Supporting Configurable Real-Time Communication Services

Xiaonan Han   Matti A. Hiltunen   Richard D. Schlichting

TR 97-10

Supporting Configurable Real-Time Communication Services[1]

Xiaonan Han Matti A. Hiltunen Richard D. Schlichting

TR 97-10

Abstract

Constructing communication services that provide real-time guarantees is important for many applications built on distributed systems. While a variety of such services have been designed and implemented, most are targeted for specific applications and are correspondingly difficult to adapt to differing requirements in other areas. This paper presents an approach to building configurable and customized versions of real-time communication services based on 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 a runtime system providing real-time guarantees. The programming model is presented, together with an implementation design based on the *x*-kernel model for building network subsystems and the OSF/RI MK 7.2 operating system. The design of a highly-configurable real-time channel abstraction built using this approach is also given. Prototype implementations of the runtime system and channel abstraction are currently underway.

May 1, 1997

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.

1 Introduction

Real-time guarantees for computing services such as network communication and task execution are important in many application areas. For example, safety critical systems usually require hard guarantees on system response time to avoid catastrophe, while multimedia applications behave optimally with at least soft guarantees. Numerous different types of real-time services and systems have been developed [ASJS96, BFM$^+$96, KDK$^+$89, KG94, Pow91, TNR90]. However, most such systems are targeted for specific applications, which makes them rigid and unable to accommodate the different requirements of a variety of application types. This is especially true for real-time communication services found in distributed systems, where applications often need guarantees related to such properties as reliable message transmission, atomic delivery, and consistent message ordering in addition to real-time guarantees.

This paper presents a new approach to constructing distributed real-time communication services that allows the service guarantees to be customized to the requirements of a wide array of different applications. The focus is on building flexible middleware services that use the functionality of an underlying real-time operating system and network to implement higher-level abstractions for distributed applications. The goal is to construct services that can provide not only real-time guarantees, but also reliability, message ordering, and other quality of service (QoS) guarantees optimized for each application and execution environment. This emphasis on integrating attributes and on providing a high degree of flexibility distinguishes the approach from other real-time and configurable systems that support a degree of customization [Her94, RBM96, SBS93, SVK93, TMR96].

Our approach is based on implementing different semantic properties and property variants of a service as separate fine-grain modules called *micro-protocols* that interact using an event-based interaction model. A service is constructed by selecting a set of micro-protocols implementing the desired properties and configuring them together into a *composite protocol* that is positioned between the application and the operating system. A runtime system within each composite protocol schedules execution of micro-protocols to ensure that real-time properties are satisfied, while an admission control module ensures that composite protocols are only created if the system has enough resources to satisfy its requirements. An implementation of the model is currently underway using the OSF/RI MK 7.2 real-time operating system [Rey95] on a cluster of Pentium PCs. An earlier version of the system has been used to develop highly-configurable versions of communication services without real-time constraints, including group RPC [HS95, BS95], membership [HS97], and atomic multicast [GBB$^+$95].

The rest of the paper is organized as follows. Section 2 describes our model for configurable real-time services and outlines an implementation of the model designed for OSF/RI MK 7.2 and the *x*-kernel [HP91]. We also outline the composite protocol runtime system, and explain how the execution time of a collection of micro-protocols can be estimated. In section 3, the model is used to construct a configurable communication channel abstraction supporting customization of real-time, reliability, and ordering properties. Section 4 outlines related work and contrasts our approach to others. Finally, section 5 offers some conclusions.

2 Implementing Configurable Real-Time Services

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.

Our specific design and implementation are oriented around the *x*-kernel model, in which the middleware is implemented as part of a graph of *protocols* (i.e., software modules) that form the network subsystem. However, the approach is independent of the *x*-kernel and can be implemented using any number of suitable substrates. For example, a prototype of the previous non-real-time version has been done using C++ on the Solaris OS [Hil96].

2.1 Event-Driven Model

Our approach is based on implementing different semantic properties and functional components of a real-time service as separate modules that interact using an event-driven execution model. The basic building block of this model is a *micro-protocol*, 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 that are executed when an *event* occurs. Events can be either user or system defined, and are used to signify changes of state potentially of interest to the micro-protocol, e.g., "message arrival from the network" (MsgFromNet). When an event is detected, all event handlers *registered* for that event are invoked; events can also be triggered explicitly by micro-protocols, with the same effect. Execution of event handlers is atomic with respect to concurrency, i.e., each handler is executed to completion without interruption.

Event registration, 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* = **register**(*event,handler,order,times*):

  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. The binding automatically expires when the handler has been executed *times* times or can remain in effect until explicitly canceled (*times* = PERM). **register** returns a handle to the binding.

- **deregister**(*bid*):

  Removes a binding previously established by **register**.

- **trigger**(*event,args,mode,urgency*):

  Causes handlers registered for *event* to be executed with *args* 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).

- **delayTrigger**(*event,args,urgency,delay,times*):

  Triggers *event* after *delay* time units have passed. A value of *times* greater than one triggers the event every *delay* units for the number of times specified, giving periodic execution. An event for which a **delayTrigger** operation has been executed is called a *timer event*.

- **stopEvent**():

  Cancels execution of the collection of event handlers scheduled by the currently executing event trigger.

Event handlers are expected to be simple and have bounded maximum execution times. This means, among other thing, that they may not include any infinite loops or use blocking synchronization like semaphore wait operations, delays, or sleeps. If a micro-protocol requires synchronization, the same effect can typically be achieved using event operations and one or more handlers. Note that the execution atomicity of event handlers eliminates the need to protect shared data structures using semaphores.

The event-driven model has a number of appealing features for real-time systems. First, event-handlers are short and simple, with predictable timing behavior. Thus, given the CPU time for each event handler and knowledge of which events are triggered by each handler and the framework, we can easily calculate the CPU time needed by the entire service for a specific situation (see section 2.3). Second, scheduling can be done at the handler level and each handler executes to completion before rescheduling. As a result, scheduling is simplified and long priority inversions [TMIM89] are avoided. Finally, atomic execution of event handlers minimizes the need for synchronization between handlers, and thus, reduces the well-known problems of synchronization, such as priority inversion.

2.2 Implementing the Model

Our prototype implementation of the real-time event-driven model is designed for the OSF/RI MK 7.2 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 thread priorities provided by the system to give services and tasks with tighter deadlines higher priorities. Note, however, that scheduling within a composite protocol still needs to be addressed to order the execution of event handlers associated with different events.

The basic design uses one CORDS path for each composite protocol implementing a real-time service. All threads associated with a single path have the same CORDS priority and thus, would be executed in arbitrary order by the MK OS scheduler. In many cases, this is not enough. For example, if two messages arrive from the network to a composite protocol 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 retransmissions.

To realize this scheduling of event 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 for which they are registered is triggered, and removed when they are 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.

In some cases, it is useful to have more than one path in a composite protocol, and hence, more than one EHQ and dispatcher. For example, consider a bidirectional real-time communication channel where the channel is used to send video in one direction and simple commands in the other. In this situation, the QoS attributes associated with message deadlines and jitter are very different for the two directions. These varying requirements can easily be implemented in this scheme by having separate paths within the composite protocol for each direction. Event handlers associated with the different traffic types within a composite protocol are assigned at configuration time to the paths corresponding to the traffic types. Note that although a composite protocol may have more than one path, the global data structures are still shared by all handlers. Note also that execution of handlers is still atomic despite potentially multiple dispatcher threads since CORDS has non-preemptive scheduling.

Figure 1 illustrates this design. In the figure, solid arrows represent threads of execution, specifically the dispatcher thread associated with each 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 EHQs as a result of trigger operations.

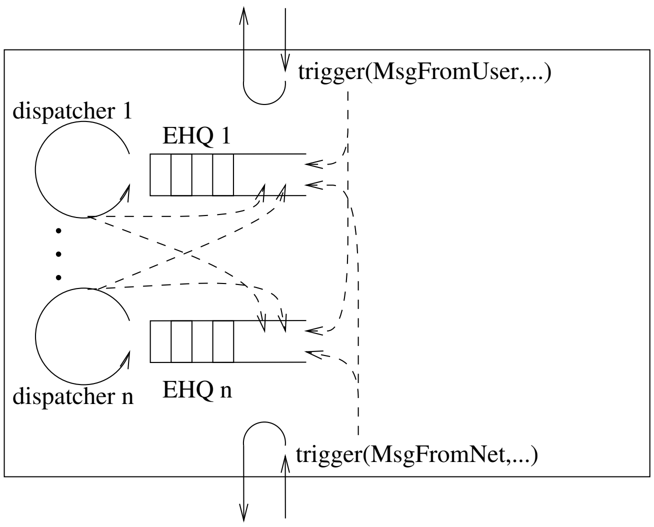

Figure 1: Framework

In addition to the EHQs, each composite protocol contains an *event/handler map* (EHM) that maintains information about which handlers are registered for each event. Handlers for a given event are stored in this structure according to their relative ordering requirements. The register and deregister operations update the EHM in the obvious way.

The implementation of the trigger operation depends on the mode supplied as an argument in the call (see Figure 2). A non-blocking trigger simply inserts the handlers registered for this event into the EHQ. A blocking trigger, however, is more complex since the handlers registered for the event must be executed prior to resuming the triggering handler. In our design, the thread executing the trigger 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 the triggering handler is resumed by the dispatcher executing a return operation. Note that more than one handler may be blocked concurrently at a trigger operation associated with a given 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 triggered handlers

```
trigger(event, mode, args) {
        handlers = EHM(event);
        addArgs(handlers, args); /* add args to handler info blocks */
        if (mode == ASYNC) insert(handlers);
        else { C_h = new counter(number of handlers);
                associate each handler with a reference to C_h;
                insert(handlers);
                call dispatcher(); }
}
dispatcher() {
        while(true) {
                yield(); P(S_n); h:HandlerInfo = head(EHQ);
                call h->func(h->args);
                if (h->C_h != NULL and --(h->C_h) == 0)    /* last SYNC triggered handler is done*/
                     return();
        }
}
insert(handlers) {
        insert handlers to path's EHQ ordered by urgency;
        S_n += number of handlers;
}
delayTrigger(event, args, delay, times) {
        handlers = EHM(event);
        addArgs(handlers, args);
        TimerRecord info; info.handlers = handlers;
        info.delay = delay; info.times = times;
        evSchedule(timeout,info,delay);
}
void timeout(TimerRecord info) {
        if (info.times == PERM or --info.times > 0)
                evSchedule(timeout,info,info.delay);
        set my thread priority to channel priority; yield();
        insert(info.handlers);
}
```

Figure 2: Event-handling pseudo code

is always greater or equal to the urgency of the handler that executed the trigger operation. Finally, the delayTrigger operation is implemented using the timer mechanisms provided by CORDS to realize the appropriate delay, as illustrated in Figure 2.

Due to the non-preemptive scheduling of CORDS, we also have to ensure that the dispatcher gives higher priority threads a chance to execute to avoid extended priority inversion. This is represented in Figure 2 by a call to a *yield* operation prior to execution of each event handler. This results in the worst case duration of a priority inversion being the maximum across all handler execution times. The details of yield are omitted for simplicity, but the operation involves releasing a lock that controls the execution of protocols in CORDS.

Several other issues must also be addressed. One is preventing the execution of dispatcher threads whose EHQs are empty to avoid needless consumption of CPU time. This is achieved using a semaphore S_n whose value corresponds to the number of handlers in the EHQ at any given time. Another is ensuring that execution of a high priority thread is not blocked by a low priority event handler if handlers from different paths are registered for the same event. This problem is avoided in our design by automatically assigning such an event handler to the higher priority path at configuration time. A third is implementing stopEvent so that canceling an event occurrence is an efficient operation. This is done by linking together handlers that are added to the EHQ as a result of a single trigger operation, which allows all the handlers associated with a given event occurrence to be removed easily if stopEvent is invoked.

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 message rate.

2.3 Estimating Execution Time

Managing the CPU to maintain real-time 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 triggered. 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 trigger. This process is continued until no more events are triggered. 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 within the system. 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 triggered by event handlers. That is, if handler H is registered for event A, then A must not be triggered by H nor by any other handler that is executed as a result of events that H directly or indirectly triggers. If this is not the case, the above calculation—and potentially system execution—is non-terminating.

2.4 Discussion

As noted in the introduction, the event-driven model presented above is based on an earlier version for constructing distributed services without real-time constraints [Bha96, HS93, Hil96]. The current version extends and refines the original model to accommodate the additional requirements that arise in real-time distributed systems. For example, in the original model handlers are executed logically in parallel with no relative priority ordering between them, which can lead to undesirable behavior such as unnecessary message retransmission [Bha96]. Furthermore, the logically concurrent model of handler execution has been modified to the atomic sequential execution model that requires less synchronization. Finally, the implementation of the model has been completely redesigned to allow real-time considerations to be addressed.

3 A Configurable Real-Time Channel

As an example, we now describe the implementation of a highly-configurable *real-time channel* using our approach. A channel is an abstraction for communicating between two or more application-level processes on different sites in a distributed system, and a real-time channel is one that provides timeliness guarantees. We first outline properties of real-time channels that can be used as the basis for a highly-configurable service, and then present a design using the event-driven model. An implementation of the design is underway.

3.1 Channel Shapes

Different types of 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. For the latter, 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 3.

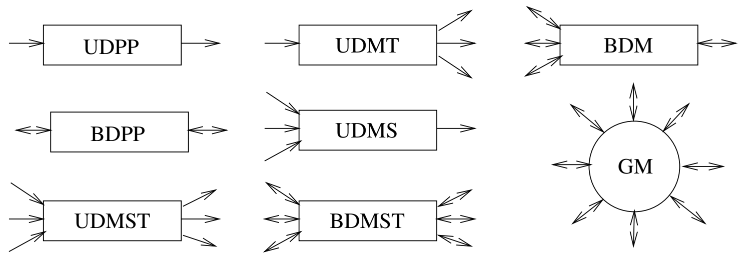

Figure 3: Channel shapes

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.

3.2 Channel Properties

A large number of properties can be defined for real-time channels, but for brevity, we describe only a representative set here. In particular, we consider *latency*—i.e., the maximum delivery delay, which affects whether a transmitted message meets a specified deadline or not, *reliability*—i.e., whether each message sent on a channel reaches its destination(s), and *ordering* properties between messages.

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].

In addition to the above, real-time services often have a specification defining the importance of the service meeting its constraints, usually specified using the phrases "hard real-time" or "soft real-time." We define a similar property for channels quantitatively using two probabilities, P_D and P_R. The first, called the *deadline probability*, is the required probability that a given message reaches its destination by the specified deadline. Various techniques (e.g., congestion control and retransmissions) can be used to implement the required P_D, although the maximum latency and the characteristics of the underlying system naturally constrain the range of feasible values. Similarly, the second probability, called the *reliability probability*, is the required probability that a given message sent on a channel eventually reaches its destination, either before or after its deadline. Note that the two metrics are related in the sense that establishing a given deadline probability necessarily fixes a minimum reliability level.

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.

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 latency 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].

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-target channels. Conversely, the need for certain channel properties can dictate the shape required for an application. 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.

3.3 Implementing Real-Time Channels

3.3.1 System Components

Figure 4 shows the major logical components in the system and their interactions. Each channel is implemented as a composite protocol consisting of selected micro-protocols and uses system resources allocated as CORDS paths. In our design, 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 translates channel arguments into general resource requirements. In particular, it translates the channel property requirements into a set of micro-protocols and paths that satisfies these requirements, and calculates the CPU requirements given the application traffic model. 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. A channel may also interact with the ACM after being created; for example, it may request additional resources or be instructed to release resources or terminate should a higher-priority service need resources. Note that the ACMs on different sites may interact, for example, to allocate the network bandwidth.

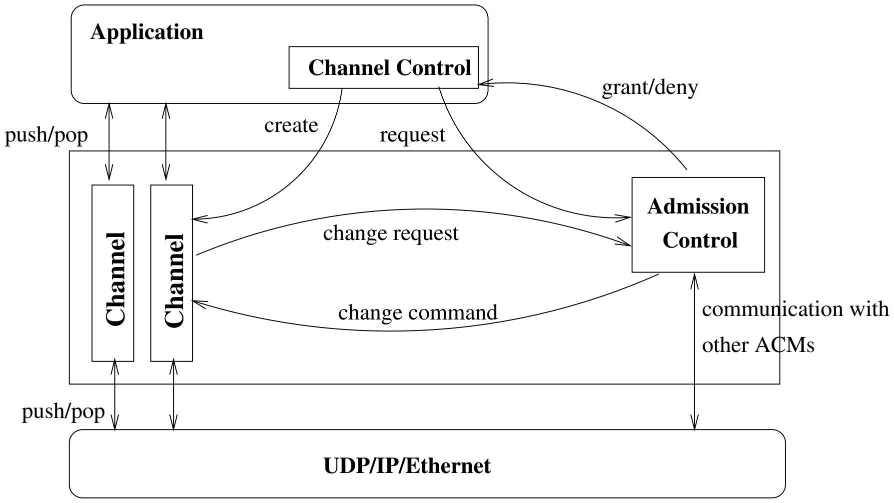

Figure 4: System components and component interactions

3.3.2 Micro-Protocol Suite

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. There are currently two retransmission micro-protocols. The first increases the probability of timely delivery by transmitting a message a specified number of times with small 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.

Any one of three ordering micro-protocols can also be configured into the composite protocol to enforce FIFO, causal, or total ordering, respectively. The ordering micro-protocols are designed so that a message will be delivered out of order if waiting any longer would result in missing the delivery deadline [AS95]. This forced unordered delivery is implemented as an event handler, so the service can easily be configured without this behavior if ordering is more important than the delivery deadline for a particular channel.

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

Finally, six timer events are used to implement message retransmissions and the forced out of order delivery mentioned above.

```
micro-protocol BasicChannel(Δ:real,DropLate:bool) /* Δ: estimated processing time of a message */ {

    event handler msg_from_net(msg){
        if (msg.type == DATA) trigger(DataMsgFromNet,msg);
        else trigger(MetaMsgFromNet,msg);
    }
    event handler data_msg_from_net(msg){
        msg.ready = DeliverOK;
        if (msg.type == RTDATA) trigger(RtDataMsgFromNet,msg);
        else trigger(NonRtDataMsgFromNet,msg);
    }
    event handler rt_data_msg_from_net(msg){
        if (msg.deadline < currentTime + Δ and DropLate) { stopEvent(); return(); } /* drop late message */
        else setReadyBits(msg,BasicChannel,UP);
    }
    event handler msg_from_app(msg){
        msg.ready = SendOK; setReadyBits(msg,BasicChannel,DOWN);
    }
    event handler rt_msg_to_net(msg){
        push(msg); trigger(RtMsgSentToNet,msg);
    }
    event handler rt_msg_to_app(msg){
        if (msg passed deadline and DropLate){ stopEvent(); return(); } /*drop late message */
        pop(msg);
        if (delivered[msg.source] < msg.id){
            delivered[msg.source] = msg.id; deliveredTime[msg.source] = currentTime; }
        trigger(RtMsgDeliveredToApp,msg);
    }
    procedure setReadyBits(msg,micro-protocol_id,direction){
        msg.ready[micro-protocol_id] = true;
        if (msg.ready true for each micro-protocol)
            if (direction == UP) trigger(RtMsgToApp,msg);
            else trigger(RtMsgToNet,msg);
    }
    initial {
        delivered = 0; SendOK[BasicChannel] = false; DeliverOK[BasicChannel] = false;
        register(MsgFromNet,msg_from_net,5); register(RtMsgToNet,rt_msg_to_net,5);
        register(MsgFromApp,msg_from_app,2); register(RtMsgToApp,rt_msg_to_app,2);
        register(RtDataMsgFromNet,rt_data_msg_from_net,2);
        register(DataMsgFromNet,data_msg_from_net,5); }
}
```

Figure 5: BasicChannel micro-protocol

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 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. Furthermore, the default value for the mode parameter of **trigger** is ASYNC, while the default value for the time parameter of **register** is PERM.

The given set of six micro-protocols can be configured to provide a large number of semantically different channels. Simply considering the different configurations of micro-protocols, we get 16 different combinations ranging from just the BasicChannel to one with both retransmission micro-protocols and one of three ordering micro-protocols. All these combinations are legal for GM channels. However, as mentioned above, all properties are not feasible for all channel shapes, so the number of combinations is less for other shapes; for example, there are five possible combinations for UDPP channels. Nonetheless, given the eight channel shapes that have been defined, dozens of different configuration are possible. The behavior of these channels can be customized further by specifying the desired message latency, P_D, P_R, and other arguments that further fine-tune their behavior. Thus, even this relatively small set of micro-protocols can be used to build channels with properties suitable for a variety of different applications.

3.4 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 different tasks get to use the CPU.

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 will undoubtedly apply to other service types. In this paper, we only address the allocation of memory and network bandwidth to different channels and the scheduling of CPU time. We do not consider network scheduling, but assume that by keeping the network load under control through network allocation, the Ethernet network will behave predictably [BMK88]. In particular, we assume that we know, for a given probability level P_n, the maximum network transmission time T_n such that the probability of a message reaching its destination in less than or equal to T_n is P_n.

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.

3.4.1 Traffic Model

Assume that the traffic on a channel is given by a function A, such that $A[\tau, \tau + t]$ denotes arrivals in the time interval $[\tau, \tau + t]$. Then, the traffic constraints of this channel can be given by a function A^*, such that $A[\tau, \tau + t] \leq A^*(t)$. The basis for our traffic model is the (σ, ρ) model from [WKZL96], where σ is the burstiness factor, ρ is the average traffic rate, and $A^*(t) = \sigma + \rho \cdot t$. 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). Thus, the message traffic constraint is $A^*(t) = B + R \cdot t$.

While a necessary starting point, this model only describes the traffic originating from the 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 $A_s^*(t) = B + R \cdot t$ at the sending site and the maximum network delay is T_n, then $A_r^*(t) = B' + R \cdot t$ at the receiving site, where $B' = B + T_n \cdot R$ [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.

3.4.2 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 2.3 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 is granted. The network bandwidth test is similar. In this case, the bandwidth requirements for each channel are calculated based on the message rate, the maximum message size, and information about any extra messages generated by the composite protocol.

Both the CPU and bandwidth tests are complicated somewhat by the retransmissions used to increase reliability of non-real-time messages or messages that have already missed their deadline. These micro-protocols are based on positive or negative acknowledgment schemes, which means that if a reliability of 1.0 is specified, the number of retransmissions cannot be bounded *a priori*. This problem is eliminated in our design by giving retransmissions a low priority so that they use the CPU only when no higher priority work is available. Additionally, a small fixed upper bound is placed on the number of messages transmitted each time the retransmission micro-protocol is activated. This ensures that retransmissions use at most a small fixed fraction of the CPU time and network bandwidth over any interval.

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 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.

3.4.3 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 tests indicate sufficient resources. In our case, a schedulability test is done on each path in the composite protocol implementing the channel, with the channel only being created if all paths pass the test. For simplicity, we only consider schedulability tests for CPU time; approaches to network scheduling can be found elsewhere, such as [MMM96].

For each message, the application specifies a deadline d. Note, however, that this deadline can be considered 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 composite protocol 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 above. 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 (see below). Then, the schedulability test determines whether values of d_s and d_r exist such that $r_s \leq d_s \leq 1/R$ and $r_r \leq d_r \leq 1/R$, where R is the message rate. If not, the channel creation request is denied.

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 event handler continues to be executed even though a higher priority path becomes available. 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 only one 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, e.g., to be 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 r_r.

4 Related Work

A number of systems support configurability and customization of communication services in distributed systems, including Adaptive [SBS93], ANSA [Her94], 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 simply to maximize performance rather than ensuring deadlines as with most real-time systems.

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 is suitable for combining existing software components, but lacks the degree of flexibility and fine-grain control found in our 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 [PSB+88, 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 the one described in section 3 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 internetworks, 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.

5 Conclusions

This paper addresses the problem of providing customized real-time communication services for different types of applications. We presented a new 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. We introduced the interaction model and outlined an implementation design based on the *x*-kernel model and the path abstraction in CORDS. The model was then applied to construct a real-time channel abstraction supporting multiple application interaction patterns, and numerous combinations of properties. As demonstrated by the example, this model supports a high level of customization without sacrificing real-time guarantees.

Prototype implementations of a runtime framework supporting the event-driven model and the channel abstraction are currently underway on a testbed of four Pentium PCs running OSF/RI MK 7.2. In addition to the implementation, future work will include further refinement of the admission control tests for the event-driven model, designing and implementing algorithms for distributed admission control, and experiments with other real-time services.

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.

[Bha96] N. T. Bhatti. *A System for Constructing Configurable High-Level Protocols*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Nov 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. T. Bhatti and R. D. 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.

[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. O. Guedes, D. E. Bakken, N. T. Bhatti, M. A. Hiltunen, and R. D. 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.

[Hil96] M. A. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Jul 1996.

[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.

[HS93] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, Oct 1993.

[HS95] M. A. Hiltunen and R. D. 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.

[HS97] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 1997. to appear.

[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.

[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 Institue, 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. D. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, Oct 1989.

[PBS89] L. Peterson, N. Buchholz, and R. D. 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.

[PSB+88] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the Eighteenth Symposium on Fault-Tolerant Computing*, Tokyo, Jun 1988.

[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.

[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.

[TMIM89] H. Tokuda, C. Mercer, Y. Ishikawa, and T. Marchok. Priority inversions in real-time communication. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 348–359, Dec 1989.

[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.