, , 1–28 ()

ARMADA Middleware and Communication Services*

T. ABDELZAHER, M. BJORKLUND, S. DAWSON, W.-C. FENG, F. JAHANIAN, S. JOHNSON, P. MARRON, A. MEHRA, T. MITTON, A. SHAIKH, K. SHIN, Z. WANG, H. ZOU

Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122, USA

Abstract. Real-time embedded systems have evolved during the past several decades from small custom-designed digital hardware to large distributed processing systems. As these systems become more complex, their interoperability, evolvability and cost-effectiveness requirements motivate the use of commercial-off-the-shelf components. This raises the challenge of constructing dependable and predictable real-time services for application developers on top of the inexpensive hardware and software components which has minimal support for timeliness and dependability guarantees. We are addressing this challenge in the ARMADA project.

ARMADA is set of communication and middleware services that provide support for fault-tolerance and end-to-end guarantees for embedded real-time distributed applications. Since real-time performance of such applications depends heavily on the communication subsystem, the first thrust of the project is to develop a predictable communication service and architecture to ensure QoS-sensitive message delivery. Fault-tolerance is of paramount importance to embedded safety-critical systems. In its second thrust, ARMADA aims to offload the complexity of developing fault-tolerant applications from the application programmer by focusing on a collection of modular, composable middleware for fault-tolerant group communication and replication under timing constraints. Finally, we develop tools for testing and validating the behavior of our services. We give an overview of the ARMADA project, describing the architecture and presenting its implementation status.

Keywords: distributed real-time systems, communication protocols, fault-tolerant systems

1. Introduction

ARMADA is a collaborative project between the Real-Time Computing Laboratory (RTCL) at the University of Michigan and the Honeywell Technology Center. The goal of the project is to develop and demonstrate an integrated set of communication and middleware services and tools necessary to realize embedded fault-tolerant and real-time services on distributed, evolving computing platforms. These techniques and tools together compose an environment of capabilities for designing, implementing, modifying, and integrating real-time distributed systems. Key challenges addressed by the ARMADA project include: timely delivery of services with end-to-end soft/hard real-time constraints; dependability of services in the presence of hardware or software failures; scalability of computation and communication resources; and exploitation of open systems and emerging standards in operating systems and communication services.

ARMADA communication and middleware services are motivated by the requirements of large embedded applications such as command and control, automated flight, shipboard

* This work is supported in part by a research grant from the Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Rome Laboratory under Grant F30602-95-1-0044.

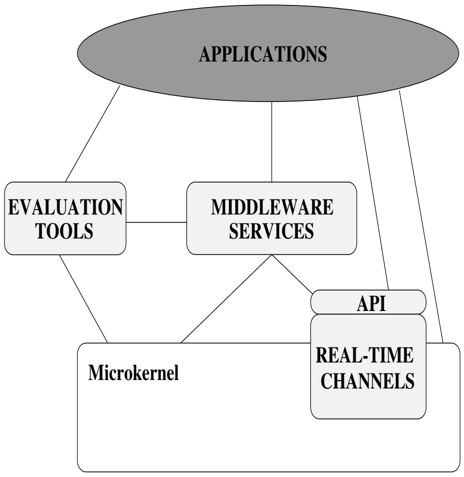


Figure 1. Overview of ARMADA Environment.

computing, and radar data processing. Traditionally, such embedded applications have been constructed from special-purpose hardware and software. This approach results in high production cost and poor interoperability making the system less evolvable and more prone to local failures. A recent trend, therefore, has been to build embedded systems using Commercial-Off-The-Shelf (COTS) components such as PC boards, Ethernet links, and PC-based real-time operating systems. This makes it possible to take advantage of available development tools, leverage on mass production costs, and make better use of component interoperability. From a real-time application developer's point of view, the approach creates the need for generic high-level software services that facilitate building embedded distributed real-time applications on top of inexpensive widely available hardware. Real-time operating systems typically implement elementary subsets of real-time services. However, monolithically embedding higher-level support in an operating system kernel is not advisable. Different applications have different real-time and fault-tolerance requirements. Thus, catering to all possible requirement ranges in a single operating system would neither be practical nor efficient. Instead, we believe that a composable set of services should be developed of which only a subset may need to exist for any given application. This philosophy advocates the use of a real-time microkernel equipped with basic real-time support such as priority-based scheduling and real-time communication, in addition to a reconfigurable set of composable middleware layered on top of the kernel. Appropriate testing and validation tools should be independently developed to verify required timeliness and fault-tolerance properties of the distributed middleware.

The ARMADA project is therefore divided into three complementary thrust areas: (i) low-level real-time communication support, (ii) middleware services for group communication and fault-tolerance, and (iii) dependability evaluation and validation tools. Figure 1 summarizes the structuring of the ARMADA environment.

The first thrust focused on the design and development of real-time communication services for a microkernel. A generic architecture is introduced for designing the communication subsystem on hosts so that predictability and QoS guarantees are maintained. The architecture is independent of the particular communication service. It is illustrated in this paper in the context of presenting the design of the *real-time channel*; a low-level communication service that implements a simplex, ordered virtual connection between two networked hosts that provides deterministic or statistical end-to-end delay guarantees between a sender-receiver pair.

The second thrust of the project has focused on a collection of modular and composable middleware services (or building blocks) for constructing embedded applications. A layered open-architecture supports modular insertion of a new service or implementation as requirements evolve over the life-span of a system. The ARMADA middleware services include a suite of fault-tolerant group communication services with real-time guarantees, called RTCAST, to support embedded applications with fault-tolerance and timeliness requirements. RTCAST consists of a collection of middleware including a group membership service, a timed atomic multicast service, an admission control and schedulability module, and a clock synchronization service. The ARMADA middleware services also include a real-time primary-backup replication service, called RTPB, which ensures *temporally consistent* replicated objects on redundant nodes.

The third thrust of the project is to build a toolset for validating and evaluating the timeliness and fault-tolerance capabilities of the target system. Tools under development include fault injectors at different levels (e.g. operating system, communication protocol, and application), a synthetic real-time workload generator, and a dependability/performance monitoring and visualization tool. The focus of the toolset research is on portability, flexibility, and usability.

Figure 2 gives an overview of a prospective application to illustrate the utility of our services for embedded real-time fault-tolerant systems. The application, developed at Honeywell, is a subset of a command and control facility. Consider a radar installation where a set of sensors are used to detect incoming threats (e.g., enemy planes or missiles in a battle scenario); hypotheses are formed regarding the identity and positions of the threats, and their flight trajectories are computed accordingly. These trajectories are extrapolated into the future and deadlines are imposed to intercept them. The time intervals during which the estimated threat trajectories are reachable from various ground defense bases are estimated; and appropriate resources (weapons) are committed to handle the threats; eventually, the weapons are released to intercept the threats.

The services required to support writing such applications come naturally from their operating requirements. For example, for the anticipated system load, communication between different system components (the different boxes in Figure 2) must occur in bounded time to ensure a bounded end-to-end response from threat detection to weapon release. Our real-time communication services compute and enforce predictable deterministic bounds on message delays given application traffic specification. Critical system components such as hypothesis testing and threat identification have high dependability requirements which are best met using active replication. For such components, RTCAST exports multicast and membership primitives to facilitate fault detection, fault handling, and consistency management of actively replicated tasks. Similarly, extrapolated trajectories of identified threats

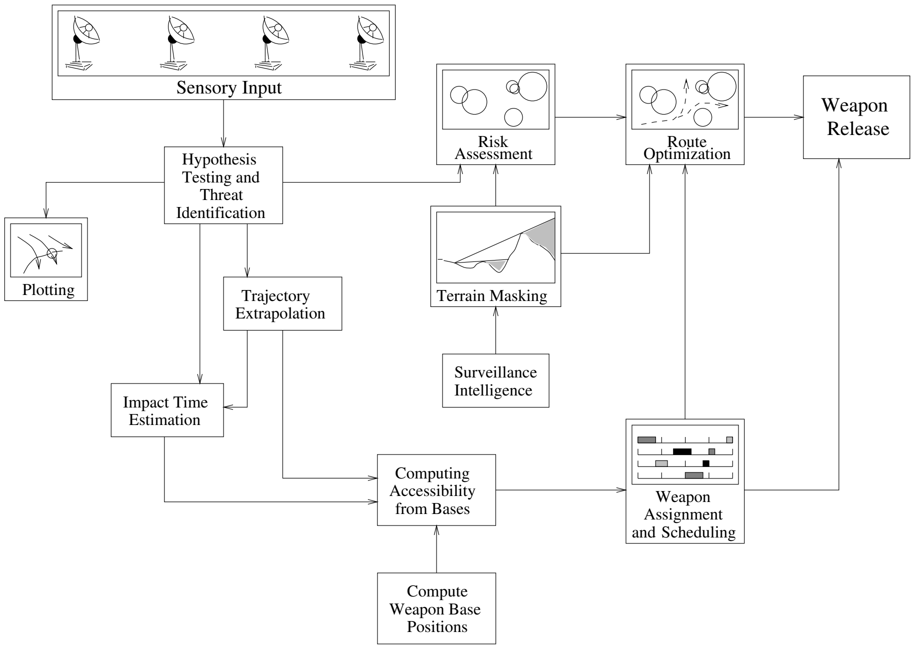


Figure 2. A command and control application

represent critical system state. A backup of such state needs to be maintained continually and updated to represent the current state within a tolerable consistency (or error) margin. Our primary-backup replication service is implemented to meet such temporal consistency requirements. Finally, our testing tools decrease development and debugging costs of the distributed application.

The rest of this paper is organized as follows. Section 2 describes the general approach for integrating ARMADA services into a microkernel framework. It also presents the experimental testbed and implementation environment of this project. The subsequent sections focus on the architecture, design, and implementation of key communication and middleware services in ARMADA. Section 3 introduces real-time communication service. Section 4 presents the RTCAST suite of group communication and fault-tolerance services. Section 5 describes the RTPB (real-time primary-backup) replication service. Section 6 briefly discusses the dependability evaluation and validation tools developed in this project. Section 7 concludes the paper.

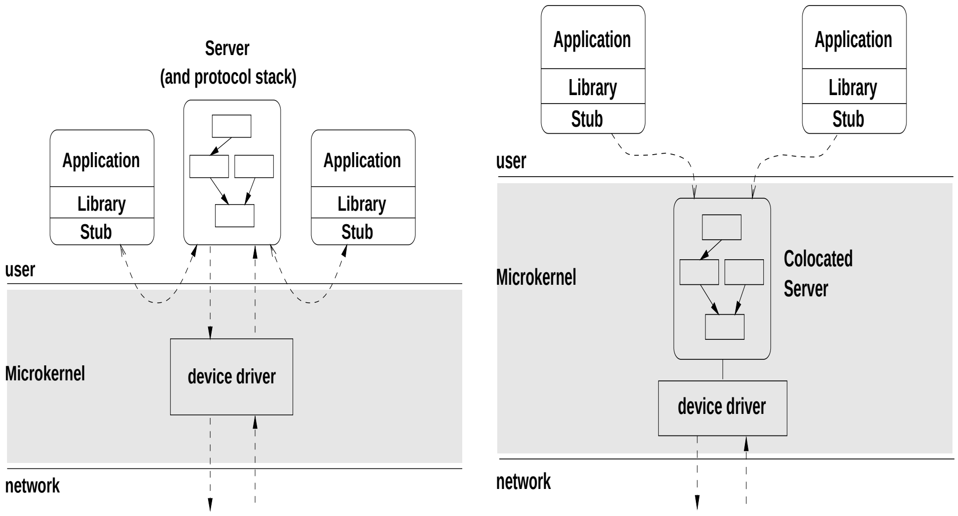
2. Platform

The services developed in the context of the ARMADA project are to augment the essential capabilities of a real-time microkernel by introducing a composable collection of communication, fault-tolerance, and testing tools to provide an integrated framework for developing and executing real-time applications. Most of these tools are implemented as separate multithreaded servers. Below we describe the experimental testbed and implementation environment common to the aforementioned services. A detailed description of the implementation approach adopted for various services will be given in the context of each particular service.

2.1. General Service Implementation Approach

One common aspect of different middleware services in a distributed real-time system is their need to use intermachine communication. All ARMADA services either include or are layered on top of a communication layer which provides the features required for correct operation of the service and its clients. For example, RTCAST implements communication protocols to perform multicast and integrate failure detection and handling into the communication subsystem. Similarly, the Real-Time Channels service implements its own signaling and data transfer protocols to reserve resources and transmit real-time data along a communication path. Since communication seemed to warrant particular attention in the context of this project, we developed a generic real-time communication subsystem architecture. The architecture can be viewed as a way of structuring the design of communication-oriented services for predictability, as opposed to being a service in itself. This architecture is described in detail in Section 3 and is illustrated by an example service: the Real-Time Channel. ARMADA communication services are generally layered on top of IP, or UDP/IP. We do not use TCP because its main focus is reliability as opposed to predictability and timeliness. Real-time communication protocols, on the other hand, should be sensitive to timeliness guarantees, perhaps overriding the reliability requirement. For example, in video conferencing and process control, occasional loss of individual data items is preferred to receiving reliable streams of stale data. To facilitate the development of communication-oriented services, our communication subsystem is implemented using the x-kernel object-oriented networking framework originally developed at the University of Arizona (Hutchinson and Peterson, 1991), with extensions for controlled allocation of system resources (Travostino et al., 1996). The advantage of using x-kernel is the ease of composing protocol stacks. An x-kernel communication subsystem is implemented as a configurable graph of protocol objects. It allows easy reconfiguration of the protocol stack by adding or removing protocols. More details on the x-kernel can be found in (Hutchinson and Peterson, 1991).

Following a microkernel philosophy, argued for in Section 1, our services are designed as user-level multithreaded servers. Clients of the service are separate processes that communicate with the server via the kernel using a user library. The library exports the desired middleware API. Communication-oriented services generally implement their own protocol stack that lies on top of the kernel-level communication driver. The x-kernel framework permits migration of multithreaded protocol stack execution into the operating system kernel. We use this feature to implement server co-location into the microkernel. Such co-location improves performance by eliminating extra context switches. Note that the advantages of server co-location do not defeat the purpose of choosing a microkernel over a monolithic operating system for a development platform. This is because with a microkernel co-located servers (i) can be developed *in user space* which greatly reduces their development and maintenance cost, and (ii) can be *selectively included*, when needed,



(a) User-level server configuration (b) Co-located server

Figure 3. Service implementation.

into the kernel in accordance with the application requirements; this is both more efficient and more sensitive to particular application needs.

The microkernel has to support kernel threads. The priority of threads executing in kernel space is, by default, higher than that of threads executing in user space. As a result, threads run in a much more predictable manner, and the service does not get starved under overload. Furthermore, the in-kernel implementation of x-kernel on our platform replaces some of the threads in the device driver by code running in interrupt context. This feature reduces communication latencies and makes the server less preemptable when migrated into the microkernel. However, since code executing in interrupt context is kept to a minimum, the reduction in preeptability has not been a concern in our experiences with co-located code.

Figure 3-a and 3-b illustrate the configurations of user-level servers and co-located servers respectively. An example of server migration into the kernel is given in the context of the RTCAST service in Section 4. The RTCAST server was developed in user space (as in Figure 3-a), then reconfigured to be integrated the into the kernel (as in Figure 3-b). Whether the server runs in user space or is co-located in the microkernel, client processes use the same service API to communicate with it. If the service is co-located in the kernel, an extra context switch to/from a user-level server process is saved. Automatically-generated stubs interface the user library (implementing the service API) to the microkernel or the server process. These stubs hide the details of the kernel's local communication mechanism from the programmer of the real-time service, thus making service code independent from specifics of the underlying microkernel.

2.2. *Testbed and Implementation Environment*

In the following sections we describe the implementation of each individual service. To provide a common context for that description, we outline here the specifics of the underlying implementation platform. Our testbed comprises several Pentium-based PCs (133 MHz) connected by a Cisco 2900 Ethernet switch (10/100 Mb/s), with each PC connected to the switch via 10 Mb/s Ethernet. We have chosen the MK 7.2 microkernel operating system from the Open Group (OG)[1] Research Institute to provide the essential underlying real-time support for our services. The MK microkernel is originally based on release 2.5 of the Mach operating system from CMU. While not a full-fledged real-time OS, MK 7.2 supports kernel threads, priority-based scheduling, and includes several important features that facilitate provision of QoS guarantees. For example, MK 7.2 supports x-kernel and provides a unified framework for allocation and management of communication resources. This framework, known as *CORDS* (Communication Objects for Real-time Dependable Systems) (Travostino et al., 1996), was found particularly useful for implementing real-time communication services. Our implementation approach has been to utilize the functionality and facilities provided in OG's environment and augment it with our own support when necessary.

From the standpoint of portability, although MK7.2 is a research operating system, CORDS support is also available on more mainstream operating systems such as Windows NT. Thus, our software developed for the CORDS environment can easily be ported to NT. In fact, such port is currently underway. Porting to other operating systems, such as Linux, is more difficult. At the time the presented services were developed Linux did not support kernel threads. Thus, it was impossible to implement multithreaded protocol stacks inside the Linux kernel. Linux 2.2, however, is expected to have full thread support. CORDS support may be replaced by appropriate packet filters to classify incoming traffic. Thus, with some modifications, our services may be ported to future versions of Linux, as well as other multithreaded operating systems such as Solaris.

3. ARMADA Real-Time Communication Architecture

ARMADA provides applications with a communication architecture and service with which they can request and utilize guaranteed-QoS connections between two hosts. In this section, we hilight the architectural components of the communication service that, together with a set of user-specified policies, can implement several real-time communication models.

Common to QoS-sensitive communication service models are the following three architectural requirements: (i) performance isolation between connections or sets of connections such that malicious behavior or overload of one does not starve resources of the other(s), (ii) service differentiation, such as assigning different priorities to connections or classes of connections, and (iii) graceful degradation in the presence of overload. We developed a Communication Library for Implementing Priority Semantics (CLIPS), that provides resource-management mechanisms to satisfy the aforementioned requirements. It exports the abstraction of guaranteed-rate communication endpoints. The endpoint, called a *clip*, guarantees a certain throughput in terms of the number of packets sent via it per period, and implements a configurable buffer to accommodate bursty sources. One or more connections (or sockets) may be "bound" to the same clip, in which case the clip sets aside enough processor bandwidth and memory resources on the end-system to guarantee an aggregate specified throughput for the entire connection set. Different clips may have different priorities to allow higher priority traffic to proceed first under overload conditions. For example, traffic of a particular application or middleware service can be bound to a high priority clip, thereby allowing that application or service to receive precedence over other services. Each clip has an associated deadline parameter. The deadline specifies the maximum communication subsystem response time for handling packets via the particular clip. The CLIPS library implements a traffic policing mechanism, as well as its own default admission control policy that can be disabled to revert to pure priority-driven scheduling or overridden by a user-specified alternate admission control policy. More details on CLIPS will be given below as we present the ARMADA real-time communication service we developed for unicast communication.

3.1. Real-time Communication Service

We have used CLIPS to implement a guaranteed-QoS communication service called the *real-time channel* (Ferrari and Verma, 1990, Kandlur et al., 1994). A real-time channel is a unicast virtual connection between a source and destination host with associated performance guarantees on message delay and available bandwidth. It satisfies three primary architectural requirements for guaranteed-QoS communication (Mehra et al., 1996): (i) maintenance of per-connection QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic. Real-time communication via real-time channels is performed in three phases. in the first phase, the source host S (sender) creates a channel to the destination host D (receiver) by specifying the channel's traffic parameters and QoS requirements. Signaling requests are sent from S to D via one or more intermediate (I) nodes; replies are delivered in the reverse direction from D to S. If successfully established, S can send messages on this channel to D; this constitutes the second phase. When the sender is done using the channel, it must close the channel (the third phase) so that resources allocated to this channel can be released.

Figure 4 illustrates the high-level software architecture of our guaranteed-QoS service at end-hosts. The core functionality of the communication service is realized via three distinct components that interact to provide guaranteed-QoS communication. Applications use the service via the real-time communication application programming interface (RTC API); RTCOP coordinates end-to-end signaling for resource reservation and reclamation during connection set-up or tear-down; and CLIPS performs run-time management of resources for QoS-sensitive data transfer. Since platform-specific overheads must be characterized before QoS guarantees can be ensured, an execution profiling component is added to measure and parameterize the overheads incurred by the communication service on a particular platform, and make these parameters available for admission control decisions. The control path taken through the architecture during connection setup is shown in Figure 4 as dashed lines. Data is then transferred via RTC API and CLIPS as indicated by the solid lines. Below, we discuss the salient features of each architectural component of the service along with its interaction with other components to provide QoS guarantees. We also describe how the components are used to realize a particular service model.

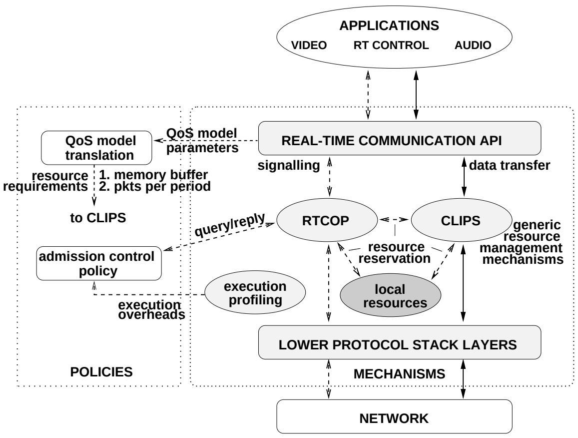


Figure 4. **Real-time communication service architecture**: Our implementation consists of four primary architectural components: an application programming interface (RTC API), a signaling and resource reservation protocol (RTCOP), support for resource management and run-time data transfer (CLIPS), and execution profiling support. Dashed lines indicate interactions on the control path while the data path is denoted by the solid lines.

3.2. RTC Application Interface

The programming interface exported to applications comprises routines for connection establishment and teardown, message transmission and reception during data transfer on established connections, and initialization and support routines. Table 1 lists some of the main routines currently available in RTC API. The API has two parts: a top half that interfaces to applications and is responsible for validating application requests and creating internal state, and a bottom half which interfaces to RTCOP for signaling (i.e., connection setup and teardown), and to CLIPS for QoS-sensitive data transfer.

The design of RTC API is based in large part on the well-known socket API in BSD Unix. Each connection endpoint is a pair `(IPaddr, port)` formed by the IP address of the host (`IPaddr`) and an unsigned 16-bit port (`port`) unique on the host, similar to an INET domain socket endpoint. In addition to unique endpoints for data transfer, an application may use several endpoints to receive signaling requests from other applications. Applications willing to be receivers of real-time traffic register their signaling ports with a name service or use well-known ports. Applications wishing to create connections must first locate the corresponding receiver endpoints before signaling can be initiated.

Each of the signaling and data transfer routines in Table 1 has its counterpart in the socket API. For example, the routine `rtcRegisterPort` corresponds to the invocation of `bind` and `listen` in succession, and `rtcAcceptConnection` corresponds to `accept`. Similarly, the routines `rtcCreateConnection` and `rtcDestroyConnection` correspond to `connect` and `close`, respectively.

The key aspect which distinguishes RTC API from the socket API is that the receiving application *explicitly approves* connection establishment and teardown. When registering

Table 1. **Routines comprising RTC API**: This table shows the utility, signaling, and data transfer functions that constitute the application interface. The table shows each function name, its parameters, the endpoint that invokes it, and a brief description of the operation performed.

Routines	Parameters	Invoked By	Function Performed
`rtcInit`	none	both	service initialization
`rtcGetParameter`	chan id, param type	both	query parameter on specified real-time connection
`rtcRegisterPort`	local port, agent function	receiver	register local port and agent for signaling
`rtcUnRegisterPort`	local port	receiver	unregister local signaling port
`rtcCreateConnection`	remote host/port, QoS: max rate, max burst size max msg size, max delay	sender	create connection with given parameters to remote endpoint; return connection id
`rtcAcceptConnection`	local port, chan id, remote host/port	receiver	obtain the next connection already established at specified local port
`rtcDestroyConnection`	chan id	sender	destroy specified real-time connection
`rtcSendMessage`	chan id, buf ptr	sender	send message on specified real-time connection
`rtcRecvMessage`	chand id, buf ptr	receiver	receive message on specified real-time connection

its intent to receive signaling requests, the application specifies an agent function that is invoked in response to connection requests. This function, implemented by the receiving application, determines whether sufficient application-level resources are available for the connection and, if so, reserves necessary resources (e.g., CPU capacity, buffers, etc.) for the new connection. It may also perform authentication checks based on the requesting endpoint specified in the signaling request. This is unlike the establishment of a TCP connection, for example, which is completely transparent to the peer applications.

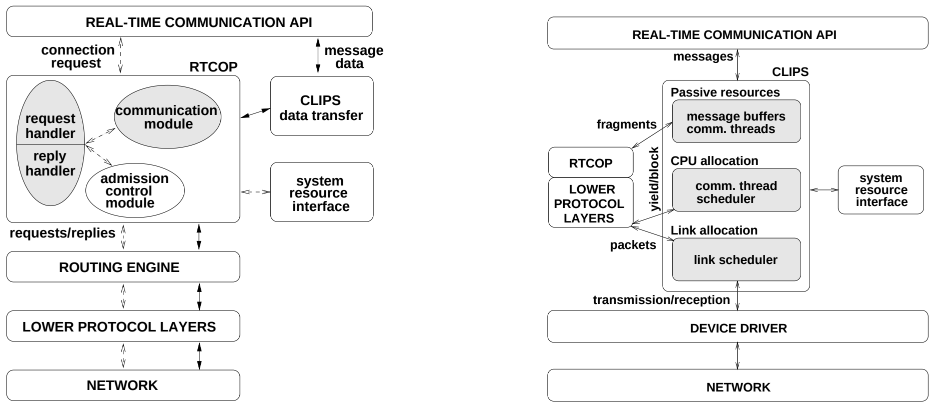
The QoS-parameters passed to `rtcCreateConnection` for connection establishment describe a *linear bounded arrival* traffic generation process (Cruz, 1987, Anderson et al., 1990). They specify a maximum message size (M_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). Parameters M_{max} and R_{max} are used to create a clip with a corresponding guaranteed throughput. The burst size, B_{max}, determines the buffer size required for the clip. In the following we describe the end-to-end signaling phase that coordinates end-to-end resource reservation.

3.3. Signaling and Resource Reservation with RTCOP

Requests to create and destroy connections initiate the Real-Time Connection Ordination Protocol (RTCOP), a distributed end-to-end signaling protocol. As illustrated in Figure 5(a), RTCOP is composed primarily of two relatively independent modules. The *request and reply handlers* manage signaling state and interface to the admission control policy, and the *communication module* handles the tasks of reliably forwarding signaling messages. This separation allows simpler replacement of admission control policies or connection state management algorithms without affecting communication functions. Note that signaling and connection establishment are non-real-time (but reliable) functions. QoS guarantees apply to the data sent on an established connection but signaling requests are sent as best-effort traffic.

The request and reply handlers generate and process signaling messages, interface to RTC API and CLIPS, and reserve and reclaim resources as needed. When processing a new signaling request, the request handler invokes a multi-step admission control procedure to decide whether or not sufficient resources are available for the new request. As a new connection request traverses each node of the route from source to destination, the request handler invokes admission control which decides if the new connection can be locally admitted. Upon successful admission, the handler passes the request on to the next hop. When a connection is admitted at all nodes on the route, the reply handler at the destination node reserves the required end-system resources by creating a clip for the new real-time channel, and generates a positive acknowledgment on the reverse path to the source. As the notification is received at each hop, the underlying network-level protocol commits network resources, such as link bandwidth, using assumed local router support. When the acknowledgement is received at the source the reply handler notifies the application of connection establishment and creates the source clip.

The communication module handles the basic tasks of sending and receiving signaling messages, as well as forwarding data packets to and from the applications. Most of the protocol processing performed by the communication module is in the control path during processing of signaling messages. In the data path it functions as a simple transport pro-



(a) RTCOP structure (b) CLIPS structure

Figure 5. **Internal structures and interfaces:** In this figure we show the internal functional structure of RTCOP and CLIPS along with their respective interfaces to other components. In (a), data and control paths are represented with solid and dashed lines, respectively.

tocol, forwarding data packets on behalf of applications, much like UDP. As noted earlier, signaling messages are transported as best-effort traffic, but are delivered reliably using source-based retransmissions. Reliable signaling ensures that a connection is considered established only if connection state is successfully installed and sufficient resources reserved at all the nodes along the route. The communication module implements duplicate suppression to ensure that multiple reservations are not installed for the same connection establishment request. Similar considerations apply to connection teardown where all nodes along the route must release resources and free connection state. Consistent connection state management at all nodes is an essential function of RTCOP.

RTCOP exports an interface to RTC API for specification of connection establishment and teardown requests and replies, and selection of logical ports for connection endpoints. The RTC API uses the latter to reserve a signaling port in response to a request from the application, for example. RTCOP also interfaces to an underlying routing engine to query an appropriate route before initiating signaling for a new connection. In general, the routing engine should find a route that can support the desired QoS requirements. However, for simplicity we use static (fixed) routes for connections since it suffices to demonstrate the capabilities of our architecture and implementation.

3.4. *CLIPS-based Resource Scheduling for Data Transfer*

CLIPS implements the necessary end-system resource-management mechanisms to realize QoS-sensitive real-time data transfer on an established connection. A separate clip is created for each of the two endpoints of a real-time channel. Internal to each clip is a *message queue* to buffer messages generated or received on the corresponding channel, a *communication handler thread* to process these messages, and a *packet queue* to stage packets waiting to be transmitted or received. The CLIPS library implements on the end-system the key functional components illustrated in Figure 5(b).

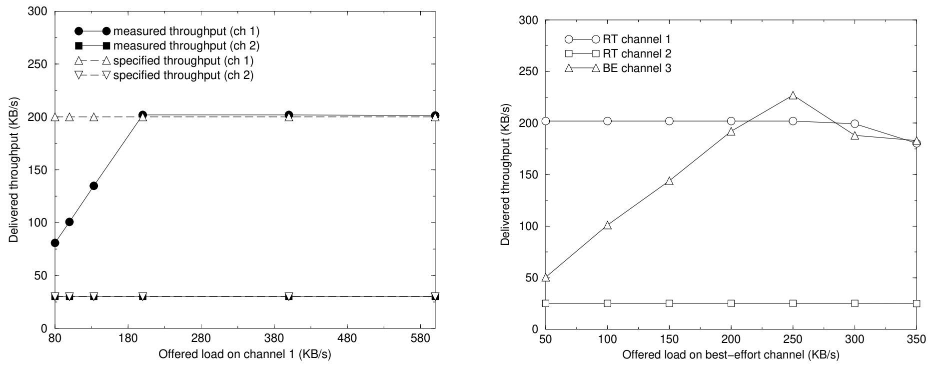
QoS-sensitive CPU scheduling: The communication handler thread of a clip executes in a continuous loop either dequeuing outgoing messages from the clip's message queue and fragmenting them (at the source host), or dequeuing incoming packets from the clip's packet queue and reassembling messages (at the destination host). Each message must be sent within a given local delay bound (deadline). To achieve the best schedulable utilization, communication handlers are scheduled based on an earliest-deadline-first (EDF) policy. Since most operating systems do not provide EDF scheduling, CLIPS implements it with a user-level scheduler layered on top of the operating system scheduler. The user-level scheduler runs at a static priority and maintains a list of all threads registered with it, sorted by increasing deadline. At any given time, the CLIPS scheduler blocks all of the registered threads using kernel semaphores except the one with the earliest deadline, which it considers in the running state. The running thread will be allowed to execute until it explicitly terminates or yields using a primitive exported by CLIPS. The scheduler then blocks the thread on a kernel semaphore and signals the thread with the next earliest deadline. Preemption is implemented via a CLIPS primitive invoked upon sending each packet. The primitive yields execution to a more urgent thread if one is pending. This arrangement implements EDF scheduling within a single protection domain.

Resource reservation: Communication handlers (implemented by CLIPS) execute a user-defined protocol stack, then return to CLIPS code after processing each message or packet. Ideally, each clip should be assigned a *CPU budget* to prevent a communication client from monopolizing the CPU. Since processor capacity reserves are not available on most operating systems, the budget is indirectly expressed in terms of a maximum number of packets to be processed within a given period. The handler blocks itself after processing the maximum number of packets allowed within its stated time period.

Policing: Associating a budget with each connection handler facilitates traffic enforcement. This is because a handler is scheduled for execution only when the budget is non-zero, and the budget is not replenished until the next (periodic) invocation of the handler. This mechanism ensures that misbehaving connections are policed to their traffic specification.

QoS-sensitive link bandwidth allocation: Modern operating systems typically implement FIFO packet transmission over the communication link. While we cannot avoid FIFO queuing in the kernel's network device, CLIPS implements a dynamic priority-based *link scheduler* at the bottom of the user-level protocol stack to schedule outgoing packets in a prioritized fashion. The link scheduler implements the EDF scheduling policy using a priority heap for outgoing packets. To prevent a FIFO accumulation of outgoing packets in the kernel (e.g., while the link is busy), the CLIPS link scheduler does not release a new packet until it is notified of the completion of previous packet transmission. Best-effort packets are maintained in a separate packet heap within the user-level link scheduler and serviced at a lower priority than those on real-time clips.

Figure 6 demonstrate traffic policing, traffic isolation and performance differentiation in real-time channels. A more detailed evaluation is found in (Mehra et al., 1998).



(a) Isolation between real-time channels (b) Isolation between best-effort and real-time

Figure 6. **Traffic isolation:** The left graph shows that real time channel 1 is policed to its traffic specification, disallowing violation of that specification. Traffic on real-time channel 1 does not affect the QoS for the other real-time channel 2. The right graph shows that increasing best-effort load does not interfere with real-time channel throughput.

4. RTCAST Group Communication Services

The previous section introduced the architecture of the ARMADA real-time communication service. This architecture sets the ground for implementing real-time services with QoS-sensitive communication. The second thrust of the project has focused on a collection of such services, that provide modular and composable middleware for constructing embedded applications. The ARMADA middleware can be divided into two relatively independent suites of services:

- RTCAST group communication services, and
- RTPB real-time primary-back replication service.

This section presents the RTCAST suite of group communication and fault-tolerance services. Section 5 describes the RTPB (real-time primary-backup) replication service.

4.1. RTCAST Protocols

The QoS-sensitive communication service described in Section 3 does not support multicast channels. Multicast is important, e.g., for efficient data dissemination to a set of destinations, or for maintaining replicated state in fault-tolerant systems. If consistency of replicated state is desired, a membership algorithm is also needed. RTCAST complements aforementioned unicast communication services by mulitcast and membership services for real-time fault-tolerant applications. RTCAST is based around the *process groups* paradigm.

Process groups are a widely-studied paradigm for designing distributed systems in both asynchronous (Birman, 1993, Amir et al., 1992, van Renesse et al., 1994, Mishra, 1993) and synchronous (Kopetz and Grünsteidl, 1994, Amir et al., 1995, Cristian et al., 1990) environments. In this approach, a distributed system is structured as a group of cooperating processes which provide service to the application. A process group may be used, for example, to provide active replication of system state or to rapidly disseminate information from an application to a collection of processes. Two key primitives for supporting process groups in a distributed environment are *fault-tolerant multicast communication* and *group membership*. Coordination of a process group must address several subtle issues including delivering messages to the group in a reliable fashion, maintaining consistent views of group membership, and detecting and handling process or communication failures. If multicast messages are atomic and globally ordered, consistency of replicated state will be guaranteed.

RTCAST is especially designed for real-time applications. In a real-time application, timing failures may be as damaging as processor failures. Thus, our membership algorithm is more aggressive in ensuring timely progress of the process group. For example, while ensuring atomicity of message delivery, RTCAST does not require acknowledgments for every message, and message delivery is immediate without needing additional "rounds" of message transmissions to ensure that a message was received consistently by all destinations. RTCAST is designed to support hard real-time guarantees without requiring a static schedule to be computed *a priori* for application tasks and messages. Instead, an on-line schedulability analysis component performs admission control on multicast messages. We envision the proposed multicast and membership protocols as part of a larger suite of middleware group communication services that form a composable architecture for the development of embedded real-time applications.

As illustrated in Figure 7, the RTCAST suite of services include a timed atomic multicast, a group membership service and an admission control service. The first two are tightly coupled and thus are considered a single service. Clock synchronization is typically required for real-time protocols and is enforced by the clock synchronization service. To support portability, a *virtual network interface* layer exports a uniform network abstraction. Ideally, this interface would transparently handle different network topologies, each having different connectivity and timing or bandwidth characteristics exporting a generic network abstraction to upper layers. The network is assumed to support unicast datagram service. Finally, the top layer provides an application programming interface for real-time process group.

RTCAST supports bounded-time message transport, atomicity, and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures. It guarantees agreement on membership among the communicating processors, and ensures that membership changes (e.g., resulting from processor joins or departures) are atomic and ordered with respect to multicast messages. RTCAST assumes that processes can communicate with the environment only by sending messages. Thus, a failed process, for example, cannot adversely affect the environment via a hidden channel. RTCAST proceeds as senders in a logical ring take turns in multicasting messages over the network. A processor's turn comes when the logical token arrives, or when it times out waiting for it. After its last message, each sender multicasts a heartbeat that is used for crash detection. The heartbeat received from an immediate predecessor also serves as the logical token. Destinations detect missed messages using sequence numbers and when

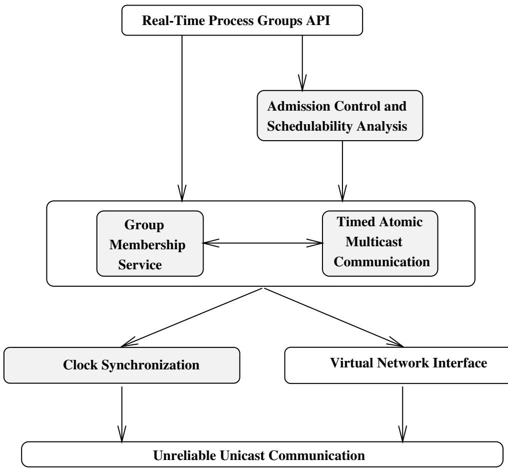


Figure 7. Software architecture for the RTCAST middleware services.

a processor detects a receive omission, it crashes. Each processor, when its turn comes, checks for missing heartbeats and eliminates the crashed members, if any, from group membership by multicasting a membership change message.

In a token ring, sent messages have a natural order defined by token rotation. We reconstruct message order at the receivers using a protocol layer below RTCAST which detects out-of-order arrival of messages and swaps them, thus forwarding them to RTCAST in correct order. RTCAST ensures that "correct" members can reach agreement on replicated state by formulating the problem as one of group membership. Since the state of a process is determined by the sequence of messages it receives, a processor that detects a message receive omission takes itself out of the group, thus maintaining agreement among the remaining ones. In a real-time system one may argue that processes waiting for a message that does not arrive will miss their deadlines anyway, so it is acceptable to eliminate the processor(s) which suffered receive omissions.[2] A distinctive feature of RTCAST is that processors which did not omit any messages can deliver messages as soon as they arrive without compromising protocol semantics. Thus, for example, if a reliable multicast is used to disseminate a critical message to a replicated server, and if one of the replicas suffers a receive omission, RTCAST will eliminate that replica from the group, while delivering the message to the remaining replicas immediately. This is in contrast to delaying delivery of the message until *all* replicas have received it. The approach is motivated by the observation that in a real-time system it may be better to sacrifice *one* replica in the group than delay message delivery potentially causing *all* replicas to miss a hard timing constraint. Finally, membership changes are communicated exclusively by *membership change messages* using our multicast mechanism. Since message multicast is atomic and ordered, so are the membership changes. This guarantees agreement on membership view.

From an architectural standpoint, RTCAST operation is triggered by two different event types, namely message reception, and token reception (or timeout). It is therefore logically

```
msg_reception_handler()
    if state = RUNNING
        if more msgs from same member
            if missed msgs → CRASH else
                deliver msg
        else if msg from different member
            if missed msgs → CRASH else
                check for missed msgs from processors between current and last senders
                if no missing msgs
                    deliver current msg
                else CRASH
        else if join msg from non-member
            handle join request
    if state = JOINING AND msg is a valid join_ack
        if need more join_acks
            wait for additional join_acks
        else state = RUNNING
end
```

Figure 8. Message reception handler

structured as two event handlers, one for each event type. The **message reception handler** (Figure 8) detects receive omissions if any, delivers messages in order to the application, and services protocol control messages. The **token handler** (Figure 9) is invoked when the token is received or when the token timeout expires. It detects processor crashes and sends membership change notifications, if any, as well as lets client processes send out their messages during the processors finite token hold time.

4.2. RTCAST Design and Implementation

This section describes some of the major issues in the design and implementation of RTCAST; our representative group communication service. A thorough performance evaluation of the service is reported on in (Abdelzaher et al., 1996) and (Abdelzaher et al., 1997).

The RTCAST application was implemented and tested over a local Ethernet. Ethernet is normally unsuitable for real-time applications due to packet collisions and the subsequent retransmissions that make it impossible to impose deterministic bounds on communication delay. However, since we use a *private* Ethernet (i.e. the RTCAST protocol has exclusive access to the medium), only one machine can send messages at any given time (namely, the token holder). This prevents collisions and guarantees that the Ethernet driver always succeeds in transmitting each packet on the first attempt, making message communication delays deterministic. The admission control service described previously can take

```
token_handler()
    if state = RUNNING
        for each processor p in current membership view
            if no heartbeat seen from all predecessors incl. p
                remove p from group view
                multicast new group view
        send out all queued messages
        mark the last msg
        send out heartbeat msg
    if state = JOINING
        send out join msg
end
```

Figure 9. Token handler

advantage of this predictability, e.g., by creating appropriate clips to manage end-system resources on each host and make real-time guarantees on messages sent with RTCAST.

4.2.1. Protocol Stack Design The RTCAST protocol was designed to be modular, so that individual services could be added, changed, or removed without affecting the rest of the protocol. Each service is designed as a separate protocol layer within the x-kernel (Hutchinson and Peterson, 1991) protocol framework. The x-kernel is an ideal choice for implementing the RTCAST middleware services because application requirements can be easily met by simply reconfiguring the protocol stack to add or remove services as necessary. The RTCAST implementation uses the following protocol layers:

Admission Control: The Admission Control and Schedulability Analysis (ACSA) layer is a distributed protocol that keeps track of communication resources of the entire process group. The protocol transparently creates a clip on each host that runs the process group to ensure communication throughput guarantees and time-bounded message processing. It can support multiple either prioritized or performance isolated process groups on the same machine by creating clips of corresponding priority and corresponding minimum throughput specification. If real-time guarantees are not needed, this layer can be omitted from the protocol stack to reduce overhead. Communication will then proceed on best-effort basis.

RTCAST: The RTCAST protocol layer encompasses the membership, logical token ring, and atomic ordering services described in section 4.

Multicast Transport: This protocol implements an unreliable multicast abstraction that is independent of the underlying network. RTCAST uses the multicast transport layer to send messages to the group without having to worry about whether the physical medium provides unicast, broadcast, or true multicast support. The details of how the messages are actually sent over the network are hidden from higher layers by the multicast transport

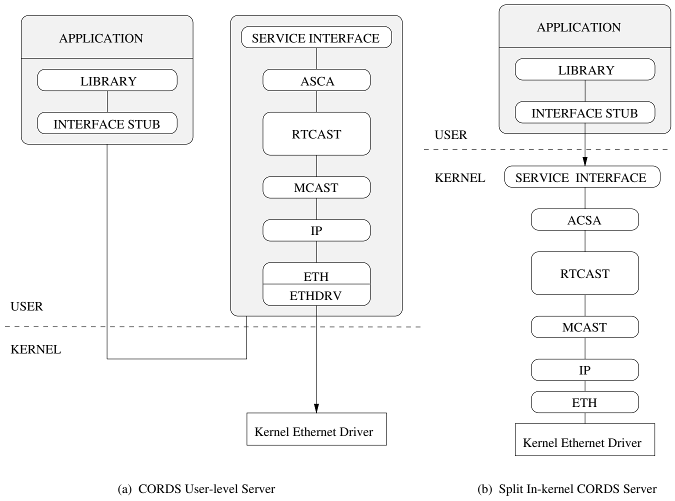


Figure 10. RTCAST protocol stack as implemented

protocol, so it is the only layer that must be modified when RTCAST is run on different types of networks.

Figure 10 shows the full protocol stack as it is implemented on our platform.

4.2.2. Integration Into the Mach Kernel As figure 10 shows, the protocol stack representing the core of the service was migrated into the Mach kernel. While actual RTCAST development took place in user space to facilitate debugging, its final co-location within the Mach kernel has several performance advantages. First, as with any group communication protocol, there can be a high amount of CPU overhead to maintain the group state and enforce message semantics. By running in the kernel, the RTCAST protocol can run at the highest priority and minimize communication latency due to processing time. Second, in the current implementation of MK 7.2 there is no operating system support for real-time scheduling or capacity reserve. Experience shows that processes running at the user level can be starved for CPU time for periods of up to a few seconds, which would be disastrous for RTCAST's predictable communication. By running in the kernel, protocol threads do not get starved significantly and are scheduled in a much more predictable manner by the operating system. Finally, there is a problem with the MK 7.2 implementation of the x-kernel, such that threads which are shepherding messages up the protocol stack can be queued to run in a different order than the messages arrive from the network. This results in out-of-order messages that must be buffered and re-ordered to maintain the total ordering guarantees provided by the protocol. Having to buffer and reorder messages also delays crash detection, since there is no way of knowing if a missing message is queued somewhere in the protocol stack or if the sender suffered a failure. By running the protocol in the kernel, message threads are interrupt driven and run immediately after arriving from the network, so the message reordering problem does not occur. Protocol performance improved almost by an order of magnitude when executed in the kernel. For example, when executed at the user-level, the minimum token rotation time was on average 2.6 ms, 5.7 ms, and 9.6 ms for groups with one, two, and three members respectively. When running in the kernel, the same measurement yielded token rotation times of 0.43 ms, 1.02 ms, and 1.55 ms. We found that this improvement extended to all aspects of protocol performance. Note that the above figures suggest a potential scalability problem for larger group sizes (such as hundreds of nodes). The problem is attributed to the need for software token passing. Integration with hardware token passing schemes, such as FDDI, will yield much better performance. Alternatively, to improve scalability, we are currently investigating an approach based on group composition. Larger process groups are formed by a composition of smaller ones. This research is presently underway. Initial results show that composite process groups scale much better than monolithic ones.

Another important focus in developing our group communication middleware was designing a robust API that would allow application developers to take advantage of our services quickly and easily. RTCAST API includes (i) bandwidth reservation calls, (ii) process group membership manipulation functions, (iii) best-effort multicast communication primitives and (iv) reliable real-time multicast. Bandwidth reservation is used on hosts to ensure that a multicast connection has dedicated CPU capacity and network bandwidth (i.e. a minimum token hold time). The token hold time and token rotation period specify the communication bandwidth allotted to the node. The node can set aside enough end-system resources to utilize its allotted communication bandwidth by creating a clip (by the ACSA layer) of a corresponding throughput thereby providing schedulability guarantees. The membership manipulation functions allow processes to join and leave the multicast group, query current group membership, create groups, etc. There are two types of group communication: real-time multicast communication that guarantees end-to-end response time, and best-effort which does not. The advantages of using a best-effort connection is that it is optimized for throughput as opposed to meeting individual message deadlines. Thus, the service protocol stack is faster on the average (e.g., no per-message admission control), but the variance in queuing delays is higher.

We collaborated with a group of researchers at the Honeywell Technology Center to implement a subset of the fault-tolerant real-time distributed application described in Section 1 using the RTCAST protocol. Using the insights gained from this motivating application, we were able to refine the API to provide the required of functionality while maintaining a simple interface that is easy to program. Based on our experience of the application's use of the protocol, we also designed a higher-level service library that can be linked with the application, and which uses the RTCAST API[3]. It is concerned with resource management in a fault-tolerant system and with providing higher-level abstractions of the protocol communication primitives. The service library provides for logical processing nodes and resource pools that transparently utilize RTCAST group communication services. These abstractions provide a convenient way for application developers to reason about and structure their redundancy management and failure handling policies while RTCAST does the actual work of maintaining replica consistency.

5. Real-Time Primary-backup (RTPB) Replication Service

While the previous section introduced a middleware service for active replication, in this section we present the overall architecture of the ARMADA real-time primary-backup replication service. We first give an introduction to the RTPB system, then describe the service framework. Finally we discuss implementation of the service that we believe meets the objectives.

5.1. Introduction to RTPB

Keeping large amounts of application state consistent in a distributed system, as in the state machine approach, may involve a significant overhead. Many real-time applications, however, can tolerate minor inconsistencies in replicated state. Thus, to reduce redundancy management overhead, our primary-backup replication exploits application data semantics by allowing the backup to maintain a less current copy of the data that resides on the primary. The application may have distinct tolerances for the staleness of different data objects. With sufficiently recent data, the backup can safely supplant a failed primary; the backup can then reconstruct a consistent system state by extrapolating from previous values and new sensor readings. However, the system must ensure that the distance between the primary and the backup data is bounded within a predefined time window. Data objects may have distinct tolerances in how far the backup can lag behind before the object state becomes stale. The challenge is to bound the distance between the primary and the backup such that consistency is not compromised, while minimizing the overhead in exchanging messages between the primary and its backup.

5.2. Service Framework

A very important issue in designing a replication service is its consistency semantics. One category of consistency semantics that is particular relevant to the primary-backup replication in a real-time environment is *temporal consistency*, which is the consistency view seen from the perspective of the time continuum. Two types of temporal consistency are often needed to ensure proper operation of a primary-backup replicated real-time data services system. One is the *external temporal consistency* between an object of the external world and its image on the servers, the other is the *inter-object temporal consistency* between different objects or events.

A primary-backup system is said to satisfy the external temporal consistency for an object i if the timestamp of i at the server is no later than a predetermined time from its timestamp at the client (the real data). In other words, in order to provide meaningful and correct service, the state of the primary server must closely reflect that of the actual world. This consistency is also needed at the backup if the backup were to successfully replace the primary when the primary fails. The consistency restriction placed on the backup may not be as tight as that on the primary but must be within a tolerable range for the intended applications.

The inter-object temporal consistency is maintained if for any object pair, their temporal constraint δ_{ij} (which is the temporal distance of any two neighboring updates for object i, and j, respectively) is observed at both primary and backup.

Although the usefulness or practical application of the *external temporal consistency* concept is easy to see, the same is not true for *inter-object temporal consistency*. To illustrate the notion of inter-object temporal consistency, considering an airplane during taking off. There is a time bound between accelerating the plane and the lifting of the plane into air because the runway is of limited length and the airplane can not keep accelerating on the runway indefinitely without lifting off. In our primary-backup replicated real-time data service, the inter-object temporal consistency constraint between an object pair placed on the backup can be different from that placed on the primary.

5.3. RTPB Implementation

A temporal consistency model for the Real-time Primary-backup (RTPB) replication service has been developed (Zou and Jahanian, 1998) and a practical version of the system that implements the models has been built. Following our composability model, the RTPB service is implemented as an independent user-level x-kernel based server on our MK 7.2 based platform. Our system includes a primary server and a backup server. A client application resides in the same machine as the primary. The client continuously senses the environment and periodically sends updates to the primary. The client accesses the server using a library that utilizes the Mach IPC-based interface. The primary is responsible for backing up the data on the backup site and limiting the inconsistency of the data between the two sites within some required window. The following assumptions are made in the implementation:

- Link failures are handled using physical redundancy such that network partitions are avoided.

- An upper bound exists on the communication delay between the primary and the backup. Missed message deadlines are treated as communication performance failures.

- Servers are assumed to suffer crash failures only.

Figure 11 shows our system architecture and the x-kernel protocol stack for the replication server. The bottom five layers (RTPB to ETHDRV) make up the x-kernel protocol stack. At the top level of the stack is our real-time primary-backup (RTPB) protocol. It serves as an anchor protocol in the x-kernel protocol stack. From above, it provides an interface to the x-kernel based server. From below, it connects with the rest of the protocol stack through the x-kernel uniform protocol interface. The underlying transport protocol is UDP. Since UDP does not provide reliable delivery of messages, we need to use explicit acknowledgments when necessary.

The top two layers are the primary-backup hosts and client applications. The primary host interacts with the backup host through the underlying RTPB protocol. There are two identical versions of the client application residing on the primary and backup hosts respectively. Normally, only the client version on the primary is running. But when the

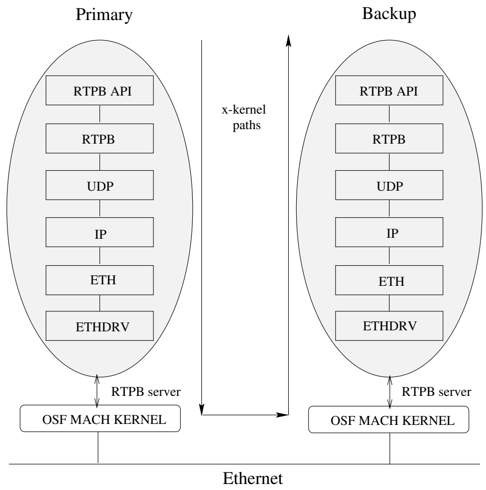


Figure 11. RTPB architecture and server protocol stack

backup takes over in case of primary failure, it also activates the backup client version and bring it up to the most recent state.

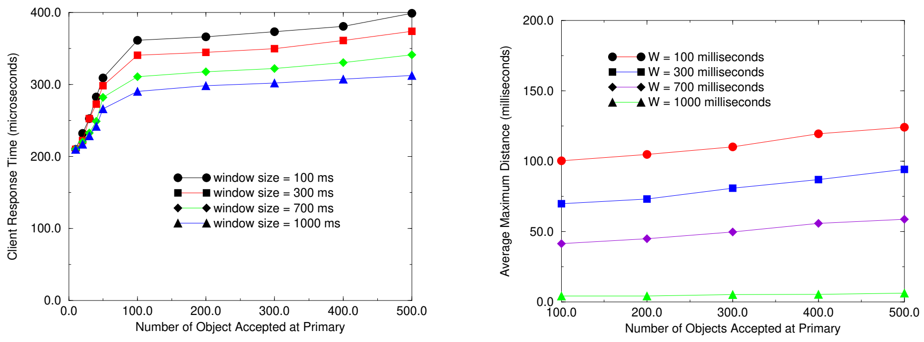
The client application interacts with the RTPB system through the Mach API interface we developed for the system. The interface enables the client to create, destroy, manipulate and query reliable objects (i.e., those backed-up by our server). Specifically, *rtpb_create, rtpb_destroy* creates objects on and destroys objects from the RTPB system; *rtpb_register* register objects with the system; *rtpb_update, rtpb_query* update and query objects; finally *rtpb_list* return a list of objects that are already registered with the RTPB system. Further detail on admission control, update scheduling, failure detection and recovery appears in a recent report (Zou and Jahanian, 1998).

5.4. RTPB Performance

The following graph shows the RTPB response time to client request and the temporal distance between the primary and backup. Both graphs are depicted as a function of the number of objects admitted into the system and are for four different client write rates of 100, 300, 700, and 1000 milliseconds.

Graph (a) shows a fast response time to client request in the range of 200 to 400 microseconds. This mainly due to the decoupling of client request process from updates to the backups. Graph (b) shows that RTPB keeps the backup very close to the primary in terms of the temporal distance between the corresponding data copies of the replicated objects. In the graph, the distance ranges from 10 to 110 milliseconds which is well within the range tolerable by most real-time applications.

The two graphs show that RTPB indeeds provide fast response to client requests while maintain backup(s) very close to the primary in system state.



(a) Response time to client

(b) Primary-backup distance

Figure 12. RTPB performance graphs

6. Evaluation Tools

The third thrust of the ARMADA project is to provide tools for validating and evaluating the timeliness and fault tolerance capabilities of the target system. Two tools have been developed to date: ORCHESTRA, a message-level fault injection tool for validation and evaluation of communication and middleware protocols, and COGENT, a network traffic workload generator. The following two subsections describe the two tools briefly.

6.1. ORCHESTRA

The ARMADA project has been primarily concerned with developing real-time distributed middleware protocols and communication services. Ensuring that a distributed system or communication protocol meets its prescribed specification is a growing challenge that confronts software developers and system engineers. Meeting this challenge is particularly important for applications with strict dependability and timeliness constraints. ORCHESTRA is a fault injection environment which can be used to perform fault injection on communication protocols and distributed applications. ORCHESTRA is based on a simple yet powerful framework, called *script-driven probing and fault injection*. The emphasis of this approach is on experimental techniques intended to identify specific "problems" in a protocol or its implementation rather than the evaluation of system dependability through statistical metrics such as fault coverage (e.g. (Arlat et al., 1990)). Hence, the focus is on developing fault injection techniques that can be employed in studying three aspects of a target protocol: i) detecting design or implementation errors, ii) identifying violations of protocol specifications, and iii) obtaining insights into the design decisions made by the implementors.

In the ORCHESTRA approach, a fault injection layer is inserted into the communication protocol stack below the protocol to be tested. As messages are exchanged between protocol participants, they pass through the fault injection layer on their path to/from the network. Each time a message is sent, ORCHESTRA runs a script called the *send filter* on the message. In the same manner, the *receive filter* is invoked on each message that is received from the network destined for the target protocol. The scripts perform three types of operations on messages:

- **Message filtering:** for intercepting and examining a message.
- **Message manipulation:** for dropping, delaying, reordering, duplicating, or modifying a message.
- **Message injection:** for probing a participant by introducing a new message into the system.

The ORCHESTRA toolset on the MK 7.2 platform is based on a portable fault injection core, and has been developed in the CORDS-based x-kernel framework provided by OpenGroup. The tool is implemented as an x-kernel protocol layer which can be placed at any level in an x-kernel protocol stack. This tool has been used to perform experiments on both the Group Interprocess Communication (GIPC) services from OpenGroup, and middleware and real-time channel services developed as part of the ARMADA project. Further details on ORCHESTRA can be found in several recent reports, e.g., (Dawson et al., 1996, Dawson et al., 1997).

6.2. COGENT*: COntrolled GEneration of Network Traffic*

In order to demonstrate the utility of the ARMADA services, it is necessary to evaluate them under a range of operating conditions. Because many of the protocols developed rely on the communication subsystem, it is important to evaluate them under a range of realistic background traffic. Generating such traffic is fairly difficult since traffic characteristics can vary widely depending on the environment in which these services are deployed. To this end, we've developed COGENT (COntrolled GEneration of Network Traffic). COGENT is a networked synthetic workload generator for evaluating system and network performance in a controlled, reproducible fashion. It is based on a simple client-server model and allows the user to flexibly model network sources in order to evaluate various aspects of network and distributed computing.

Implemented in C++ with a `lex/yacc` front end, the current version of the tool takes a high level specification of the distributed workload and generates highly portable C++ code for all of the clients and servers specified. The user can select from a number of distributions which have been used to model a variety of network sources such as Poisson (Paxson and Floyd, 1994, Paxson, 1994), Log Normal (Paxson and Floyd, 1994), Pareto (Leland et al, 1994, Crovella and Bestavros, 1996, Garret and Willinger, 1994), and Log Extreme (Paxson, 1994). The tool then generates the necessary compilation and distribution scripts for building and running the distributed workload.

COGENT has also been implemented in JAVA. Both the generator and the generated code are JAVA based. Because of the portability of JAVA, this implementation simplifies both the compilation and distribution of the workload considerably. We also plan on addressing

CPU issues in order to model common activities at the end hosts as well. Another feature being added is the ability for a client or a server to be run in trace-driven mode. That is, to run from a web server or a `tcpdump` (McCanne and Jacobson, 1993) log file. Finally, we will be implementing additional source models in order to keep up with the current literature.

7. Conclusions

This paper presented the architecture and current status of the ARMADA project conducted at the University of Michigan in collaboration with the Honeywell Technology Center. We described a number of communication and middleware services developed in the context of this project, and illustrated the general methodology adopted to design and integrate these services. For modularity and composability, ARMADA middleware was realized as a set of servers on top of a microkernel-based operating system. Special attention was given to the communication subsystem since it is a common resource to middleware services developed. We proposed a general architecture for QoS sensitive communication, and also described a communication service that implements this architecture.

We are currently redesigning an existing command and control application to benefit from ARMADA middleware. The application requires bounded time end-to-end communication delays guaranteed by our communication subsystem, as well as fault-tolerant replication and backup services provided by our RTCAST group communication and membership support, and the primary-backup replication service. Testing tools such as ORCHESTRA will help assess communication performance and verify the required communication semantics. Controlled workload generation using COGENT can assist in creating load conditions of interest that may be difficult to exercise via regular operation of the application.

Our services and tools are designed independently of the underlying microkernel or the communication subsystem; our choice of experimentation platform was based largely on the rich protocol development environment provided by x-kernel and CORDS. For better portability, we are extending our communication subsystem to provide a socket-like API. We are also investigating the scalability of the services developed. Scaling to large embedded systems may depend on the way the system is constructed from smaller units. We are looking into appropriate ways of defining generic structural system components and composing large architectures from these components such that certain desirable properties are globally preserved. Developing the “tokens” and “operators” of such system composition will enable building predictable analytical and semantic models of larger systems from properties of their individual constituents.

Notes

1. Open Group is formerly known as the Open Software Foundation (OSF)
2. A lower communication layer may support a bounded number of retransmissions.
3. The APIs for both the service library and the RTCAST protocol are available at http://www.eecs.umich.edu/RTCL/armada/rtcast/api.html.

References

Tarek Abdelzaher, Anees Shaikh, Scott Dawson, Farnam Jahanian, and Kang Shin. Rtcast: Lightweight multicast for real-time process groups. in submission, available at http://www.eecs.umich.edu/RTCL/armada/rtcast/, 1997.

Tarek Abdelzaher, Anees Shaikh, Farnam Jahanian, and Kang Shin. RTCAST: Lightweight multicast for real-time process groups. In *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 250–259, Boston, MA, June 1996.

Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. Technical Report TR CS91-13, Dept. of Computer Science, Hebrew University, April 1992.

Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for continuous media in the DASH system. In *Proc. Int'l Conf. on Distributed Computing Systems*, pages 54–61, 1990.

Jean Arlat, Martine Aguera, Yves Crouzet, Jean-Charles Fabre, Eliane Martins, and David Powell. Experimental evaluation of the fault tolerance of an atomic multicast system. *IEEE Trans. Reliability*, 39(4):455–467, October 1990.

Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

F. Cristian, B. Dancy, and J. Dehn. Fault-tolerance in the advanced automation system. In *Proc. of Fault-Tolerant Computing Symposium*, pages 6–17, June 1990.

Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *SIGMETRICS '96*, May 1996.

Rene Leonardo Cruz. *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks*. PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU–ENG–87–2246.

Scott Dawson, Farnam Jahanian, and Todd Mitton. Experiments on six commercial tcp implementations using a software fault injection tool. to appear in *Software Practice & Experience*.

Scott Dawson, Farnam Jahanian, and Todd Mitton. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *International Symposium on Fault-Tolerant Computing*, pages 404–414, Sendai, Japan, June 1996.

Domenico Ferrari and Dinesh C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

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

Mark Garrett and Walter Willinger. Analysis, modeling and generation of self-similar vbr video traffic. In *SIGCOMM '94*, pages 269–280, 1994.

Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):1–13, January 1991.

D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1044–1056, October 1994.

Hermann Kopetz and Günter Grünsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

Will Leland, Murad S. Taqqu, Walter Willinger, and Daniel Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

Steve McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, January 1993.

Ashish Mehra, Atri Indiresan, and Kang Shin. Structuring communication software for quality of service guarantees. In *Proc. 17th Real-Time Systems Symposium*, pages 144–154, December 1996.

Ashish Mehra, Anees Shaikh, Tarek Abdelzaher, Zhiqun Wang, and Kang G. Shin. Realizing services for guaranteed-qos communication on a microkernel operating system. In *Proc. Real-Time Systems Symposium*, Madrid, Spain, December 1998.

S. Mishra, L.L. Peterson, and R.D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

Vern Paxson. Empirically-derived analytic models of wide-area tcp connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, August 1994.

Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. In *SIGCOMM '94*, pages 257–268, August 1994.

R. van Renesse, T.M. Hickey, and K.P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR94-1442, Dept. of Computer Science, Cornell University, August 1994.

Hengming Zou and Farnam Jahanian. Real-time primary backup (RTPB) replication with temporal consistency guarantees. In *Proceedings Intl. Conf. on Distributed Computing Systems*, pages 48–56, Amsterdam, Netherlands, May 1998.