Distributed Resource Kernels: OS Support for End-To-End Resource Isolation

Karthik Lakshmanan and Raj Rajkumar
Real-Time and Multimedia Systems Laboratory
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213, USA
lakshmanan@cmu.edu, raj@ece.cmu.edu

Abstract

The notion of resource reservation for obtaining real-time scheduling guarantees and enforcement of resource usage has gained strong support in recent years. However, much work on resource reservation has primarily focused on single-processor systems. In this paper, we propose the distributed resource kernel framework to deploy distributed real-time applications with end-to-end timing constraints, and to efficiently enforce and monitor their usage. Modern distributed real-time systems host multiple applications, where each application can span two or more processors. Timing bugs in one distributed application can affect the timing properties of other applications in the system. Our framework introduces the abstraction of a distributed resource container as an isolated virtual operating environment for a distributed real-time application. We have implemented this framework by extending our open-source single-node Linux/RK platform [15]. A deployment and monitoring tool called dMon is also provided. We evaluate the framework's ability to provide timing guarantees by stress-testing the system using the Distributed Hartstone benchmarks. An audio processing pipeline is then used to illustrate the temporal isolation support provided by the Distributed RK framework. The distributed container abstraction can also be extended in the future to support security and fault-tolerance attributes.

1. Introduction

Distributed real-time systems are widely employed in many networked cyber-physical applications like telepresence, defense systems, process control, factory automation, motion control and multimedia systems. Many of these systems are often shared by multiple (often independently developed or off-the-shelf) tasks with varying time requirements and resource demands. Some of these requirements represent hard deadlines, while others do not. The necessity of sharing resources in a distributed environment leads to contention and can lead to serious problems in meeting these deadlines. Providing deadline guarantees is therefore crucial for the successful deployment of such systems.

Modern automobiles have several tens of electronic control units (ECUs)[2] and evolving applications like drive-by-wire are expected to add more functionality and complexity. Financial systems have hundreds of nodes hosting a large number of tasks with varying characteristics. In such large-scale systems, intricate timing dependencies get introduced between different sub-systems. When an individual component fails, it normally takes down shared resources like processors, memories and network links, thereby indirectly affecting the other components. In order to avoid this problem, distributed real-time systems should have built-in support for temporal isolation.

Multimedia systems like audio and video conferencing applications are an important category of distributed real-time systems. Audio streaming has stringent jitter requirements while video streaming needs large bandwidth. In such systems, it would be useful to aggregate the requirements corresponding to each end-to-end flow separately into higher-level entities. These abstractions can simplify the orchestration of high-level resource management policies for the entire system.

Practical solutions for building large-scale distributed real-time systems must also support remote management of distributed resources and dynamic adaptation. Global resource management decisions can be undertaken at administrator nodes, while fine-grained scheduling decisions can be made locally. In order to dynamically make adaptive resource management decisions, the underlying system should efficiently support monitoring of resource utilization levels and provide aggregation of status information from individual nodes. Deployment utilities that allow easy configuration and automatic deployment of distributed applications will also prove to be useful.

In this paper, we present the distributed resource kernel framework to support the requirements of large-scale distributed real-time systems hosting multiple different applications, and develop the distributed RK framework to efficiently develop, debug, deploy, isolate, monitor and maintain applications in such systems.

1.1. Related Work

Resource reservation (sometimes also called “separation kernels” or “resource partitioning”) has become a popular approach for developing predictable and enforceable real-time systems. Operating systems like Linux/RK [5] and QLinux [4] support performance isolation at the level of individual applications. The Real-Time Specification for Java [35] also includes support for specifying CPU requirements for RT-Java threads. The latest version of Microsoft Windows, Vista, supports a version of CPU reservation for time-sensitive multimedia data [34]. Virtual machines like Xen [3] provide resource isolation at operating system granularity. All these systems,
however, focus on single-processor systems\(^1\). Our distributed RK framework extends this paradigm to provide support for end-to-end resource isolation for distributed real-time applications.

Distributed (non-real-time) operating systems have been well-studied in the literature. Many research systems like Amoeba [6], Locus [7], COSY [8] and Clouds [9] have established powerful primitives such as RPC, threads and group-communication that are required to develop such systems. Communication infrastructures like MPI [21] and OpenMP [26] have been developed for enabling parallel-computing systems. However, these systems do not include support for resource reservation and temporal isolation.

Many middleware-based approaches support end-to-end resource management in distributed systems [19, 20]. Priority-driven preemptive scheduling, priority-based queuing of messages and some form of priority inheritance constitute the key features of these systems. In these systems, one misbehaving high-priority application can seriously disrupt other applications. Our approach offers temporal isolation between all applications. In addition, we adopt a kernel-level implementation to (a) avoid the assumption that all applications cooperate and adhere to a common middleware interface, and (b) obtain fine-grained control and efficient performance. Some of the services provided in our framework, like the global time service, also benefit greatly from this kernel-level approach.

In the networking community, resource reservation protocols like RSVP [22] are popular for providing QoS guarantees over the Internet. Our idea of performing distributed admission control is similar to RSVP. However, RSVP is restricted to managing the network resource, whereas our framework has a broader responsibility of managing all distributed system resources.

Distributed real-time scheduling approaches have been investigated since early distributed real-time operating systems like the Spring Kernel [33]. Other systems like ARTS [24] have adopted fixed-priority schemes for real-time scheduling in the distributed environment. Holistic schedulability analysis for such fixed-priority distributed real-time systems have also been developed [28]. In such systems, however, “back-to-back” execution-offsets that disrupt periodic arrival/execution patterns can arise. Therefore, solutions like Period Enforcers [18] and Release Guards [29] must be adopted. Our OS framework integrates these constructs in a user-transparent fashion, and employs a custom schedulability test that integrates computational deadlines, network latencies, time-synchronization error and arrival phasing. This test is specific for a distributed task graph model and the period enforcer technique that we use in our system. It also assumes bounded latency network reservation support. While the schedulability analysis itself is admittedly not novel, we believe that its kernel-level support within a distributed system framework is new, useful and a necessary first step.

In summary, distributed RK appears to be the first system to provide end-to-end temporal resource isolation through system-wide reservation and enforcement. Distributed RK brings together multiple (known) concepts into a coherent and usable framework for realizing practical large-scale distributed real-time systems. The distributed RK abstraction of a distributed resource container can also be extended in the future to provide a unified interface for managing the timing, security and fault-tolerance properties of distributed real-time applications. For example, on the security front, a distributed resource container is a logical entity within which different encryption, authentication and authorization policies may be specified (and enforced) independent of that of other containers. Similarly, redundant executions on different processors can be included and synchronized within a distributed container for built-in extensions that support Fault-Tolerant or highly available systems. Our early experience with the system to date is positive and encouraging.

1.2. Organization Of The Paper

The rest of this paper is organized as follows. Section 2 provides a brief background on resource kernels, describes the design goals of our framework and introduces the distributed task graph model we employ. Section 3 describes the architecture of Distributed RK, while Section 4 details its implementation. Section 5 presents an evaluation of the performance of the design and implementation. Finally, conclusions and future work are discussed in Section 6.

2. Distributed Resource Kernels

We now provide the context for our distributed RK framework. First, we introduce the resource kernel paradigm that we build upon. Next, we develop the design goals of our framework. Following which, the distributed task graph model used to represent distributed real-time applications in our framework is described.

2.1. Resource Kernels

Resource kernels [15, 16] represent resource-centric approaches for building real-time kernels that provide timely, guaranteed and enforced access to system resources. The applications using the system can specify their resource demands and the kernel assumes the responsibility of satisfying the specified demands using its own resource management policies. The kernel also guarantees that the demands will be met throughout the lifetime of the application by performing admission control tests before admitting any new application into the system, and preventing the applications from overrunning their pre-specified demands. Thus, the key features of the resource kernel architecture are the use of a uniform resource model for dynamic sharing of different resources, and the provision of fine-grained timing guarantees and temporal isolation through admission control and enforcement. Temporal isolation is the property of the system that guarantees that the application timing behaviors are independent of each other. Misbehaving applications therefore, cannot affect the timing properties of other applications sharing the system.

Resource kernels have been shown to provide guaranteed and timely access to various resources including processor cycles [10], disk bandwidth [11], network bandwidth [12] and physical memory [13]. The core first-class resource management entities introduced in resource kernels are:

1. A Reserve represents a share of any available resource in the system. Applications in the system make use of the

\(^1\)It may be noted that [5] also supports network bandwidth reservation. However, end-to-end support is not provided and would have to be built at the user-level through middleware or user-level libraries. Our proposed framework provides comprehensive and efficient support within the kernel space for end-to-end reservations.
available resources through reserves. The resource can be any shareable resource including processor time, memory, network bandwidth, disk bandwidth or physical memory.

2. A Resource Set represents an aggregation of reserves on various resources. This aggregation, to which applications can attach themselves, serves as the basic resource management unit. Resource allocation policies are enacted at the granularity of resource sets.

2.2. Design Goals For Distributed RK

In this section, we describe the major goals of our framework. We also summarize how the framework strives to accomplish each goal. Goals listed first are fundamental to the framework, while the remaining goals are necessary for practical acceptance, and for ease of programming, diagnostics, adoption and usage.

2.2.1 End-to-End Deadline Guarantees

The Distributed RK framework must enable distributed real-time applications to meet their end-to-end deadline requirements. This goal is achieved by performing appropriate real-time scheduling at each node. The distributed RK approach to meeting end-to-end deadlines is described in more detail in Section 2.3.

2.2.2 Temporal Resource Isolation

Timing guarantees provided by the framework to each independent application should hold irrespective of the behavior of other applications, given that other applications may be spawned dynamically or even misbehave at run-time. Hence, the framework supports admission control and distributed enforcement mechanisms to ensure that the timing behavior of a task is not compromised by the behavior of the other tasks. High-resolution timers are employed to provide fine-grained temporal isolation. Dynamic modification of resource reservation parameters is also feasible.

2.2.3 Powerful End-to-End Application Abstractions

Each application in a distributed real-time system may use a number of nodes, with different resource demands on multiple resource types, including processor cycles, network bandwidth, disk bandwidth and physical memory. An abstraction that collects these requirements into a single entity and presents the view of a virtual isolated operating environment can be very desirable for control, reference, monitoring and adaptation purposes. This programming abstraction will also greatly facilitate application development, while the temporal isolation support will significantly ease testing requirements and serve to contain timing faults. Our framework proposes an abstraction called the distributed resource container, which collects the distributed resource reservations on multiple nodes and presents a single reference to all the entities and resources corresponding to the application. This abstraction decouples the application requirements from the resource management policies used by the system. In other words, resource management policies could change independent of the application requirements. In fact, extensions to our framework along the dimensions of security, fault-tolerance and other attributes can also leverage this abstraction in the future.

2.2.4 Efficient System Utilization

An important goal of the distributed RK framework is to utilize the system resources efficiently and effectively. In other words, any admission control tests and enforcement mechanisms used should obtain provably good resource utilizations at acceptably low overheads. The distributed RK framework leverages well-known real-time scheduling policies like rate-monotonic and deadline-monotonic [14] to meet this goal. However, higher-level goals such as load-balancing are left for system-wide QoS and load managers [32] which can be built on top of the framework.

2.2.5 Remote Resource Management

For scalability and ease of administration, the framework must allow all resource management operations to be carried out remotely. The resource management API should be uniform and transparent to the locations of the resources being managed. This enables the administrator to view the overall system as a collection of resources under his control. The amount of unallocated resources must be easily viewable. Our framework has utilities for remote creation, modification and deletion of application threads and associated distributed resource containers.

2.2.6 Lightweight Runtime Monitoring

The run-time framework should enable the system administrator to monitor the resource utilization levels of any deployed application. These resource usage statistics are useful for diagnostic and performance monitoring tools. Utilization levels can also be used to identify potential bugs in the applications. Higher-level resource managers can also adapt the size of containers to adjust for workloads and changing QoS requirements. In order to be efficient, the framework has to support lightweight monitoring of the entire system. In our framework, the individual nodes are responsible for monitoring their resources (performed within the OS kernel) and aggregation is performed at the administrator node on an as-needed basis.

2.2.7 Ease Of Application Deployment

Deploying distributed applications can be a cumbersome process in large-scale distributed real-time systems. It would therefore be convenient to have an application deployment utility that allows easy mapping of subtasks to resources, remote creation of needed reservations and automatic aggregation into distributed resource containers.

We now describe the distributed task graph model used to represent distributed real-time applications for our framework.

2.3. Distributed Task Graphs

We represent a distributed real-time application using a distributed task graph, where each application is divided into a set of communicating sub-tasks. The nodes of the graph comprise the sub-tasks, and directed edges represent the communication between these sub-tasks. A sub-task (node) may receive inputs from zero or more sub-tasks, performs local computation and generates one or more outputs for consumption by other tasks. Each sub-task can be allocated to a different processor, but a sub-task cannot span more than one processor. An example...
distributed task graph is shown in Figure 1. We assume that every distributed task graph is acyclic\(^2\).

Given a distributed task graph, the next step is to map each of its subtasks to a system resource. Due to length restrictions, a detailed discussion of the allocation problem is beyond the scope of this paper. A wealth of literature exists on the task allocation problem in distributed real-time systems [30, 31] on task graphs similar to the one we use. For example, end-to-end deadlines can be partitioned into local deadlines equally or in proportion to substage utilizations. Partitioning deadlines based on total processor loads or not having local deadlines at all would be somewhat counter to the spirit of resource kernels. The focus of our framework is to provide end-to-end timing guarantees and temporal isolation, given the distributed task graph as well as its corresponding task allocation.

2.3.1 Notation

Assume that there are \(n\) end-to-end applications in the system, represented as \(\tau_i\), \(i = 1, ..., n\). Each application \(\tau_i\) is divided into \(m_i\) sub-tasks. Each subtask \(\tau_{i,j}\) is represented by a node in its distributed task graph, where

- \(\tau_{i,j} = (C_{i,j}, S_{i,j}, T_{i,j}, D_{i,j}), i = 1, ..., n; j = 1, ..., m_i\)
- \(C_{i,j}\) = worst-case computation time of the sub-task.
- \(S_{i,j}\) = time offset when the sub-task is released.
- \(T_{i,j}\) = recurrence interval or period.
- \(D_{i,j}\) = deadline requirement of the sub-task.

Communication flows between subtasks are modeled as directed edges. If subtask \(\tau_{i,j}\) provides input to subtask \(\tau_{k,l}\), this is represented by edge \(L_{i,j,k}\). The weight of edge \(L_{i,j,k}\) is the communication latency along the network path due to media contention and propagation delays. The value of \(L_{i,j,k}\) depends on whether \(j\) and \(k\) are co-located or on different nodes. The assumption here is that the protocol processing time is added appropriately to the computation times \(C_{i,j}\) and \(C_{i,k}\). Obtaining an upper bound on the communication latency is easier in some topologies (such as point-to-point links and priority-arbitrated buses) and harder in others. Distributed RK assumes asynchronous communications between sub-tasks, where the sender does not wait for the receiver to acknowledge each packet. For example, protocols like UDP (User Datagram Protocol) may be used for communications and suitable buffer sizes must be used to accommodate all packets that can be generated during every period \(T_{i,j}\) along a link. We represent the upper bound on the maximum clock skew present in the system by \(\Pi > 0\).

We now list some assumptions made. We assume that global locks are not held perhaps requiring that rendezvous-type mechanisms using servers may be needed. Intra-processor protocols to minimize priority inversion can however be used [36]. Pipelined systems as used in many mission control platforms and signal processing applications are natural candidates for this computational model. While reservations use a periodic model, an application using a container may or may not be periodic; the reservations can then be viewed as a “traffic shaper”. For now, nodes in the system are assumed not to crash. In the future, we plan to extend distributed resource containers to support security and fault-tolerance attributes. Fault-Tolerant extensions will support failure detection and application migration. Similarly, security extensions will add support for container-specific authentication, authorization, and encryption.

2.3.2 End-To-End Schedulability

Any data block \(k\) originating at a source node of task \(\tau_i\) flows along the acyclic task graph before it reaches one or more of the sink nodes. The end-to-end deadlines are requirements that require bounding the time it takes to reach the individual sink nodes from the source node. If the source node is \(p\) and the sink node is \(q\), the basic feasibility test for the end-to-end deadline \((D_{i,p,q})\) is: for each path \(R\) from the source \(p\) to sink \(q\)

\[
D_{i,p,q} \geq \sum_{j \in R(v)} D_{i,j} + \sum_{(j,k) \in R(v)} L_{i,j,k} + \Pi
\]

where \(R(v)\) is the set of vertices in \(R\) and \(R(v)\) is the set of edges \((u,v)\) in \(R\).

If this basic feasibility test is satisfied, then the admission control test is performed on the individual nodes on which the subtasks are allocated. These single-node admission control tests are the standard admission control tests in [14], and are chosen based on the real-time scheduling algorithm configured for the system (rate-monotonic or deadline-monotonic). If the admission control test fails in one of the nodes, reservations already made for the application \(\tau_i\) are reclaimed and the application is declared unschedulable with the current mapping of subtasks to resources. Phasing control of reservations can play a key role in end-to-end worst-case response times. Without phasing control, worst-case response times become pessimistic,
but can be tightened with such control. Clock skew needs to be considered under such scenarios.

Remark: One subtle yet important feature of the above model is that different instances of a subtask $\tau_{i,j}$ can complete at different times (even if all are within the local relative deadline of $D_{i,j}$) due to the different preemption sequences they encounter and due to their own variations in execution times. If the completion of $\tau_{i,j}$ instantaneously triggers instances of subtask $\tau_{i,j+1}$, this can introduce serious schedulability penalties on the node hosting $\tau_{i,j+1}$, due to deferrable server-like back-to-back preemptions [17]. These penalties can cause deadlines to become infeasible even at relatively low-levels of resource utilization. The start-times $S_{i,j}$ must therefore be adjusted such that the task is released only at appropriate start-times irrespective of the completion times of earlier subtasks. This scheme represents a simple implementation of a period enforcer [18]. This scheme requires that the nodes in the system be relatively synchronized with respect to each other and the synchronization error between the nodes therefore needs to be accommodated in the basic feasibility test.

We now describe the architecture of Distributed RK and how it accomplishes our goals.

3. Distributed RK Architecture

The architecture of the Distributed RK layering on each (local) node of the distributed system is shown in Figure 3. The major components of this layering are described next.

3.1. Dist. Resource Manager Daemons

Every node in the system has a resource manager daemon that listens for resource reservation requests from its counterparts. Whenever such a request is received, it is evaluated, and if the admission control succeeds, the resource is reserved. A handle to the reference is piggybacked along with the reply to the requesting resource manager. In order to present a uniform interface for resource reservation, the framework identifies each resource by the resource address and automatically dispatches the reservation request to the corresponding resource management daemon. As opted by servers like kHTTP [27], our daemons reside in the kernel, minimize kernel-to-userspace copies, and enhance performance for operations including logging and aggregation.

The resource manager daemon is a core component of the distributed RK framework. It is responsible for:

- Managing the resources present in the local node,
- Coordinating with the resource managers on other nodes to remotely reserve resources, and
- Aggregating the remotely managed resource reservations into a distributed resource container and registering it with the name service (discussed next).

It is desirable for scalability and (future) fault-tolerance purposes that there be no central command and control center. Therefore needs to be accommodated in the basic feasibility test.

There can also be multiple administrative domains within the same system, sharing the resources, and therefore having a single administrator node is not practical. We therefore built the distributed RK framework to support multiple co-operating administrator nodes. This led to the development of distributed resource manager daemons, which can seamlessly enable systems with different administrative configurations.

3.2. Name Service

Location-transparency and portability are core requirements for the distributed RK framework. Subtasks in the distributed system need not be aware of their host resource or any location information. Such transparency enables our framework to be extended easily later for fault-tolerance purposes through replication and migration. In order to support such transparent operations, it is required that each application creates a distributed resource container with a unique id and the sub-tasks of the application attach themselves to the container with this id. Note that such an id will be decided at build-time of the individual applications. When the sub-tasks are deployed on different nodes, they can use this globally unique id to query a name server provided as a part of the distributed RK framework, and inherit the resources allocated to the application. The id can also be used to query the location of other sub-tasks of the application and set up the communication paths during application initialization.

The naming service is jointly provided by a group of name servers running in the system. The name servers have a globally partitioned set of ids for which they are responsible. Name servers advertise their presence periodically and the resource managers can query them, whenever they wish to resolve a global id. When applications are to be cleaned up, the name servers are notified to update their entries. The global ids used in our framework are currently alphanumeric strings of fixed length.

3.3. Message-Passing Subsystem

The distributed resource-manager daemons need to coordinate with each other frequently during the system operation. The message-passing subsystem acts as an efficient kernel-level communication framework to enable such co-ordination. It is lightweight, and currently uses datagrams for point-to-point communication. Each message has the following format:

1. **Message Type:** Specifies the command requested by the message (e.g. remote resource set creation).
2. **Sequence Number:** Used to discard duplicates and manage buffers. A timeout mechanism is built into the sender and receiver state machines.
3. **Length:** Identifies the payload size of the message.
4. **Payload:** Specifies the parameters needed to carry out the command specified by the message type or the return values from the result of the command. The format of the payload varies with message type.

The message-passing subsystem in our framework is similar to other message-passing systems like MPI [21]. The major
difference in our architecture is that our communication framework is implemented at the kernel-level rather than in userspace. Our primary intent is to reap the performance benefits of the kernel-level implementation obtained by various other servers like kHTTPD [27].

3.4. The dMon Deployment Utility

The dMon deployment manager is a utility that can automatically deploy one or more end-to-end applications on the nodes of the distributed system. A system configuration file (an example is shown in Figure 5) describing the allocation of tasks is used by the deployment manager. The utility loads each application described in the configuration, automatically creates a distributed resource container, creates the remote resource sets on the required nodes, aggregates them into the distributed resource container and automatically spawns the sub-tasks of the application on the different nodes. When an application exits, the deployment manager is notified and it automatically cleans up the resource sets and associated distributed resource containers. The deployment utility thus enables the resource management layer to be operated independent of the applications in the system. The dMon utility has a graphical interface, which presents a global view of the system, with facilities to add (or delete) nodes, resource containers, resource applications and to monitor these entities.

3.5. Status Aggregation Engine

The status aggregation engine is used to monitor resource utilization statistics. The utilization information is exported by the individual nodes onto their respective \proc interfaces. The message-passing infrastructure is then used to export this information on an as-needed basis to the administrator nodes. Various performance monitors and diagnostic tools can benefit from such global system-wide status information. Unusual patterns in resource utilization levels are commonly used to detect timing bugs in applications deployed in the system.

3.6. Global Time Service

Distributed RK provides a global time service through a separate time daemon. A client for this service is spawned in each of the nodes and they listen for messages from the time servers. This service is important to synchronize all the nodes in the distributed system and hence incur less penalty in scheduling, during period enforcements. Global timing information can also be used by distributed applications to compute global event-ordering. Our time service resembles the commonly used NTP [23] service for synchronization over the Internet, except for some simplifications and optimizations targeting relatively smaller network topologies.

4. Distributed RK Implementation

We have implemented the distributed resource kernel framework described in the previous section. Our current implementation runs on X86 architectures using the Linux/RK platform. Linux/RK is available as a kernel module for a minimally modified version of the 2.6.18 kernel and is available at http://www.cs.cmu.edu/~rtml

Our current implementation of distributed resource kernels supports only remote reservation of processor cycles (CPU reserves). In the future, we plan to also provide support for network and disk bandwidth reservations.

Our message-passing subsystem consists of the following message types: (1) Advertisement messages - sent by the name server (2) Request messages - sent for creating, deleting and modifying resource reserves, resource sets and distributed resource containers (3) Reply messages - Success or Error acknowledgement messages corresponding to the return status of the requests. The message-passing framework supports the following wrappers over the kernel implementation of the UDP protocol: rk_create_socket(), rk_bind_socket(), rk_release_socket(), rk_sendto(). This API is used to communicate between the various distributed resource manager daemons.

The distributed resource manager daemon is implemented as a kernel thread rk_local_rm and encapsulated within the Linux/RK module as a single module. The name server and global time service are available as separate modules. The name server advertises its presence using the advertisement messages and maintains the naming registry as a table hashed by distributed resource container ids. The time service is a collection of local global time service daemons, one for each node in the system. These time daemons are used to correct the local clocks, based on the synchronization service provided by time servers in the system. rk_get_current_time() can be used by applications to obtain the global time as maintained by the service.

4.1. Distributed RK System Calls

Table 1 lists the current set of system calls supported by the distributed RK framework. This table also includes a brief description of each supported system call. Return values and error conditions are not included due to space limitations.

4.2. Application Deployment

We now illustrate the typical sequence of steps involved in developing and deploying an application using the distributed RK framework.

(1) First, a resource container is created using the rk_distributed_resource_container_create() system call. This system call hands control to the distributed resource manager daemon, which creates an entry in the local registry (maintained as a hash table) and sends a registration request to the name server. The calling task is suspended until a response is received from the name server. Once the name server responds, the local registry is updated and the calling task is awakened by the resource manager.

(2) The next step is to create resource sets in the remote nodes to scope and use the resource on those nodes. The rk_distributed_resource_set_create() system call is used to accomplish this objective. The distributed resource manager uses the distributed resource container handle and talks to the remote resource managers to create the resource sets. The resource set handle stored in the local registry and a copy of the handle is returned to the calling task.

(3) The reserves with the exact parameters are created using the rk_distributed_reserve_create() System call. The resource set handle is used to obtain information about the location of the remote resource set and the distributed resource manager daemon then co-ordinates with the corresponding manager to create the reserve. The reserve is then linked to the remote resource set.
distribution

distributed

Table 1. Distributed Resource Kernel API

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed resource container creation</td>
<td>635</td>
</tr>
<tr>
<td>distributed resource set creation</td>
<td>555</td>
</tr>
<tr>
<td>local resource set creation</td>
<td>013</td>
</tr>
<tr>
<td>distributed resource creation</td>
<td>480</td>
</tr>
<tr>
<td>local reserve creation</td>
<td>20</td>
</tr>
<tr>
<td>rk_setCurrentTime()</td>
<td>0.000280</td>
</tr>
</tbody>
</table>

Table 2. System Call Execution Times

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>distributed resource container creation</td>
<td>635</td>
</tr>
<tr>
<td>distributed resource set creation</td>
<td>555</td>
</tr>
<tr>
<td>local resource set creation</td>
<td>013</td>
</tr>
<tr>
<td>distributed resource creation</td>
<td>480</td>
</tr>
<tr>
<td>local reserve creation</td>
<td>20</td>
</tr>
<tr>
<td>rk_setCurrentTime()</td>
<td>0.000280</td>
</tr>
</tbody>
</table>

(4) The final step is to attach the sub-task process to the resource set created in the above steps. This is accomplished by the `rk_distributed_resource_set_attach_process()` system call. In order to perform this operation, the process id of the task being attached remotely should be known. This information is propagated from the remote resource manager beforehand.

(5) When an application exits, the corresponding cleanup system calls are executed to recover the allocated resources. The system administrator can use these system calls to create a custom application deployment infrastructure for the distributed real-time system. The individual applications need not be aware of their location information or host source, whereas the deployment subsystems heavily rely on this information to deploy the distributed applications. We present some sample code in Figure 4 as an example of the above initialization sequence and the real-time processing loop of an application. In summary, the system calls of the distributed RK framework are designed to facilitate easy deployment, maintenance, and management of distributed real-time applications.

5. Distributed RK Evaluation

In this section, we conduct a series of experiments to evaluate the performance of distributed RK and its ability to provide timing guarantees under different conditions.

5.1. Execution Times of Primitives

In order to measure the run-time overhead introduced by distributed RK, the time taken to execute the different system calls provided by distributed RK was measured. The results obtained using the 2-node hardware configuration shown in Table 3 are summarized in Table 2. The measurements are for the very

first reservations and container created, and are only meant to be qualitative. As one might expect, the distributed resource management system calls take a considerable amount of time (hundreds of milliseconds) compared to their local counterparts. Major contributors for these long latencies include (i) network delays (ii) name resolution time at the name server and (iii) lack of optimizations in the current implementation (like name caches). A major performance difference can also be seen between the local resource creation and local resource set creation operations. The major reason being the relatively expensive admission control test performed for the former operation, while the latter requires simple aggregation and accounting. It is very important to note that the design of Distributed RK is such that most of the distributed system calls are invoked only during the initialization and cleanup phases.

Figure 4. Example Code

```c
/* Do the application processing here */

while (application_process()) {
  /&...*/
  if (application_process()) {
    /&...*/
  }
}
```
of applications. During the steady-state execution of an application, the global time service is the only service that may be regularly used. The \( \text{current_time}() \) system call, however, is relatively lightweight since it only queries the local copy of global time. Its functionality could be subsumed into the Linux \( \text{get_time}() \) call. The time service is running in the background and the kernel-level communication framework reduces the processing overhead. These measurements suggest that care must be taken to ensure that the non-real time system calls (such as creating, deleting and modifying resource, resource sets and distributed resource containers) must be executed outside the real-time loop.

5.2. Distributed Hartstone Benchmarks

In our next set of experiments, we ran key Distributed Hartstone (DH) Benchmarks [25] to quantify the real-time scheduling capabilities of the Distributed RK framework. For our DH experiments, we used two nodes whose configurations are shown in Table 3. The DH benchmarks were developed specifically to evaluate the performance of distributed real-time systems. This is accomplished by stressing different parts of the OS subsystem that are critical for real-time tasks. The breaking point of the system, at which deadlines are beginning to be missed, is noted as the performance metric value for the system under evaluation. The idea behind these tests is that the entire system is only as strong as the weakest sub-system. The core workloads of the DH benchmarks originally specified in 1990 are seriously outdated and we scaled them up for use in modern architectures that have significantly higher performance. The original DH benchmark workloads were developed to evaluate the ARTS kernel on a SUN3 which took 4.2ms to execute a synthetic workload of 1 kilo-Whetstone. We had to scale this workload until the first deadline was missed. Under the distributed RK framework, deadlines were missed when the server execution time is 38ms, there are cases when \( \tau_2 \), \( \tau_3 \), or \( \tau_4 \) begins executing at exactly the same instant when \( \tau_5 \) is released. When \( \tau_5 \) sends a request to the server, it may have to wait for 38ms since the server is non-preemptible and then it takes another 38ms for its request to be serviced, leading to a total of 76ms, which when followed by the 4.2ms computation delay causes it to miss the deadline of 80ms.

This DHcl-37 result can be compared to the performance metric of DHcl-35 reported for the ARTS kernel [25]. From this experiment, we conclude that distributed RK performs efficient and appropriate real-time scheduling. The main features of distributed RK, are providing support for end-to-end resource reservation and temporal isolation, in addition to end-to-end real-time scheduling.

5.2.2 DSHpq Series: Priority Queuing

The next set of experiments tests for the priority queuing of communication packets, attempting to show the advantage of priority queuing over conventional FIFOs used in communication. The task-set for this experiment is given in Table 4. All these stages have to be completed before the task deadline. This task-set is used to test the system’s communication latency. Prioritization of requests is done at the application level.

Our hardware configuration comprised of nodes connected by a 100 Mbps Ethernet link with a round-trip network latency of about 0.2 ms. The workload of the server was increased until it caused one of the clients to miss a deadline. We found that the deadlines were missed by task \( \tau_1 \) when the computation time of the server was increased to 37 ms. Thus the performance of the distributed RK platform on this benchmark is DSHcl-37. The reason that the task \( \tau_1 \) misses the deadline is that when the server execution time is 38 ms, there are cases when \( \tau_2 \), \( \tau_3 \), \( \tau_4 \) or \( \tau_5 \) begins executing at exactly the same instant when \( \tau_5 \) is released. When \( \tau_5 \) sends a request to the server, it may have to wait for 38ms since the server is non-preemptible and then it takes another 38ms for its request to be serviced, leading to a total of 76ms, which when followed by the 4.2ms computation delay causes it to miss the deadline of 80ms.

This DSHcl-37 result can be compared to the performance metric of DSHcl-35 reported for the ARTS kernel [25]. From this experiment, we conclude that distributed RK performs efficient and appropriate real-time scheduling. The main features of distributed RK, are providing support for end-to-end resource reservation and temporal isolation, in addition to end-to-end real-time scheduling.

5.3. Temporal Isolation

Our next goal was to evaluate the temporal isolation support provided by the distributed RK framework. We designed two applications such that the first application has a reservation of 20ms of CPU time every 40ms each on \( \text{Node}_1 \) and \( \text{Node}_2 \). The second application has a reservation of 20ms every 120ms.
Figure 6. Hard and Soft Enforcement on Node 1 and 40ms every 120ms on Node 2. To maximize the potential interference between tasks, we attached infinite loop processes to each of these reservations. These applications were deployed using our automatic deployment framework and the results were monitored using our status aggregation engine. If there were no temporal isolation, one of these tasks will never get the chance to execute in a purely fixed-priority-based preemptive scheduling environment.

Figure 6 presents the CPU consumption on Node 1 (left column) and Node 2 (right column) by these two tasks. Each point in these graphs represents the fraction of cycles obtained by the application sub-task during its most recent reservation period. We studied the performance of the system under both hard reserves and soft reserves [15]. Hard reserves represent the policy where the application execution will be suspended when its CPU reserve is used up, whereas in soft reserves the application will also be allowed to contend for and use unreserved and unused CPU cycles.

It can be seen from the results that Application 1 obtains 50 percent on Node 1 (as P11) and 50 percent on Node 2 (as P12), corresponding to its reserves of 20ms every 40ms. Application 2 obtains roughly 17 percent on Node 1 (as P21) and 33 percent on Node 2 (as P22) using hard reserves. When Application 2 was attached to soft reserves, it can be seen that it obtains extra CPU cycles whenever no other application is utilizing the processor.

The tested applications, were infinite loops and this behavior would not be seen in other purely priority-based real-time operating systems, since the tasks would never suspend themselves. The observed behavior under Distributed RK illustrates its distributed enforcement capabilities and hence its ability to provide temporal isolation.

5.4. Audio Processing Pipeline

We finally conducted a set of experiments to further validate the temporal isolation properties and deadline guarantees, provided by the distributed RK framework for real-life applications. Towards this goal, we created an audio processing application that is pipelined across three nodes managed by distributed RK. The pipeline, shown in Figure 7, consists of an audio server which reads in two audio streams from a source, forwards them to a gateway, which in turn delivers the content to a client node. The audio server was exclusively dedicated to the streaming service, while the gateway and client had other applications sharing the processing cycles. The audio signal was sampled at 8 KHz, with 8 bits per PCM sample. A few hundred samples were batched together in each packet transmission. The end-to-end deadline for each stream was set at 300ms and the individual subtask deadline was set as 80ms. The remaining 60ms is to primarily accommodate any network delays. Buffers along the pipeline were modified to reflect these timing constraints such that the buffers would be overwritten whenever newer data arrives and the older data have not been dequeued. The behavior of this setup was compared with 3 nodes running the standard Linux 2.6.18 kernel. The results of our comparison are shown in Figure 8. The left column corresponds to Stream 1 and the right column corresponds to Stream 2. The 1st row corresponds to the audio stream as seen at the server (first pipeline stage). The 2nd row shows the streams seen at the gateways under Linux (no RK) and the 3rd row shows the audio stream received by the clients running Linux (no RK). It can be clearly seen that when standard Linux is used, each audio stream exhibits discrete pauses, when the other stream or application gets scheduled. The Linux kernel does not inherently understand the timing properties of the audio streams and therefore cannot guarantee their end-to-end timing constraints. The results obtained with the distributed RK framework are shown in the 4th and 5th rows. The local reserve parameters were set as (30ms,80ms,80ms) for (C,T,D). Packets were received and processed without any pauses or bursty packet drops, despite the presence of other reserved and unreserved applications.

The end-to-end delays were measured as follows. Each packet was time-stamped at the source (using the globally synchronized clock) and the subtasks dropped packets that exceeded their end-to-end deadlines. We found that a sizable fraction of packets missed their deadlines with standard Linux, but not with the distributed RK reservations in place.

The above experiments demonstrate the capability of the distributed RK framework to provide end-to-end deadline guarantees and temporal isolation.

6. Concluding Remarks and Future Work

In this paper, we proposed and developed the distributed RK framework to realize robust distributed real-time systems. It accomplishes the important requirements of guaranteeing end-to-end deadlines that span multiple processor boundaries, temporally isolating each end-to-end application from the other and
offering facilities to ease development, deployment and diagnostics. The major highlights of our framework include end-to-end enforcement, strong predictability, analyzability and excellent performance.

The architecture supports an abstraction of a distributed resource container that represents a virtual operating environment for each application. Name services and time synchronization services are also provided as a part of our framework. The framework has been implemented on top of Linux 2.6.18. Detailed evaluation using benchmarks, artificial but stressful workloads, and realistic application scenarios show that our goals are satisfied. Implementation overheads were also shown to be acceptable.

In the future, we plan to extend distributed resource kernels to incorporate fault-tolerance semantics and security attributes. A primary motivation behind the development of distributed RK is to encapsulate the end-to-end properties of distributed real-time applications and guarantee those properties in a user-transparent way. Its distributed resource container abstraction can be further extended to incorporate security properties of the distributed application like access-capabilities and confidentiality requirements. For example, the distributed resource container could specify the level of confidentiality required for communication across processors within a container and the underlying system would transparently guarantee this property through appropriate encryption techniques on messages passed between the subtasks of the application. Fault-tolerance properties can also be added to the distributed resource container abstraction in a similar fashion and properties like redundancy-requirements can be automatically handled by the OS framework. The distributed container abstraction is therefore a sound foundation for enabling such a unified interface where applications specify their properties, and in turn the system guarantees that they are indeed satisfied.

References