Constructing Real-time Group Communication Middleware
Using the Resource Kernel *

Scott Johnson and Farnam Jahanian
University of Michigan
Dept. of EECS
Ann Arbor, MI 48109-2122
{scottdj,farnam}@eecs.umich.edu

Akihiko Miyoshi, Dionisio de Niz, and Ragunathan Rajkumar
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213-3891
{miyos+, dionisio+, raj+}@cs.cmu.edu

Abstract

Group communication is a widely studied paradigm which is often used in building real-time and fault-tolerant distributed systems. RTCAST is a real-time group communication protocol which has been designed to work with commercial, non-real-time, off-the-shelf hardware and operating systems, such as Solaris, Linux, and Windows NT. RTCAST makes probabilistic real-time guarantees based on assumptions about the performance of the underlying system. Unfortunately, the high variability of the access to system resources that these operating systems provide may limit the predictability of the real-time guarantees provided by RTCAST. By taking advantage of a service that provides resource scheduling and reservation in these operating systems, both the hardness and timing granularity of RTCAST's real-time services can be greatly improved. This paper describes an implementation of RTCAST which makes use of the Resource Kernel to provide highly predictable, real-time communication guarantees.

Keywords: group communication, distributed systems, real-time middleware, resource management

1. Introduction

This paper describes an effort to provide predictable, real-time communication guarantees with RTCaste, a group communication protocol designed at the University of Michigan, which runs on several widely available off-the-shelf operating systems and hardware. Process groups are a widely-studied paradigm for designing dependable distributed systems in both asynchronous [24, 3, 18] and synchronous [15, 4, 9] environments. In this approach, a distributed system is structured as a group of cooperating processes using two key primitives: group membership and fault-tolerant multicast communication. It has been argued that process groups are particularly effective for managing the complexity of large applications and for providing dependability and timeliness guarantees in the presence of faults. 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 potentially anonymous recipients.

Providing these services is more complicated in distributed real-time applications which operate under strict timing and dependability constraints. In particular, we are concerned with fault-tolerant real-time systems which must perform multicast communication and group management activities in a timely fashion, even in the presence of faults. In these systems, multicast messages must be received and handled at each process by their stated deadlines. In addition, membership agreement must be achieved in bounded time when processes in a group fail or rejoin, or when the network suffers an omission or communication failure.

RTCasting was designed to enforce probabilistic real-time guarantees based on estimates of system resource availability. Because RTCAST depends on the operating system for access to system resources (CPU, disk, network), the timeliness of its real-time guarantees will be limited by the predictability of the resource scheduling provided by the OS. Process scheduling delays, CPU load, and background network traffic can all cause the service to violate its assumptions about available bandwidth and computation delay, sometimes resulting in missed deadlines. None of the widely-used operating systems mentioned above provide any kind of guarantees on process scheduling delay, CPU allocation, or bandwidth allocation to processes. This greatly limits the hardness of the real-time guarantees that RTCAST can provide to applications, and increases the probability of deadline violations for guarantees that it does make. It also increases the failure-detection latency, since

---

*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.
failure detection timeouts must be increased to account for variability in scheduling.

We could improve the timeliness of RTCAST's real-time services if we could ensure that it had predictable access to system resources. The Resource Kernel [19, 20] is a resource-centric approach for building operating system kernels that provides timely, guaranteed and enforced access to system resources. It is designed modularly, so it can be easily integrated into non-real-time operating systems to add real-time support. By modifying RTCAST to use the resource reservation services provided by the Resource Kernel, we were able to greatly increase the predictability of RTCAST, which allowed us to increase both the timeliness of its real-time guarantees and reduce the latency of some of its other services such as failure detection.

We believe that this is the first effort which attempts to integrate these two distinct research areas. By taking advantage of the guarantees provided by resource management, we believe that real-time communication middleware can realize significant improvements in predictability and timeliness without having to rely on dedicated real-time operating systems or hardware. In this paper, we show how we were able to use resource reservations with RTCAST to make real-time guarantees on group message delivery and failure detection without otherwise modifying the RTCAST protocol. We also present a timing analysis of the protocol under resource reservation, and calculate predictable upper bounds on message queuing delay, timeout latency, and network access. Finally, we quantify the improvements in predictability and timeliness of the protocol on both loaded and unloaded systems with a variety of performance tests. As these experimental results indicate, by combining the regulated access to system resources provided by the Resource Kernel with the predictable operation of the RTCAST protocol, we can provide accurate, real-time communication guarantees which are enforced despite high system load or resource contention from other applications.

The remainder of this paper is organized as follows. In Section 2, we present an overview of the RTCAST protocol. In Section 3 we discuss the operation of the resource kernel and the resource reservation services it provides. Section 4 shows how we integrated RTCAST with the resource kernel to meet our real-time requirements and presents a timing analysis of the integrated protocol. Section 5 presents the results of a performance study showing how the resource kernel improved RTCAST's real-time performance, and Section 6 summarizes our results.

2. RTCAST

There are many group communication protocols which have been proposed for use in asynchronous distributed systems. Non-real-time protocols include TTP [15], which requires special hardware support, and XPA [26]. Other approaches including the fail-awareness framework [11], quasi-synchronous model [25], and Cactus [12] are based on the best effort paradigm. These systems allow a set of processes to communicate using various delivery semantics such as FIFO, causal, or total ordering while providing a consistent membership view of participants believed to be accessible. In addition, the real-time protocols attempt to add timeliness guarantees on communication and failure detection.

RTCAST is a real-time group communication protocol that provides atomic, totally ordered communication with real-time guarantees[1]. It supports deadline-based message delivery, and timely detection of failed processes. It is designed to run on off-the-shelf, non-real-time operating systems and hardware, and has been implemented for Solaris, Linux, Windows NT, and OpenGroup MK. It is implemented on top of the IP protocol, and sends messages using broadcast or IP multicast if available. If neither of these services is available, RTCAST can resort to multiple point-to-point transmissions to send each message to the group.

The RTCAST protocol architecture is shown in Figure 1. RTCAST has been used to implement a real-time distributed radar tracking and hypothesis testing application [13], in collaboration with Honeywell Technology Center. In general, it supports the construction of distributed real-time applications which require timely multicast communication or timely process membership and failure detection. The full details of RTCAST, including performance analysis and a full comparison to other group communication protocols, are available in [2]. In this section, we present an overview of RTCAST's design and services, which will be used in Section 4 to demonstrate the issues involved in using resource reservations to realize real-time guarantees.

2.1. Basic RTCAST Operation

RTCAST utilizes a logical token ring algorithm to regulate access to the network (Figure 2). Processes are or-
Figure 2. The logical token ring

Figure 3. Relationships between the Resource Kernel and RTCAST threads

message reception, timeouts (for example the token timeout), and application requests. Each event handler is implemented as a separate thread which services events of a particular type.

There are two primary event handling threads in RTCAST which have real-time requirements (Figure 3). The first is the thread responsible for receiving messages from the network. This receive thread is responsible for copying incoming messages into a protocol buffer, and performing the message ordering and failure detection checks. This thread is also responsible for sending queued messages during the token possession, and sending the heartbeat message to pass the token to the next process.

The second thread is the thread which handles timeouts, such as the token timeout which is used to detect failures in the token ring protocol. RTCAST implements a timeout service where registered timeouts are placed in a queue. A timeout thread is responsible for periodically checking the queue and executing the event handler for any timeout which has expired. This timeout service is implemented using the nano.sleep() call, which supports per-thread blocking and fine-grained waiting periods.

RTCAST is structured as a library which is linked to the application. This means that each process in an RTCAST group runs its own instance of the protocol stack and operates independently of other RTCAST processes, even ones residing on the same physical host. We also lock all pages of the RTCAST protocol in physical memory using the mlockall() system call to prevent current and future pages of the process from being swapped out, which could result in unpredictable processing delays when they are next referenced.

In the following section, we describe the resource kernel which we used to provide predictable resource access for RTCAST. In Section 4, we show how we used its services to meet RTCAST’s real-time requirements.
3. Resource Kernel

The key philosophy behind the resource kernel is that precise timing guarantees and temporal protection between applications can be obtained by imposing a well-defined resource usage model on time-multiplexed resources. In other words, an application running on a resource kernel can request the reservation of a certain amount of a resource, and the kernel can guarantee that the requested amount is exclusively available to the application.

Reservations enable applications to make predictable progress under arbitrary system load. For example, a movie player application without resource reservations may not be able to provide an ideal frame rate because of other competing activities demanding resources. This often results in jitter and loss of frame rate in the movie. By making reservations, the movie application can provide the desired frame rate even under heavy system load.

The resource kernel has already been described in detail elsewhere [19, 20]. We present a brief overview of its services here to aid in discussing our integration efforts. We also present a set of experiments to evaluate the jitter an application can experience. Finally, we provide a brief description of other projects which provide similar services, and which might also be used to support the implementation of real-time group communication middleware.

3.1. Linux/RK Overview

Implementation of CPU reservation on Linux/RK includes the following operations: admission control, scheduling, enforcement, and accounting. When an application makes a request for CPU reservation, admission control determines whether it can be accepted or not based on the requirements imposed by the scheduling policy. If the request can be admitted, a reserve based on the requested parameters is created. Once an application is granted a reservation by admission control, it is the job of the scheduler to ensure the availability of that resource.

To enforce reservations, Linux/RK adjusts priority levels and suspends and resumes processes based on their reservation parameters. Context switches are used to start/stop the accounting mechanism to support the replenishment and enforcement.

3.2. Jitter Measurements

For the jitter experiments we assumed processes that take an input, process it and generate an output in each activation as in [14]. The jitter of three latencies were measured: Input to input, input to output, and output to output. To measure the variations of these latencies we use three categories: Single reserve, multiple reserve with the same period, and multiple reserve with different periods.

We use an empty loop as the workload of the threads to reduce the jitter from other sources. For our test cases we used a common computation time (C) of 1 ms and four different periods: 10 ms, 100 ms, 500 ms, and 1 second.

Figure 4 shows the results for the single reserve. As you can see in the figure, the percentage of jitter increases as the period decreases. Figure 5 shows the results for multiple reserves with a single period. In both tests with multiple reserves, we observed the start-to-end period can go up to several times the C time. This happens when the activation of some of the reserves are delayed. Then, in the middle of its activation, the replenishment mechanism replenishes only the used part of its reserve; leading to a subsequent activation for only this replenished amount and completing at the next replenishment.

Finally, in the case for the multiple reserves with different periods (not shown) the jitter will accumulate in the larger periods giving them a larger variation.

3.3. Related Work

 Numerous systems have addressed different aspects of reservation-based systems. An early approach to this
paradigm can be found in the Spring kernel [23] and Maruti [17].

In addition to Linux/RK, there are other systems which provide real-time guarantees using existing operating system kernels. Real-Time Linux [5] is a combination of a simple real-time executive and patches to the Linux kernel to make it run on the executive. KURT [22] provides firm real-time functionality, which is defined as a hard timing requirement with a soft deadline. KURT features a high-resolution timer as in RT-Mach RK to satisfy hard timing requirements. Neither of these Linux real-time extensions provides resource kernel features, such as guaranteed reservation and enforcement.

There are several commercially available real-time extensions to Microsoft Windows NT. Their approaches are similar to RT-Linux. Typically, a real-time executive is buried at the bottom of the NT kernel and provides real-time functionality with real-time tasks. Thus, real-time tasks cannot utilize NT's services. DREAMS [21] is a user-level real-time extension to Microsoft Windows NT. It took a similar approach to ours for controlling applications' resource utilization, but its capability is more limited because of its user-level implementation. The EPIQ project at the University of Illinois at Urbana-Champaign has created an NT implementation of an open framework which supports deadline-based process scheduling [10].

Other projects which have similar goals to the resource kernel include the HARTIK kernel from Italy and the Nemesis OS at Cambridge. The HARTIK kernel [16] emphasizes variants of earliest deadline scheduling in admission control and scheduling, while the Nemesis OS [7] focuses on building a clean implementation from scratch while minimizing interference arising from application interaction.

4. Integration of RTCast with the Resource Kernel

Now that we have described the services provided by the resource kernel, we will show how they can be used to meet the real-time requirements of RTCast. RTCast requires regular access to two system resources: CPU time and network bandwidth. RTCast needs CPU time to process messages for transmission and reception, to check incoming message headers for proper ordering, to detect process failures, and to perform membership changes. RTCast requires regular access to the network medium so it can send and receive messages in a timely fashion.

For our implementation, we selected the Linux/RK version of the Resource Kernel, since we already had a Posix compliant RTCast implementation that would run on Linux. This also allowed us to build the system using low-cost, widely available hardware. The Linux/RK implementation of the Resource Kernel supports CPU reservations, as described in Section 3. Unfortunately, it does not currently implement reservations for network bandwidth. However, support for this is planned, and we plan to utilize this service for RTCast once it is available. In the meantime, our implementation uses RTCast's logical token ring protocol to regulate access to the network, with the added requirement that no non-RTCast processes may generate network traffic.

In the remainder of this Section, we describe how we integrated the CPU reservation mechanism of Linux/RK with RTCast to implement its real-time guarantees. We also present some real-time software engineering issues which were raised by RTCast's integration with the resource kernel, and which we believe may be of use in the construction of other real-time middleware.

4.1. Real-time Requirements of RTCast

As we described in Section 2, there are two protocol threads which have real-time requirements: the receive thread, and the timeout handler thread. In order to make reservations for these threads, we first need to determine their periods and CPU time requirements.

4.1.1. RTCast Thread Periods

The receive thread can be treated as a periodic task with a period \( T_{\text{en}} \) equal to the smallest token hold time of any process in the ring, denoted \( T_{\text{min}} \). This ensures that the thread has sufficient time to process all messages from each member before the next one starts transmitting. For our performance study, each process had a token hold time of 20 ms.

The timeout thread's period, \( T_T \), is the length of time it waits before checking the queue for new timeouts. We chose a period of 20 ms, which provides a short latency for executing expired timeouts without adding too much overhead to the CPU requirements of the protocol.

4.1.2. RTCast Thread Execution Times

In order to determine the CPU time requirements of these threads, we made use of the resource kernel's profiling service. This service uses Linux's /proc filesystem to report information about running threads which have CPU reservations. It reports both maximum and average CPU time.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Period</th>
<th>Avg. CPU</th>
<th>Max. CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive Thread</td>
<td>( T_{\text{en}} )</td>
<td>4.5 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>Timeout Thread</td>
<td>20 ms</td>
<td>0.05 ms</td>
<td>2 ms</td>
</tr>
</tbody>
</table>

Table 1. RTCast per-thread execution time requirements
consumed each period, which we used to determine the execution times for the threads under varying operating conditions such as group size and message load.

The CPU reserve required for the receive thread, denoted by $C_R$, is directly proportional to its period, since the maximum number of messages it may have to process is proportional to the token hold time. Based on this profiling we determined that in the worst case, when every group member is sending a constant stream of messages and using a very short token hold time, the receive thread required a maximum of 9 ms of CPU time for a period of 20 ms, and 4.5 ms on average. Note that this is a much higher CPU utilization than would be generally experienced, since in an average system most processes would not be sending large numbers of messages on every token rotation, and the token hold time could be longer. Under the resource kernel, threads that are blocked or sleeping are not eligible to be scheduled, and other threads can use the CPU until the blocked threads are ready to run. Therefore, unused CPU reserves are not wasted but can be utilized by other threads. By reserving the maximum CPU time required by the receive thread, we can ensure that it can meet its requirements in the worst case while not wasting CPU resources the rest of the time. In the future, we plan to conduct additional testing to determine a formula for the dynamic computation requirements of the receive thread, based on group size and presented load. However, such a formula would be very specific to RTCAST and may not be applicable to other real-time communication protocols.

Unlike the receive thread, the timeout thread only requires a very small CPU reserve, $C_T$, since most of the timeout handlers it runs are small. We found that it used at most 2 ms of CPU time for each 20 ms period, and only 0.05 ms on average. Most of the time it requires almost no CPU time, since it merely checks the current system time and compares it to a small number of timeout values before sleeping again. A summary of the thread timing requirements is presented in table 1.

Based on our profiling of the receive thread, we discovered a more optimal way to structure the system. One of the reasons that the receive thread's CPU utilization is so high is that it was responsible for both receiving messages from the network, and when a token arrived performing all of the failure checking and message transmission. However, the latter task has a much longer period than that required for receiving messages - the full token rotation time, $T_{token}$, as opposed to the shortest token hold time, $T_{min}$. Therefore, by placing the latter functionality into a new thread, called the token thread, we can reduce the CPU requirements of the receive thread and more efficiently guarantee RTCAST’s CPU requirements. One caveat however: if we simply gave the token thread a period equal to the token rotation time, it could be executed at any point during that period. This is undesirable, since it needs to run immediately after the token arrives so it can send all of its messages by the end of the token hold time. To avoid this problem, we used the resource kernel’s `rk_super_reserve()` call to set this thread to execute at a higher priority, which precedes all other reservations. This will ensure that the thread will be scheduled to run for it’s full CPU reserve as soon as it is ready (i.e. right after the token arrives).

After making this change, we profiled RTCAST again, and found that the CPU requirements of the receive thread were now a maximum of 7.5 ms with a period of 20 ms and a lower average of 3.3 ms. The new token thread had a maximum CPU requirement of 3.7 ms with an avg. of just 1.01 ms, and a period equal to $T_{token}$. This reduced the total CPU reservation for RTCAST, leaving more available for other real-time reservations. Once again, note that in a production system, the periods of these reservations would likely be longer, resulting in a lower overall utilization. The remainder of this paper assumes the two thread implementation for its analysis.

4.2. Implementation of the Integrated System

The most difficult part of integrating RTCAST with the resource kernel was determining the threads’ timing requirements. Once this was done, the actual code modifications required for RTCAST to use the resource kernel services were very minor. It required the addition of 20 lines of code for each thread, in order to initialize and register the CPU reservation with the Resource kernel. For verification purposes, we also wrote a program to monitor the status of each reservation at run-time, and give an alert if a thread overran its reservation.

4.3. Integration Timing Analysis

In addition to the timing of the thread reservations in the resource kernel, there were a few other timing issues which affected the real-time performance of RTCAST. As we mentioned above, the timeout thread uses the `nano_sleep()` function to sleep when it’s not checking the event queue for expired timeouts. The timeout thread calls this function with a sleep duration of 20 ms. However, due to operating system overhead and scheduling granularity, it is quite possible that the thread won’t be run immediately when the 20 ms expires. The OS kernel won’t place the thread in the runnable state until the next time its process scheduler runs, which occurs with period $W_S$. In Linux, $W_S$ is equal to 10 ms. Since the resource kernel will only schedule threads that are in the runnable state, this may result in an additional delay of up to $W_S$ after the timer expires. In our profiling, we found that the timeout thread was frequently invoked anywhere from 3 to 10 ms after the exact timeout value. With non-real-time operating systems, there could also be an arbitrarily long additional delay while the operating system scheduled other processes to run.
Another timing issue involved the receive thread. This thread uses the select() system call to wait for the arrival of messages from the network. As with nano.sleep(), it is possible for the operating system to invoke this thread up to $W_S$ later than the actual arrival time of the message due to the process scheduling latency. This time must also be taken into account by RTCAST when determining propagation delay for schedulability analysis, and calculating the token timeout.

These problems could be mitigated by a real-time timer service, which the resource kernel does not currently provide. Both the nano.sleep() and select() calls use the standard Linux kernel services, which the resource kernel has no control over. Even if it did provide a timer service, it would probably have a similar scheduling latency, since using a shorter scheduling granularity results in higher CPU overhead. However, when using the resource kernel, a thread can at least be guaranteed to get access to the CPU within one scheduling period of waking up from the timer, which provides a hard upper bound on the latency associated with using one of these blocking calls. Without the resource kernel, this delay could be arbitrarily long, since the OS could continue to schedule other processes ahead of the real-time threads even after they were ready to run.

4.4. Effective Timeout Values

Using these upper bounds on thread execution with CPU reserves, we can now calculate the worst-case token timeout $T_{token}$ and propagation delay $\Delta$ that RTCAST must use when detecting failures and scheduling real-time messages. We assume that each process $P_i$ in a group of size $S$ has a token hold time of $T_i$. The worst-case delay $D_R$ for a receive thread with a maximum CPU requirement of $C_R$ each period would be

$$D_R = W_S + W_T - C_R$$

since the thread could be delayed up to $W_S$ by the Linux kernel, and might then be run at the end of its next period, which could happen at the latest after a delay of $W_T - C_R$. Assuming the propagation delay is some small value $\delta$, the schedulability analysis should use $\Delta = D_R + \delta$ for the propagation delay when making admission control decisions. In our test implementation, $W_R = 20$, $W_S = 10$, and $C_R = 9$, which yields an effective value for $\Delta = 21 + \delta$ ms.

Similarly, the worst-case delay $D_T$ between a timer’s expiration and its event handler’s execution would be

$$D_T = W_S + 2 \cdot W_T - C_T$$

since the timeout thread sleeps for $W_T$ time units, might take up to $W_S$ time units to be marked runnable by the Linux kernel, and then might not be run for an additional $W_T - C_T$ time units by the resource kernel. In our test implementation, with $W_T = 20$ and $C_T = 2$, $D_T = 48$ ms. This yields a maximum token rotation time $T_{token} = D_T + \sum_{i=1}^{S}(T_i + \Delta)$. This is also the maximum failure detection latency, since if a process failed immediately after the end of its token possession, it would take a full token round before it would be expected to send another heartbeat. This also gives the maximum message queuing delay, which would occur if the application submitted a message right after it had given up the token, and that message was the last one sent on the following token round. Note that this time can be shortened by reducing the timeout thread’s reservation and sleep periods to the minimum of 10 ms each.

5. Experimental Results

After integrating RTCAST with the resource kernel as described in previous sections, we conducted a thorough performance evaluation to examine the effect of resource reservations on the predictability and end-to-end latency of the protocol. Our main goal was to determine whether using the resource kernel would improve RTCAST’s ability to meet message deadlines and perform timely failure detection. As our results show, using CPU reservations from the resource kernel had a significant impact on the timeliness of RTCAST, and greatly improved its ability to provide predictable real-time guarantees to the application.

To measure the system’s performance, we created a simple distributed application which used the standard RTCAST interface to send and receive messages. Communication within the test application is organized into rounds. At the start of a round, each process uses RTCAST to send a fixed number of messages to the group. Each process then receives all of the messages that were sent that round before proceeding to the next round. Messages are sent using RTCAST’s best-effort service, in order to evaluate the ac-
tual communication performance of the underlying middleware without interference from RTCAST's real-time message scheduling algorithm.

Since messages are not queued between rounds, this eliminates most of the receiver-side queuing delay, leaving only the latency of sending the messages through the RTCAST protocol itself. Messages are timestamped at the sender and receiver, and the one-way latency is calculated for each message. To ensure the accuracy of the measurement, we used a software-based clock synchronization protocol to synchronize the hardware clocks of the machines before each test. The physical hardware consisted of 8 Pentium III machines running at 500 MHz. Each machine had 128 MB of RAM, and they were connected using a 100BaseT Ethernet LAN and Intel Express 330T hub.

Using the test application, we conducted four experiments using group sizes from one to seven. For each experiment, we ran the test application either with or without background CPU load, and with or without the CPU reservations provided by the resource kernel. In these experiments, the test application performed 60 rounds of communication, with each process sending 10 messages per round. The application was run 10 times for each configuration and the results from all runs were used to calculate the final performance statistics. CPU load was generated by spawning 15 additional processes which spun in tight loops. For tests with CPU reservations, RTCAST used the two thread reservation implementation described in section 4, and the application thread also used a reservation to ensure that it would have adequate CPU time to send and receive messages and process the timestamps in a timely fashion.

Figure 6 shows a graph of the average message latency observed for each test configuration. We observed no significant difference in average latency between the various configurations, indicating that the resource kernel does not add significant overhead to the execution of the system. It may seem surprising that tests which didn't use CPU reservations achieved similar average performance to those which did. Recall, however, that the main purpose of the resource kernel (and in fact any real-time scheduling service) is to improve predictability and provide timely access to system resources. Even when running without resource reservations, the RTCAST threads can usually get enough CPU time to complete their tasks in a timely fashion. As we shall show, the real benefit to using the resource kernel is that it reduces the number of transient scheduling delays, resulting in fewer late messages and missed deadlines when resource reservations are used.

We also calculated the standard deviation of the message latencies for all test cases. As with average message latency, there is no significant difference in the standard deviation with or without resource reservations. A typical set of values for the standard deviation is listed in Table 2.
<table>
<thead>
<tr>
<th>Group Size</th>
<th>No RK/Loaded</th>
<th>All Other Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>261</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>320</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>448</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>571</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3. Late messages observed

We do see a significant difference in performance, however, if we look at the maximum observed message latencies. As Figure 7 shows, using the resource kernel ensures that real-time threads receive timely, periodic access to the CPU. Threads on a loaded system without reservations see transient delays or overload conditions which cause them to miss deadlines or fail to perform in a timely fashion, with observed delays reaching into the many hundreds of milliseconds. Using the resource kernel, RTCAST is able to provide timely message delivery even in presence of high system load.

A more detailed comparison of average to maximum message latency is shown in Figures 8 and 9. From Figure 8, we can see that although increasing the system load does have an impact on performance, CPU reservations enable RTCAST to maintain a predictable upper bound on message delay. In contrast, Figure 9 shows that without resource reservations, system load can cause extremely large delays in performance, which result in missed deadlines and also require the protocol to employ a higher failure detection timeout to account for the possible delay.

The observed maximum message latencies using the resource kernel are well within the maximum predicted in Section 4. We believe this indicates that the worst-case latencies predicted by the timing analysis are too conservative compared to real-world performance, but we will need to perform additional experiments to determine a more appropriate upper bound on the timeliness of the system. Without resource reservations messages can exceed this worst-case prediction by a significant amount, as shown in the figures above. Table 3 shows an exact count of messages which missed this time bound for each of the tests. The only case in which messages missed the worst-case bound is with a background CPU load and no resource reservations.

6. Conclusion

In this paper, we described how we were able to use the Resource Kernel and the RTCast group communication protocol to provide predictable group communication performance with real-time guarantees. Building on the Resource Kernel’s CPU reservation service, RTCast was able to achieve more predictable communication performance and lower failure detection latencies on both loaded and unloaded systems. We were also able to calculate an upper bound on the token rotation and message propagation times, taking into account the maximum process scheduling latency provided by the CPU reservation. This predictability was verified through performance tests, which showed that without CPU reservations, RTCast would experience scheduling delays and slowdowns, resulting in late messages and high failure detection timeouts.

In the future, we hope to also make use of periodic network bandwidth reservations, which are not currently provided in the Linux implementation of the Resource Kernel. This would enable RTCast to provide its guarantees even in the presence of external network traffic. We also hope to do a more extensive profiling of RTCast to be able to more accurately determine the processing requirements of its threads based on dynamic traffic load. In addition, we would like to conduct further experiments to determine a less conservative upper-bound for the token rotation time and propagation delay. Our performance results indicate that the theoretical maximums may occur infrequently enough that lower bounds could be used in practice, resulting in faster operation of the protocol and a lower failure detection latency.

References


