be difficult to use this model to construct the same type of configurable services.

A large number of real-time systems have been designed and implemented, including Chimera [SVK93], Delta-4 [Pow91], HARTS [KS91], Mars [KDK$^+$89], MK [Rey95, TMR96], RT-Mach [TNR90], and TTP [KG94]. While suitable for their target application areas, the lack of support for configurability and customization in such systems typically limits their applicability to new areas. Two exceptions are [ASJS96] and [TMR96], which have adopted principles from the *x*-kernel to add coarse-grain modularity and a limited degree of configurability to certain real-time communication services.

Real-time channel abstractions similar to RTD channels have been developed elsewhere as well. In some cases, these channels address real-time communication at the network level; for example, a type of real-time channel that is established across multiple point-to-point network links is introduced in [KS91, KSF94]. Similarly, Tenet provides real-time channels over heterogeneous inter-networks, as well as a real-time network layer protocol RTIP and two real-time transport layer protocols RMTP and CMTP [BFM$^+$96]. The latest Tenet suite also provides multicast channels and resource sharing between related channels. An atomic real-time multicast protocol ensuring total ordering of messages is introduced in [ASJS96]. This protocol uses a logical token ring, and integrates multicast and membership services, but without explicitly introducing a real-time channel abstraction. None of these are customizable to the same degree as the RTD channels introduced here.

6 Conclusions

This paper addresses the problem of providing customized real-time dependable communication services for different types of applications. We presented RTD channels, an example of how such services can be implemented using an event-driven interaction model that allows different abstract service properties and their variants to be implemented as independent modules called micro-protocols. A set of micro-protocols is then selected based on the desired properties and configured together into a composite protocol implementing a custom version of the service. Specifically, we first introduced the design of RTD channels, which support multiple application interaction patterns and numerous combinations of properties. We then described the CactusRT interaction model and outlined the implementation of a prototype system based on the *x*-kernel and the path abstraction in CORDS. Finally, we described how RTD channels can be implemented using this model to realize a mechanism with a high level of customization.

RTD channels are being constructed on a cluster of Pentium PCs connected by a 10Mbit/sec Ethernet and running the OpenGroup/RI MK 7.3 Mach operating system. In addition to completing the implementation, future work will include refinement of admission control and scheduling for the event-driven model, and experimentation with other real-time services using CactusRT.

References

[AS95] F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 36–43, May 1995.

[ASJS96] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 250–259, Jun 1996.

[BFM$^+$96] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang. The Tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transactions on Networking*, 4(1), Feb 1996.

[BMK88] D. Boggs, J. Mogul, and C. Kent. Measured capacity of an Ethernet: Myths and reality. In *Proceedings of SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 222–234, Aug 1988.

[BS95] N. Bhatti and R. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of SIGCOMM '95*, pages 138–150, Cambridge, MA, Aug 1995.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[CASD95] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[Cru91] R. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, Jan 1991.

[GBB$^+$95] D. Guedes, D. Bakken, N. Bhatti, M. Hiltunen, and R. Schlichting. A customized communication subsystem for FT-Linda. In *Proceedings of the 13th Brazilian Symposium on Computer Networks*, pages 319–338, May 1995.

[Her94] A. Herbert. An ANSA overview. *IEEE Network*, 8(1), Jan 1994.

[HP91] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HS95] M. Hiltunen and R. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 288–295, Vancouver, BC, Canada, May 1995.

[HS98] M. Hiltunen and R. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[JLT85] D. Jensen, D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the Real-Time Systems Symposium*, pages 112–122, San Diego, CA, Dec 1985.

[KDK$^+$89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[KG94] H. Kopetz and G. Grunsteidl. TTP - A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994.

[KKG$^+$90] H. Kopetz, H. Kantz, G. Gruensteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in Mars. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing*, pages 466–473, Jun 1990.

[Kni96] E. Knightly. H-BIND: A new approach to providing statistical performance guarantees to VBR traffic. In *Proceedings of IEEE INFOCOM '96*, pages 1091–1099, San Francisco, CA, Mar 1996.

[KS91] D. Kandlur and K. Shin. Design of a communication subsystem for HARTS. Technical Report CSE-TR-109-91, University of Michigan, Oct 1991.

[KSF94] D. Kandlur, K. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 1044–1056, Oct 1994.

[LWF94] J. Liebeherr, D. Wrege, and D. Ferrari. Exact admission control for networks with bounded delay services. Technical Report TR-94-033, International Computer Science Institute, University of California at Berkeley, Berkeley, CA, Aug 1994.

[MG95] V. Murty and V. Garg. An algorithm for guaranteeing synchronous ordering of messages. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 208–214, Phoenix, AZ, Apr 1995.

[MHCN96] A. Mauthe, D. Hutchison, G. Coulson, and S. Namuye. Multimedia group communications: towards new services. *Distributed Systems Engineering*, 3(3):197–210, Sep 1996.

[MMM96] C. Martel, W. Moh, and T.-S. Moh. Dynamic prioritized conflict resolution on multiple access broadcast networks. *IEEE Transactions on Computers*, 45(9):1074–1079, Sep 1996.

[MMSA$^+$96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr 1996.

[MPS89] S. Mishra, L. Peterson, and R. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, Oct 1989.

[OMG98] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 2.2)*, Feb 1998.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[Pow91] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Computing*. Springer-Verlag, 1991.

[RBM96] R. van Renesse, K. Birman, and S Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[Rey95] F. Reynolds. The OSF real-time micro-kernel. Technical report, OSF Research Institute, 1995.

[SBS93] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[SRL90] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, 1990.

[SVK93] D. Stewart, R. Volpe, and P. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. Technical Report CMU-RI-TR-93-11, Carnegie Mellon University, July 1993.

[TMR96] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Feb 1996.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Toward a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, pages 73–82, Oct 1990.

[Ver94] P. Verissimo. Ordering and timeliness requirements of dependable real-time programs. In *Real-Time Systems*, pages 105–128. Kluwer Academic Publishers, Boston, 1994.

[WKZL96] D. Wrege, E. Knightly, H. Zhang, and J. Liebeherr. Deterministic delay bounds for VBR video in packet-switching networks: Fundamental limits and practical tradeoffs. *IEEE/ACM Transactions on Networking*, pages 352–362, Jun 1996.