

Sharc: Managing CPU and Network Bandwidth in Shared Clusters

Abstract

In this paper, we argue the need for effective resource management mechanisms for sharing resources in commodity clusters. To address this issue, we present the design of Sharc—a system that enables resource sharing among applications in such clusters. Sharc depends on single node resource management mechanisms such as reservations or shares and extends the benefits of such mechanisms to clustered environments. We present techniques for managing two important resources—CPU and network interface bandwidth—on a cluster-wide basis. Our techniques allow Sharc to (i) support reservation of CPU and network interface bandwidth for distributed applications, (ii) dynamically allocate resources based on past usage, and (iii) provide performance isolation to applications. Our experimental evaluation has shown that Sharc can scale to 256 node clusters running 100,000 applications. These results demonstrate that Sharc can be an effective approach for sharing resources among competing applications in moderate size clusters.

1 Introduction

1.1 Motivation

Due to the rapid advances in computing and networking technologies and falling hardware prices, server clusters built using commodity hardware have become an attractive alternative to the traditional large multiprocessor servers. Commodity clusters are being increasingly used in a variety of environments such as third-party hosting platforms and as workgroup servers. For instance, hosting platforms are using commodity clusters to provide computational resources to third-party applications—application owners pay for resources on the platform and the platform provider in turn guarantees resource availability to applications [23]. Workgroups (e.g., a research group in a university department) are using commodity clusters as compute servers to run scientific applications, large-scale simulations, and batch jobs such as application builds.

In this paper, we focus on the design, implementation, and evaluation of resource management mechanisms in a *shared cluster*—a commodity cluster where the number of applications is significantly larger than the number of nodes, necessitating resource sharing among applications. Shared clusters are different from *dedicated* clusters, where a single application runs on a cluster of nodes (e.g., clustered mail servers [24], replicated web servers with a load balancing switch [4]), or each application runs on a dedicated node in the cluster (e.g., dedicated hosting platforms such as those used by application service providers). Due to economic reasons of space, power, cooling and cost, shared clusters are more attractive for many application environments than dedicated clusters. Whereas dedicated clusters are widely used for many niche applications that warrant their additional cost, the widespread deployment and use of shared clusters has been hampered by the lack of effective mechanisms to share cluster resources among applications. Consequently, today’s clusters avoid resource sharing altogether (i.e., employ a dedicated cluster model), or share resources either in a best-effort manner or based on informal agreements among users (e.g., researchers wanting to run simulation experiments agree to do so at mutually exclusive times or on mutually exclusive nodes).

There are a number of research issues that must be addressed to enable effective resource sharing in commodity clusters. Since lots of applications share a relatively small number of machines, resource management is a central issue in shared clusters. The ability to reserve resources for individual applications (especially when application owners may be paying for these resources), the ability to isolate applications from one another, and the need to manage the heterogeneous performance requirements of applications are some challenges that must be addressed in shared environments. High availability and scalability are other important issues, although they are common to dedicated clusters as well. This paper focuses on the design of resource management mechanisms for shared clusters that meet these requirements.

1.2 Research Contributions of this Paper

In this paper, we present *Sharc*: a system for managing resources in shared clusters.¹ Sharc extends the benefits of single node resource management mechanisms to clustered environments.

The primary advantage of Sharc is its simplicity. Sharc typically requires no changes to the operating system—so long as the operating system supports resource management mechanisms such as reservations or shares, Sharc can be built on top of commodity hardware and commodity operating systems. Sharc is not a cluster middleware; rather it operates in conjunction with the operating system to facilitate resource allocation on a cluster-wide basis. Applications continue to interact with the operating system and with one another using standard OS interfaces and libraries, while benefiting from the resource allocation features provided by Sharc. Sharc supports resource reservation both within a node and across nodes; the latter functionality enables aggregate reservations for distributed applications that span multiple nodes of the cluster (e.g., replicated web servers). The resource management mechanisms employed by Sharc provide performance isolation to applications, and when desirable, allow distributed applications to dynamically share resources among resource principals based on their instantaneous needs. Finally, Sharc provides high availability of cluster resources by detecting and recovering from many types of failures.

In this paper, we discuss the design requirements for resource management mechanisms in shared clusters and present techniques for managing two important cluster resources, namely CPU and network interface bandwidth. We discuss the implementation of our techniques on a cluster of Linux PCs and demonstrate its efficacy using an experimental evaluation. Our results show that Sharc can (i) provide predictable allocation of CPU and network interface bandwidth, (ii) isolate applications from one another, and (iii) handle a variety of failure scenarios. A key advantage of our approach is its efficiency—unlike previous approaches [3] that have polynomial time complexity, our techniques have complexity that is linear in the number of applications in the cluster. Our experiments show that this efficiency allows Sharc to easily scale to moderate size-clusters with 256 nodes running 100,000 applications.

The rest of this paper is structured as follows. Section 2 lists the design requirements for resource management mechanisms in shared clusters. Section 3 presents an overview of the Sharc architecture, while Section 4 discusses the mechanisms and policies employed by Sharc. Section 5 describes our prototype implementation, while Section 6 presents our experimental results. We present directions for future work and related work in Sections 7 and 8, respectively. Finally Section 9 presents our conclusions.

2 Resource Management in Shared Clusters: Requirements

Consider a shared cluster built using commodity hardware and software. Applications running on such a cluster could be centralized or distributed and could span multiple nodes in the cluster. We refer to that component of an application that runs on an individual node as a *capsule*. Each application has at least one capsule and more if the application is distributed. The component of the cluster that manages resources (and capsules) on each individual node is referred to as the *nucleus*. The component of the cluster that coordinates various nuclei and manages resources on a cluster-wide basis is referred to as the *control plane*. Together, the control plane and the nuclei enable the cluster to share resources among multiple applications. In such a scenario, the control plane and the nuclei should address the following requirements.

Application Heterogeneity. Applications running on a shared cluster will have diverse performance requirements. To illustrate, a third-party hosting platform can be expected to run a mix of applications such as game servers (e.g., Quake), vanilla web servers, streaming media servers, ecommerce, and peer-to-peer applications. Similarly, shared clusters in workgroup environments will run a mix of scientific applications, simulations, and batch jobs. Observe that these applications have heterogeneous performance requirements. For instance, game servers need good interactive performance and thus low average response times, ecommerce applications need high aggregate throughput (in terms of transactions per second), and streaming

¹As an acronym, SHARC stands for Scalable Hierarchical Allocation of Resources in Clusters. As an abbreviation, Sharc is short for a *shared cluster*. We prefer the latter connotation.

media servers require real-time performance guarantees. In addition to heterogeneity across applications, there could be heterogeneity *within* each application. For instance, an ecommerce application might consist of capsules to service HTTP requests, to handle electronic payments, and to manage product catalogs. Each such capsule imposes a different performance requirement. Consequently, the resource management mechanisms in a shared cluster will need to handle the diverse performance requirements of capsules within and across applications.

Resource Reservation. Since the number of applications exceeds the number of nodes in a shared cluster, applications in this environment compete for resources. In such a scenario, soft real-time applications such as streaming media servers need to be guaranteed a certain level of service in order to meet timeliness requirements of streaming media. Resource guarantees may be necessary even for non-real-time applications, especially in environments where applications owners are paying for resources. Consequently, a shared cluster should provide the ability to reserve resources for each application and enforce these allocations on a sufficiently fine time-scale.

Resources could be reserved either based on the aggregate needs of the application or based on the needs of individual capsules. In the former case, applications specify their aggregate resource needs but do not specify how these resources are to be partitioned among individual capsules. An example of such an application is a replicated web server that runs on multiple cluster nodes—the aggregate throughput achieved by such an application is of greater concern than the throughput of any individual replica. At the other end of the spectrum are applications that need fine-grain control over the allocation to each individual capsule. An ecommerce application exemplifies this scenario, since each individual capsule (e.g., catalog database, payment handler) performs a different task and has different resource requirements. For such applications, the cluster should provide the flexibility of resource reservation on a per-capsule basis. Finally, the ability of a capsule to trade resources with other peer capsules is also important. For instance, application capsules that are not utilizing their allocations should be able to temporarily lend resources, such as CPU cycles, to other needy capsules of that application. Since resource trading is not suitable for all applications, the cluster should allow applications to refrain from trading resources when undesirable.

Capsule Placement and Admission Control. A shared cluster that supports resource reservation for applications should ensure that sufficient resources exist on the cluster before admitting each new application. In addition to determining resource availability, the cluster also needs to determine *where* to place each application capsule—due to the large number of application capsules in shared environments, manual mapping of capsules to nodes may be infeasible. Admission control and capsule placement are interrelated tasks—both need to identify cluster nodes with sufficient unused resources to achieve their goals. Consequently, a shared cluster can employ a unified technique that integrates both tasks. Further, due to the potential lack of trust among applications in shared clusters, especially in third-party hosting environments, such a technique will also need to consider trust (or lack thereof) among applications, in addition to resource availability, while admitting applications and determining their placement onto nodes.

Application Isolation. Third party applications running on a shared cluster could be untrusted or mutually antagonistic. Even in workgroup environments where there is more trust between users (and applications), applications could misbehave or get overloaded and affect the performance of other applications. Consequently, a shared cluster should isolate applications from one another and prevent untrusted or misbehaving applications from affecting the performance of other applications. This could be achieved, for instance, by employing resource management mechanisms that also provide performance isolation.

Scalability and Availability. Most commonly used clusters have sizes ranging from a few nodes to a few hundred nodes; each such node runs tens or hundreds of application capsules. Consequently, resource management mechanisms employed by a shared cluster should scale to several hundred nodes running tens of thousands of applications (techniques that scale to very large clusters consisting of thousands or tens of thousands of nodes are beyond the scope of our current work). A typical cluster with several hundred nodes will experience a number of hardware and software failures. Consequently, to ensure high availability, such a cluster should detect common types of failures and recover from them with minimal or no human intervention (for instance, by restarting failed nodes or by offloading capsules from an overloaded node to another node).

Compatibility with Existing OS Interfaces. Whereas the use of a middleware is one approach for managing resources in clustered environments [9, 10], this approach typically requires applications to use the interface exported by the middleware

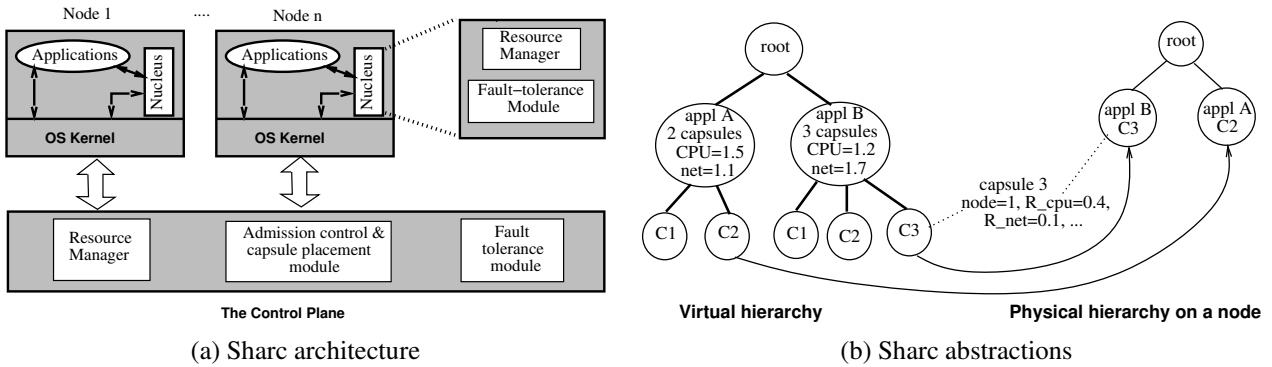


Figure 1: Sharc architecture and abstractions. Figure (a) shows the overall Sharc architecture. Figure (b) shows a sample cluster-wide virtual hierarchy, a physical hierarchy on a node and the relationship between the two.

to realize its benefits. Sharc employs a different design philosophy. We are interested in exploring techniques that allow applications to use standard operating system interfaces and yet benefit from cluster-wide resource allocation mechanisms. Compatibility with existing OS interfaces and libraries is especially important in commercial environments such as hosting platforms where it is infeasible to require third-party applications to use proprietary or non-standard APIs. Such an approach also allows existing and legacy applications to benefit from these resource allocation mechanisms without any modifications. Our goal is to use commodity PCs running commodity operating systems as the building block for designing shared clusters. The only requirement we impose on the underlying operating system is that it support some notion of quality of service such as reservations [18, 19] or shares [11, 14]. Many commercial and open-source operating systems such as Solaris [26], IRIX [25] and FreeBSD [8] already support such features.

Next we present the architecture, mechanisms, and policies employed by Sharc to address these requirements.

3 Sharc Architecture Overview

Sharc consists of two main components—the *control plane* and the *nucleus*—that are responsible for managing resources in the cluster (see Figure 1). Whereas the control plane manages resources on a cluster-wide basis, the nucleus is responsible for doing so on each individual node. Architecturally, the nucleus is distinct from the operating system kernel on a node. Moreover, unlike a middleware, the nucleus does not sit between applications and the kernel; rather it complements the functionality of the operating system kernel (see Figure 1(a)). In general, applications are oblivious of the nucleus and the control plane except at application startup time where they interact with these components to reserve resources. Once resources are reserved, applications interact solely with the OS kernel and with one another, with no further interactions with Sharc. The control plane and the nucleus act transparently on the behalf of applications to determine allocations for individual capsules. To ensure compatibility with different OS platforms, these allocations are determined using OS-independent QoS parameters that are then mapped to OS-specific QoS parameters on each node. The task of *enforcing* these QoS requirements is left to the operating system kernel. This provides a clean separation of functionality between resource reservation and resource scheduling, with Sharc responsible for the former and the OS kernel for the latter.

In this paper, we show how Sharc manages two important cluster resources, namely CPU and network interface bandwidth. Techniques for managing other resources such as memory and disk bandwidth are beyond the scope of this paper.

3.1 The Control Plane

As shown in Figure 1(a), the Sharc control plane consists of a resource manager, an admission control and capsule placement module, and a fault-tolerance module. The admission control and capsule placement module performs two tasks: (i) it ensures

that sufficient resources exist for each new application, and (ii) it determines the placement of capsules onto nodes in the cluster. Capsule placement is necessary not only at application startup time but also to recover from failures or resource exhaustion on a node, since this involves moving affected capsules to other nodes. Once an application is admitted into the system, the resource manager is responsible for ensuring that the aggregate allocation of each application and those of individual capsules are met. For those applications where trading of resources across capsules is permitted, the resource manager periodically determines how to reallocate resources unused by under-utilized capsules to other needy capsules of that application. The fault-tolerance module is responsible for detecting and recovering from node and nucleus failures.

The key abstraction employed by the control plane to achieve these tasks is that of a cluster-wide *virtual hierarchy* (see Figure 1(b)). The virtual hierarchy maintains information about what resources are currently in use in the cluster and by whom. This information is represented hierarchically in the form of a tree. The root of the tree represents all the resources in the cluster. Each child represents an application in the cluster. Information about the number of capsules and the aggregate reservation for that application is maintained in each application node. Each child of an application node represents a capsule. A capsule node maintains information about the location of that capsule (i.e., the node on which the capsule resides), its reservation on that node, its current CPU and network usage and the current allocation (the terms *reservation* and *allocation* are used interchangeably in this paper). Note that the current allocation may be different from the initial reservation if the capsule borrows (or lends) resources from another capsule

3.2 The Nucleus

As shown in Figure 1(a), the nucleus on each node consists of a resource manager and a fault-tolerance module. The resource manager is responsible for reserving resources for capsules as directed by the control plane. The resource manager also translates OS-independent QoS parameters employed by the control plane into node-specific QoS parameters and conveys them to the CPU and network interface schedulers. In addition, the resource manager tracks resource usage for each capsule and periodically reports these statistics to the control plane; this usage information is then used to adjust the instantaneous allocation of capsules if necessary. As indicated earlier, the resource manager does not depend on a particular CPU or network scheduling algorithm to achieve these goals; any scheduler suffices so long as it supports some form of resource reservation (see Section 4.2). The fault tolerance module is responsible for detecting and recovering from control plane failures and is described in Section 4.4.

The nucleus uses the abstraction of a *physical* hierarchy to achieve these goals (see Figure 1(b)). The physical hierarchy maintains information about what resources are in use on a node and by whom. Like the virtual hierarchy, the physical hierarchy is a tree with the root representing all the resources on that node. Each child represents a capsule on that node; information about the initial reservation for the capsule, the current usage, and the current allocation is maintained with each capsule node. As shown in Figure 1(b), there exists a one to one mapping between the virtual hierarchy and the physical hierarchy; this mapping and the resulting replication of state information in the two hierarchies is exploited by Sharc to recover from failures.

4 Sharc Mechanisms and Policies

In this section, we describe resource specification, admission control, and capsule placement policies employed by Sharc. We also describe how Sharc enables capsules to trade resources with one another based on their current usage.

4.1 Resource Specification, Admission Control and Capsule Placement

Each application in Sharc specifies its resource requirement to the control plane at application startup time. The control plane then determines whether sufficient resources exist in the cluster to service the new application and the placement of capsules onto nodes.

Sharc provides applications with an OS-independent mechanism called a *reservation* to specify their resource requirements. Formally, a reservation is a pair (x, y) that requests x units of the resource every y units of time. In case of the CPU, an application that needs x units of CPU time every y time units specifies a reservation of (x, y) . In case of network bandwidth, an application that transmits b bits of data every y time units needs to specify a reservation of $(b/c, y)$, where c is the bandwidth capacity of the network interface. Throughout this paper, we use R to abbreviate the term $\frac{x}{y}$, $0 < R \leq 1$. Intuitively, R is the fraction of the resource requested by the application. Let R_{ij} denote the fraction of a resource requested by the j^{th} capsule of application i , and let R_i denote the aggregate resource requirement of that application. Assuming that application i has C_i capsules, we have $R_i = \sum_{j=1}^{C_i} R_{ij}$. Observe that $0 < R_i \leq C_i$ since $0 < R_{ij} \leq 1$.

Applications specify their requirements to Sharc using a simple resource specification language [2]. The specification language allows applications the flexibility of either specifying the reservation of each individual capsule or specifying an aggregate reservation for the application without specifying how this aggregate is to be partitioned among individual capsules. The language also allows control over the placement of capsules onto nodes—the application can either specify the precise mapping of capsules onto nodes or leave the mapping to the control plane if any such mapping is acceptable. An application is also allowed to specify if resource trading is permitted for its capsules. Resource trading allows unutilized resources to be temporarily lent to other peer capsules under the condition that they are returned when needed—Sharc adjusts the instantaneous allocations of capsules based on usages when resource trading is permitted.

Given such a resource specification, the admission control algorithm proceeds as follows. First, it ensures that admitting the new application will not exceed the capacity of the cluster. Assuming n nodes and m existing application, the following condition should hold for both the CPU and network bandwidth

$$\sum_{i=1}^m R_i^{cpu} + R_{m+1}^{cpu} \leq n \quad \text{and} \quad \sum_{i=1}^m R_i^{net} + R_{m+1}^{net} \leq n \quad (1)$$

Next the admission controller invokes the capsule placement algorithm. The capsule placement algorithm determines a mapping of capsules onto nodes such that there is sufficient spare capacity on each node for the corresponding capsule. That is, the algorithm determines a mapping *capsule* $j \rightarrow$ *node* k such that $R_{ij}^{cpu} \leq S_k^{cpu}$ and $R_{ij}^{net} \leq S_k^{net}$ where S_k^{cpu} and S_k^{net} are the spare CPU and network capacity on node k . One heuristic to find such a mapping is to create a linear ordering of capsules in approximately decreasing order of their CPU and network bandwidth requirements and then place capsules onto nodes using a best-fit strategy. Since there may be no correlation between the CPU and network requirements of a capsule, such a heuristic may not always find a feasible mapping if one exists. We have considered various strategies for capsule placement, in addition to best-fit, and have compared it to a brute-force policy that searches all possible mappings to find a feasible one [1]. Due to space constraints, we do not report on those results here and assume a simple best-fit strategy for this paper.

The application is admitted into the system if the capsule placement algorithm can compute a feasible mapping. The control plane then creates a new application node in the virtual hierarchy and notifies all affected nuclei, which then update their physical hierarchies. Each nucleus in turn maps capsule reservations onto the OS-specific QoS parameters (as discussed in the next section) and conveys these parameters to the CPU and network interface schedulers.

4.2 Mapping Capsule Requirements to Node-specific QoS Requirements

As explained earlier, Sharc employs an OS-independent representation of the application resource requirements to enable interoperability with different OS platforms as well as to manage heterogeneous clusters consisting of nodes with different operating systems. Given a particular node, the nucleus employs the following techniques to map capsule requirements to OS-specific QoS parameters.

Reservation-based schedulers: A reservation-based scheduler allows resource requirements to be specified in *absolute* terms. Numerous reservation-based schedulers such as Nemesis [19], Rialto [18] and DSRT [20] have been proposed recently. A typical resource specification for such schedulers is a pair (x, y) , where x units of the resource are requested every y time

units (effectively requesting x/y fraction of the resource). In such a scenario, the mapping to OS parameters is trivial—Sharc reservations map directly to the underlying OS reservations.

Proportional-share and Lottery schedulers: In contrast to reservation-based schedulers that allocate resources in absolute terms, proportional-share and lottery schedulers allow resource requirements to be specified in relative terms. In such schedulers, each application is assigned a weight and is allocated resources in proportion to its weight. Thus, applications with weights w_1 and w_2 are allocated resources in ratio $w_1 : w_2$. Whereas proportional-share schedulers achieve such allocations in a deterministic manner, lottery-schedulers use probabilistic techniques (via the notion of a lottery). When used in conjunction with admission control algorithms, these schedulers can provide *absolute* bounds on their allocations [15]. For instance, if an admission controller limits the sum of the weights (or lottery tickets) in the system to 100, then an application with w_i tickets will receive $w_i\%$ of the resource (in absolute terms). Since the Sharc control plane employs admission control, it can guarantee (lower) bounds on the resources allocated to each application when such schedulers are used. In such a scenario, the nucleus maps capsule reservation to weights by setting $w_{ij} = R_{ij}$. Since the pair (x, y) is specified as a *single* parameter x/y to the operating system, the underlying scheduler will only approximate the desired reservation. The nature of approximation depends on the exact scheduling algorithm—the finer the time-scale of the allocation supported by the scheduler, the better will the actual allocation approximate the desired reservation.

Rate regulators: A rate regulator regulates the amount of resources allocated to an application by limiting the number of concurrent requests from that application. Thus, a rate regulator acts as a policing mechanism that complements the resource scheduler and is typically distinct from the scheduler itself. Rate regulators are commonly used to police the network bandwidth allocated to an application. The leaky bucket is an example of a widely used regulator—a leaky bucket ensures the outgoing network packets from an application conforms to the (σ, ρ) profile, where σ is the average bandwidth of the flow and ρ is the maximum allowed burst size. The nucleus can map a network reservation (x, y) to leaky bucket parameters by setting $\sigma = \frac{x}{y} \cdot c$ and $\rho = x \cdot c$, where c denotes the bandwidth capacity of the network interface. Observe that, $\frac{x}{y} \cdot c$ denotes the transmission rate of the application for a reservation (x, y) . Further, since the application can transmit for x time units every y time units, in the worst case, all packets destined for a y time period could arrive simultaneously, necessitating a burst size of $x \cdot c$.

Next, we describe how the Sharc control plane adjusts the resources allocated to capsules based on their usages.

4.3 Trading Resources based on Capsule Needs

Consider a shared cluster with n nodes that runs m applications. Let A_{ij} and U_{ij} denote the current allocation and current resource usage of the j^{th} capsule of application i . A_{ij} and U_{ij} are defined to be the fraction of resource allocated and used, respectively, over a given time interval; $0 \leq U_{ij} \leq 1$ and $0 < A_{ij} \leq 1$. Recall also that R_{ij} is the fraction of the resource requested by the capsule at application startup time. The techniques presented in this section apply to both CPU and network bandwidth—the same technique can be used to adjust CPU and network bandwidth allocations of capsules based on their usages.

The nucleus on each node tracks the resource usage of all capsules over an interval \mathcal{I} and periodically reports the corresponding usage vector $\langle U_{i_1 j_1}, U_{i_2 j_2}, \dots \rangle$ to the control plane. Nuclei on different nodes are assumed to be unsynchronized, and hence, usage statistics from nodes arrive at the control plane at arbitrary instants (but approximately every \mathcal{I} time units). Resource trading is the problem of temporarily increasing or decreasing the reservation of a capsule to match its usage, subject to aggregate reservation constraints for that application. Intuitively, the allocation of a capsule is increased if its past usage indicates it could use additional resources; the allocation of the capsule is decreased if it is not utilizing its reserved share and this unused allocation is then lent to other needy capsules.

To enable such resource trading, the control plane recomputes the instantaneous allocation of all capsules every \mathcal{I} time units. To do so, it first computes the resource usage of a capsule using an exponential smoothing function.

$$U_{ij} = \alpha \cdot U_{ij}^{new} + (1 - \alpha) \cdot U_{ij} \quad (2)$$

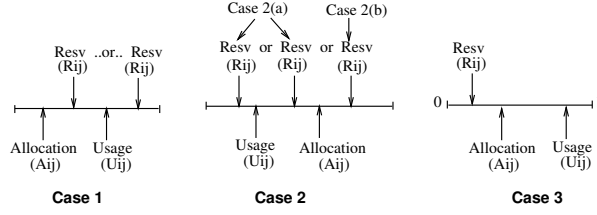


Figure 2: Various scenarios that occur while trading resources among capsules.

where U_{ij}^{new} is the usage reported by the nuclei and α is a tunable smoothing parameter; $0 \leq \alpha \leq 1$. Use of an exponentially smoothed moving average ensures that small transient changes in usages do not result in corresponding fluctuations in allocations, yielding a more stable system behavior. In the event a nucleus fails to report its usage vector (due to clock drift, failures or overload problems, all of which delay updates from the node), the control plane conservatively sets the usages on that node to the initial reservations (i.e., $U_{ij}^{new} = R_{ij}$ for all capsules on that node). As explained in Section 4.4, this assumption also helps deal with possible failures on that node.

Our algorithm to recompute capsule allocations is based on three key principles: (1) Trading of resources among capsules should never violate the invariant $\sum_j A_{ij} = \sum_j R_{ij} = R_i$. That is, redistribution of resources among capsules should never cause the aggregate reservation of the application to be exceeded. (2) A capsule can borrow resources only if there is another capsule of that application that is under-utilizing its allocation (i.e., there exists a capsule j such that $U_{ij} < A_{ij}$). Further there should be sufficient spare capacity on the node to permit borrowing of resources. (3) A capsule that lends its resources to a peer capsule is guaranteed to get it back at any time; moreover the capsule *does not* accumulate credit for the period of time it lends these resources.² Resource trading is only permitted between capsules of the same application, never across applications.

Our re-computation algorithm proceeds in three steps. First, capsules that lent resources to other peer capsules but need them back reclaim their allocations. Second, allocations of under-utilized capsules are reduced appropriately. Third, any unutilized bandwidth is distributed (lent) to any capsules that could benefit from additional resources. Thus, the algorithm proceeds as follows.

Step 0: *Determine capsule allocations when resource trading is prohibited.* If resource trading is prohibited, then the allocations of all capsules of that application are simply set to their reservations ($\forall j, A_{ij} = R_{ij}$) and the algorithm moves on to the next application.

Step 1: *Needy capsules reclaim lent resources.* A capsule is said to have lent bandwidth if its current allocation is smaller than its reservation (i.e., allocation $A_{ij} <$ reservation R_{ij}). Each such capsule signals its desire to reclaim its due share if its resource usage equals or exceeds its allocation (i.e., usage $U_{ij} \geq$ allocation A_{ij}). Figure 2, Case 1 pictorially depicts this scenario.

For each such capsule, the resource manager returns lent bandwidth by setting

$$A_{ij} = \min(R_{ij}, (1 + \epsilon) \cdot U_{ij})$$

where ϵ is a small positive constant, $0 < \epsilon < 1$. Rather than resetting the allocation of the capsule to its reservation, the capsule is allocated the smaller of its reservation and the current usage. This ensures that the capsule is returned only as much bandwidth as it needs (see Figure 2). The parameter ϵ ensures that the new allocation is slightly larger than the current usage, enabling the capsule to (gradually) reclaim lent resources.

Step 2: *Underutilized capsules give up resources.* A capsule is said to be under-utilizing resources if its current usage is strictly smaller than its allocation (i.e., usage $U_{ij} <$ allocation A_{ij}). Figure 2, Case 2 depicts this scenario.

²Accumulating credit for unused resources can cause starvation. For example, a capsule could sleep for an extended duration of time and use its accumulated credit to continuously run on the CPU, thereby starving other applications. Resource schedulers that allow accumulation of credit need to employ techniques to explicitly avoid this problem [6].

Since the allocated resources are under-utilized, the resource manager should reduce the new allocation of the capsule. The exact reduction in allocation depends on the relationship of the current allocation and the reservation. If the current allocation is greater than the reservation (Case 2(a) in Figure 2), then the new allocation is set to the usage (i.e., the allocation of a capsule that borrowed bandwidth but didn't use it is reduced to its actual usage). On the other hand, if the current allocation is smaller than the reservation (implying that the capsule is lending bandwidth), then any further reductions in the allocations are made gradually (case 2(b) in Figure 2). Thus,

$$A_{ij} = \begin{cases} U_{ij} & \text{if } A_{ij} \geq R_{ij} \\ (1 - \epsilon) \cdot A_{ij} & \text{if } A_{ij} < R_{ij} \end{cases} \quad (3)$$

where ϵ is a small positive constant.

After examining capsules of all applications in Steps 1 and 2, the resource manager can then determine the unused resources for each application and the spare capacity on each node; the unused resources can then be lent to the remaining (needy) capsules of these applications.

Step 3: *Needy capsules are lent additional (unused) bandwidth.* A capsule signals its need to borrow additional bandwidth if its usage exceeds its allocation (i.e., usage $U_{ij} \geq$ allocation A_{ij}). An additional requirement is that the capsule shouldn't already be lending bandwidth to other capsules ($A_{ij} \geq R_{ij}$), else it would have been considered in Step 1. Figure 2, Case 3 depicts this scenario.

The resource manager lends additional bandwidth to such a capsule. The additional bandwidth allocated to the capsule is smaller of the spare capacity on that node and the unallocated bandwidth for that application. That is,

$$A_{ij} = A_{ij} + \min\left(\frac{1 - \sum_{j \in \text{node}} A_{ij}}{\mathcal{N}_1}, \frac{R_i - \sum_{j=1}^{C_j} A_{ij}}{\mathcal{N}_2}\right) \quad (4)$$

where $1 - \sum A_{ij}$ is the spare capacity on a node, $R_i - \sum_{j=1}^{C_j} A_{ij}$ is the unallocated bandwidth for the application, and \mathcal{N}_1 and \mathcal{N}_2 are the number of needy capsules on the node and for the application, respectively, all of whom desire additional bandwidth. Thus, the resource manager distributes unused bandwidth equally among all needy capsules.

An important point to note is that the spare capacity on a node or the unallocated bandwidth for the application could be *negative* quantities. This scenario occurs when the amount of resource reclaimed in Step 1 is greater than the unutilized bandwidth recouped in Step 2. In such a scenario, the net effect of Equation 4 is to *reduce* the total allocation of the capsule; this is permissible since the capsule was already borrowing bandwidth which is returned back.³ Thus, Equation 4 accounts for both positive and negative spare bandwidth in one unified step.

Step 4: *Ensure the invariant for the application.* After performing the above steps for all capsules of the application, the resource manager checks to ensure that the invariant $\sum_j A_{ij} = \sum_j R_{ij} = R_i$ holds. Additionally, $\sum_{j \in \text{node}} A_{ij} \leq 1$ should hold for each node. Under certain circumstances, it is possible that the total allocation may be slightly larger or smaller than the aggregate reservation for the application after the above three steps, or an increase in capsule allocation in Step 1 may cause the capacity of the node to be exceeded. These scenarios occur when capacity constraints on a node prevent redistribution of all unused bandwidth or the total reclaimed bandwidth is larger than the total unutilized bandwidth. In either case, the resource manager needs to adjust the new allocations to ensure these invariants. This requires a small, constant number of additional scans of all capsules so as to increase or decrease their allocations slightly using a simple heuristic (details omitted due to space constraints).

The newly computed allocations are then conveyed as an allocation vector $\langle A_{i_1 j_1}, A_{i_2 j_2}, \dots \rangle$ to each nucleus. The nucleus then maps the new reservations ($A_{ij} \cdot y_{ij}, y_{ij}$) to OS-specific QoS parameters as discussed in Section 4.2 and conveys them to the OS scheduler.

³For simplicity of exposition, we omitted one detail in Eq. 4. After computing A_{ij} in Eq. 4, the allocation is constrained as $A_{ij} = \max(A_{ij}, R_{ij})$ to prevent it from becoming smaller than R_{ij} when the spare capacity is negative.

A salient feature of the above algorithm is that it has two tunable parameters—the interval length \mathcal{I} and the smoothing parameter α . As will be shown experimentally in Section 6, use of a small re-computation interval \mathcal{I} enables fine-grain resource trading based on small changes in resource usage, whereas a large interval focuses the algorithm on long-term changes in resource usage of capsules. Similarly, a large α causes the resource manager to focus on immediate past usages while computing allocations, while a small α smoothes out the contribution of recent usage measurements. Thus, \mathcal{I} and α can be chosen appropriately to control the sensitivity of the algorithm to small, short-term changes in resource usage.

4.4 Failure Handling in Sharc

In this section, we provide a brief overview of the failure recovery mechanisms employed by Sharc (the details of these mechanisms can be found in the extended version of this paper [2]). Sharc can handle three types of failures—nucleus failure, control plane failure, and node and link failures. The key principle employed by Sharc to recover from these failures is replication of state information—information replicated in the virtual and physical hierarchies enables Sharc to reconstruct state lost due to a failure. Sharc deals with failures as follows.

Nucleus failure: A nucleus failure occurs when the nucleus on a node fails but the node itself remains operational. The control plane uses heartbeat messages to monitor the health of each nucleus. Upon detecting a failure, the control plane starts up a new nucleus on the node, reconstructs the state of its physical hierarchy (using the virtual hierarchy), and synchronizes its state with the nucleus. The allocations of capsules is also reset to their initial startup values (i.e., R_{ij}).

Control plane failure: A control plane failure is caused by the failure of the node running the control plane or the failure of the control plane itself. In either case, the control plane becomes unreachable from the nuclei. In such an event, the nuclei run a distributed leader election algorithm to elect a new node, subject to the constraint that this node should have sufficient free resources to run control plane tasks. A new control plane is started on this node, which then reconstructs the virtual hierarchy by fetching the physical hierarchies from all nuclei.

Node and link failures: A node failure occurs when the operating system on a node crashes due to a software or hardware fault. A link failure occurs when the link connecting the node to the cluster interconnect fails. From the perspective of the control plane, both kinds of failures have the same effect—the node becomes unreachable. Whereas recovering from a node or link failure requires human intervention (to reboot the system or to repair faults), the control plane can aid the recovery process. Upon detecting an unreachable node, the control plane can automatically reassign any capsule running on that node to other nodes in the cluster (by rerunning admission control and capsule placement algorithms). The affected capsules can then be restarted on that node. Note that this process only helps in the transition of failed capsules to new nodes and doesn't help recover the state of these capsules (which is left to the application).

5 Implementation Considerations and Complexity

The complexity of the mechanisms employed by the control plane and the nuclei is as follows.

Admission Control and Capsule Placement. For each new application, the control plane first sorts capsules in order of their CPU and network requirements, which requires two $O(k \cdot \log k)$ operations for k capsules. Capsules are then linearly ordered in approximately decreasing order of their CPU and network requirements. Assuming a best-fit placement strategy, a linear scan of all nodes is needed in order to determine the “best” node to house each capsule. This is a $O(n \cdot k)$ operation for a cluster of n nodes. Thus, the overall complexity of admission control and capsule placement is $O(n \cdot k + k \cdot \log k)$.

Resource trading. The resource trading algorithm described in Section 4.3 proceeds one application at a time; capsules of an application need to be scanned a constant number of times to determine their new allocations (once for the first three steps and a constant number of times in Step 4). Thus, the overall complexity is linear in the number of capsules and takes $O(mk)$ time in a system with m applications, each with k capsules (total of mk capsules). Each nucleus on a node participates in this process by determining resource usages of capsules and setting new allocations; the overhead of these tasks is two system calls

every \mathcal{I} time units. Thus, the overall complexity of resource trading is linear in the number of capsules, which is more efficient than the polynomial time complexity of prior approaches [3].

Communication overheads. The number of bytes exchanged between the control plane and the various nuclei is a function of the total number of capsules in the system and the number of nodes. Although the precise overhead is $\beta \cdot n + \beta' \cdot mk$, it reduces to $O(mk)$ bytes in practice, since $mk \gg n$ in shared clusters (β, β' are constants).

We have implemented a prototype of Sharc on a cluster on Linux PCs.⁴ We chose Linux as the underlying operating system since implementations of several commonly used QoS scheduling algorithms are available for Linux, allowing us to experiment with how capsule reservations in Sharc map onto different QoS parameters supported by these schedulers. Briefly, our Sharc prototype consists of two components—the control plane and the nucleus—that run as privileged processes in user space and communicate with one another on well-known port numbers. The implementation is multi-threaded and is based on Posix threads. The control plane consists of threads for (i) admission control and capsule placement, (ii) resource management and trading, (iii) communication with the nuclei on various nodes, and (iv) for handling nucleus and node failures. The resource specification language described in Section 4 is used to allocate resources to new applications, to modify resources allocated to existing applications, or to terminate applications and free up allocated resources. Each nucleus consists of threads that track resource usage, communicate with the control plane, and handle control plane failures. For the purposes of this paper, we chose a Linux kernel that implements the H-SFQ proportional-share scheduler [22] and the leaky bucket rate regulator for allocating CPU and network interface bandwidth, respectively. This allows us to demonstrate that Sharc can indeed inter-operate with different kinds of kernel resource management mechanisms.

Next, we discuss our experimental results.

6 Experimental Evaluation

In this section, we experimentally evaluate our Sharc prototype using two types of workloads—a commercial third-party hosting platform workload and a research workgroup environment workload. Using these workloads and micro-benchmarks, we demonstrate that Sharc: (i) provides predictable allocation of CPU based on the specified resource requirements, (ii) can isolate applications from one another, (iii) can scale to clusters with a few hundred nodes running 100,000 capsules, and (iv) can handle a variety of failure scenarios. In what follows, we first describe the test-bed for our experiments and then describe our experimental results.

6.1 Experimental Setup

The testbed for our experiments consists of a cluster of Linux-based workstations interconnected by a 100 Mb/s switched ethernet. Our experiments assume that all machines are lightly loaded and so is the network. Unless specified otherwise, the Sharc control plane is assumed to run on a dedicated cluster node, as would be typical on a third-party hosting platform.

Our experiments involved two types of workloads. Our first workload is representative of a third-party hosting platform and consists of the following applications: (i) an ecommerce application consisting of a front-end web server and a back-end relational database, (ii) a replicated web server that uses Apache version 1.3.9, (iii) a file download server that supports download of large audio files, and (iv) a home-grown streaming media server that streams 1.5 Mb/s MPEG-1 files. Our second workload is representative of a research workgroup environment and consists of (i) *Scientific*, a compute-intensive scientific application that involved matrix manipulations, (ii) *Summarize*, an information retrieval application, (iii) *Disksim*, a publicly-available compute-intensive disk simulator, and (iv) *Make*, an application build job that compiles the Linux 2.2.0 kernel using GNU make.

Next, we present the results of our experimental evaluation using these applications.

⁴The Sharc system is publicly available for download. The URL for download has been withheld for purposes of blind reviewing. Interested reviewers may contact us with the permission of the program chair.

Table 1: Capsule Placement and Reservations

Applications	Capsules & % (cpu,net) Reservations				Workload	Resource trading
	N1	N2	N3	N4		
Ecommerce 1 (EC1)	(10,5)	(10,5)	–	–	Mixed	no
Ecommerce 2 (EC2)	–	(10,5)	(10,5)	–	Mixed	yes
File download (FD)	(5,10)	–	(5,10)	(5,10)	I/O intensive	yes
Streaming (S1)	–	–	(5,5)	(5,5)	I/O intensive	no
Streaming (S2)	(5,5)	(5,5)	–	–	I/O intensive	no
Dynamic HTTP server (WS)	–	(20,5)	–	(20,5)	CPU intensive	yes

6.2 Predictable Resource Allocation and Application Isolation

Our first experiment demonstrates the efficacy of CPU and network interface bandwidth allocation in Sharc. We emulate a shared hosting platform environment with six applications. The placement of various application capsules and their CPU and network reservations are depicted in Table 1. Our first two applications are ecommerce applications with two capsules each—a front-end web server and a back-end database server. For both applications, a fraction of requests received by the front-end web server are assumed to trigger (compute-intensive) transactions in the database server (to simulate customer purchases on the ecommerce site). Our file download application emulates a music download site that supports audio file downloads; its workload is predominantly I/O intensive. Each streaming server application streams 1.5 Mb/s MPEG-1 files to multiple clients, while the web server application services dynamic HTTP requests (which involves dynamic HTML generation via Apache’s PHP3 scripting). For the purposes of this experiment, we focus on the behavior of the first three applications, namely the two ecommerce applications and the file download server. The other three applications serve as the background load for our experiments.

To demonstrate the efficacy of CPU allocation in Sharc, we introduced identical, periodic bursts of requests in the two ecommerce applications. Resource trading was turned off for the first application and was permitted for the other. Observe that each burst triggers compute-intensive transactions in the database capsules. Since resource trading is permitted for EC2, the database capsule can borrow CPU cycles from the web server capsule (which is I/O intensive) and use these borrowed cycles to improve transaction throughput. Since resource trading is prohibited in EC1, the corresponding database capsule is unable to borrow additional resources, which affects its throughput. Figure 3 plots the CPU allocations of the various capsules for the two applications and the throughput of both applications. The figure shows that trading CPU resources in EC2 allows it to process each burst faster than EC1. Specifically, trading CPU bandwidth among its capsules enables the database capsule of EC2 to finish the two bursts 85 sec and 25 sec faster, respectively, than the database capsule of EC1.

Next we demonstrate the efficacy of network bandwidth allocation in Sharc. We consider the file download application that has three replicated capsules. To demonstrate the efficacy of resource trading, we send a burst of requests at $t = 70$ sec to the application; the majority of these requests go to the first capsule and the other two capsules remain underloaded. To cope with the increased load, Sharc reassigns unused bandwidth from the two under-loaded capsules to the overloaded capsule. We then send a second similar burst at $t = 160$ sec and observe a similar behavior. We send a third burst at $t = 300$ sec that is skewed towards the latter two capsules, leaving the first capsule with unused bandwidth. In this case, both overloaded capsules borrow bandwidth from the underutilized capsule; the borrowed bandwidth is shared equally among the two overloaded capsule. Finally, at $t = 500$ sec, a similar simultaneous burst is sent to the two capsules again with similar results. Figure 4 plots the network allocations of the three capsules and demonstrates the above behavior.

An interesting feature exhibited by these experiments is related to the exponential smoothing parameter α mentioned in Section 4.3. For CPU bandwidth allocation, α was chosen to be 1.0 (no history), causing Sharc to reallocate bandwidth to the database capsule of EC2 very quickly. For network bandwidth allocation, α was chosen to be 0.5 resulting in a more gradual trading of network bandwidth among the capsules of the file download application. Figures 3 and 4 depict this behavior. Thus,

the value of α can be used to control the sensitivity of resource trading. One additional aspect of the above experiments (not shown here due to space constraints) is that Sharc isolates the remaining three applications, namely S1, S2 and WS, from the bursty workloads seen by the first three applications. This is achieved by providing each of these applications with a guaranteed resource share, which is unaffected by the bursty workloads of the ecommerce and file download applications.

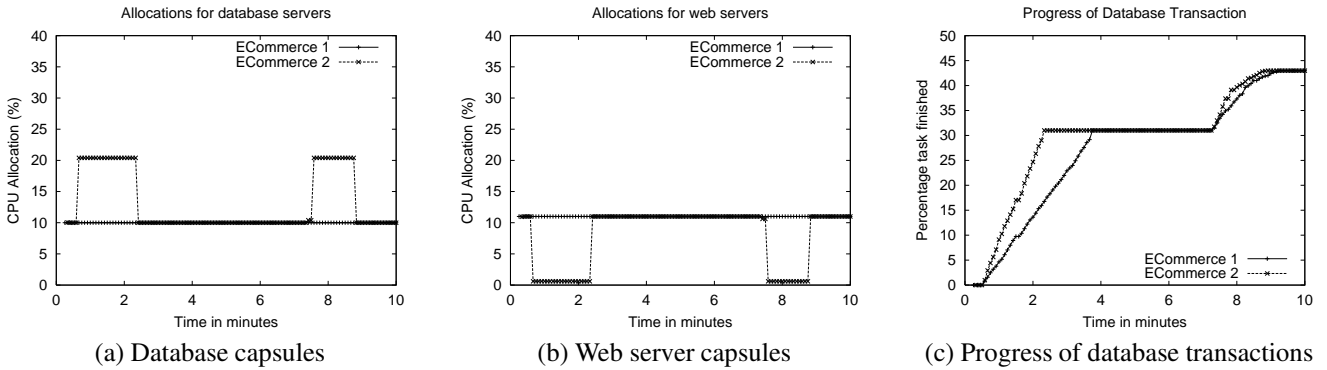


Figure 3: Predictable CPU allocation and trading. Figures (a) and (b) show the CPU allocation for the database server and the web server capsules, Figure (c) shows the progress of the two bursts processed by these database servers.

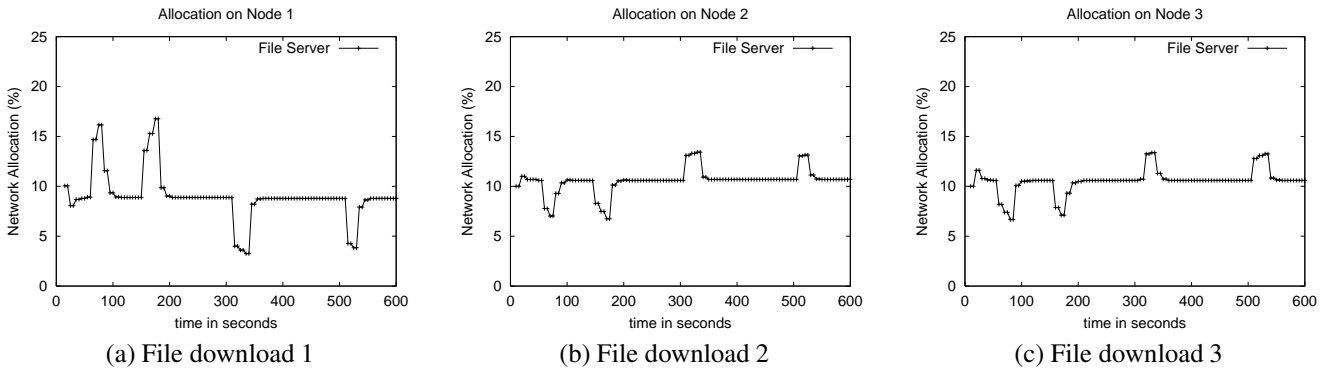


Figure 4: Predictable network allocation and trading. Figure (a), (b) and (c) depict network allocations of capsules of the File download application.

6.3 Performance of a Scientific Application Workload

We conducted an experiment to demonstrate resource sharing among four applications representing a research workgroup environment. The placement of various capsules and their CPU reservations are listed in Table 2 (since these applications are compute-intensive, we focus only on CPU allocations in this experiment). As shown in the table, the first two applications arrive in the first few minutes and are allocated their reserved shares by Sharc. The capsule of the scientific application running on node 2 is put to sleep at $t = 25\text{min}$, until $t = 38\text{min}$. This allows the other capsules of that application on nodes 3 and 4 to borrow bandwidth unused by the sleeping capsule. The diskSim application arrives at $t = 36\text{min}$ and the bandwidth borrowed on node 3 by the scientific application has to be returned (since the total allocation on the node reaches 100%, there is no longer any spare capacity on the node, preventing any further borrowing). Finally, two kernel builds startup at $t = 37\text{min}$ and are allocated their reserved shares. We measured the CPU allocations and the actual CPU usages of each capsule. Since there are ten capsule in this experiment, due to space constraints, we only present results for the three capsules on node 3. As shown in

Table 2: Capsule Placement and Reservations

Applications	Arrival (min)	Capsules & their Reservations			
		N1	N2	N3	N4
Summarizer	1	20%	30%	20%	—
Scientific	2.5	—	20%	30%	20%
Disksim	36	50%	—	50%	—
Make	37	—	50%	—	50%

Figure 5, the allocations of the three capsule closely match the above scenario. The actual CPU usages are initially larger than the allocations, since SFQ is a fair-share CPU scheduler and fairly redistributes unused CPU bandwidth on that node to runnable capsules (regardless of their allocations). Note that, at $t = 36$ min, the total allocation reaches 100%; at this point, there is no longer any unused CPU bandwidth that can be redistributed and the CPU usages closely match their allocations as expected. Thus, a proportional-share scheduler behaves exactly like a reservation-based scheduler at full capacity, while redistributing unused bandwidth in presence of space capacity; this behavior is independent of Sharc, which continues to allocate bandwidth to capsules based on their initial reservations and instantaneous needs.

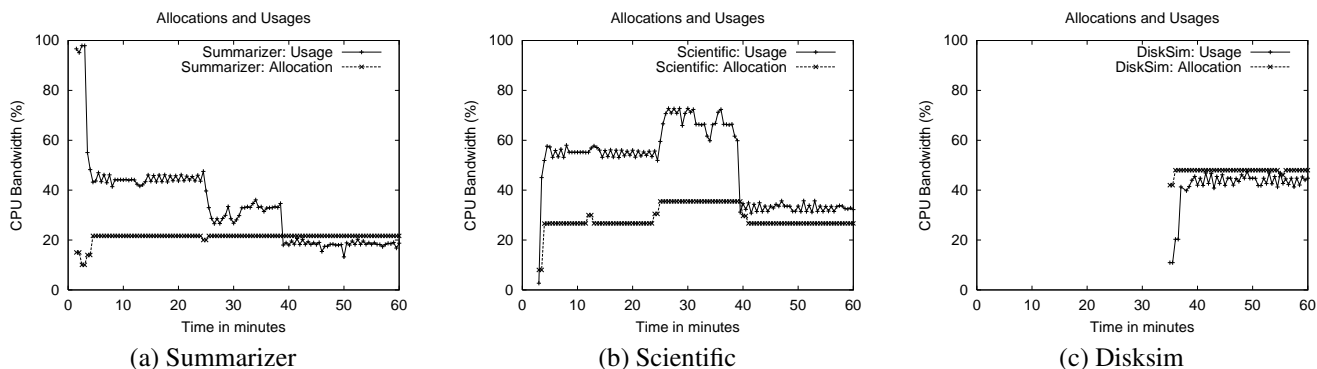


Figure 5: Predictable allocation and resource trading. Figure (a), (b) and (c) depict CPU usages and allocations of capsules residing on node 3.

6.4 Impact of Resource Trading

To show that resource trading can help applications provide better quality of service to end-users, we conducted an experiment with a streaming video server. The server has two capsules, each of which streams MPEG-1 video to clients. We configure the server with a total network reservation of 8 Mb/s (4 Mb/s per capsule). At $t=0$, each capsule receives two requests each for a 15 minute long 1.5 Mb/s video and starts streaming the requested files to clients. At $t=5$ min, a fifth request for the video arrives and the first capsule is entrusted with the task of servicing the request. Observe that the capsule has a network bandwidth reservation of 4 Mb/s, whereas the cumulative requirements of the three requests is 4.5 Mb/s. We run the server with resource trading turned on, and then repeat the entire experiment with resource trading turned off. When resource trading is permitted, the first capsule is able to borrow unused bandwidth from the second capsule and service its two clients at their required data rates. In the absence of resource trading, the token bucket regulator restricts the total bandwidth usage to 4 Mb/s, resulting in late packet arrivals at the three clients. To measure the impact of these late arrivals on video playback, we assume that each client can buffer 4 seconds of video and that video playback is initiated only after this buffer is full. We then measure the number of playback discontinuities that occur due to a buffer underflow (after each such glitch, the client is assumed to pause until the buffer fills up again). Figure 6(a) plots the number of discontinuities observed by the clients of the first capsule in the

two scenarios. The figure shows that when resource trading is permitted, there are very few playback discontinuities (the two observed discontinuities are due to the time lag in lending bandwidth to the first capsule—the control plane can react only at the granularity of the re-computation period \mathcal{I} , which was set to 5 seconds in our experiment). In contrast, lack of resource trading causes a significant degradation in performance. Figures 6(b) and (c) show a 150 second long snapshot of the reception and playback of one of the streams provided by the first capsule (stream 2) for the two cases. Observe that the client is receiving data at nearly 1.5 Mbps when trading is allowed, but only at about 1.4 Mbps in the absence of trading. As shown in Figure 6(b), there are repeated buffer underflows (represented by the horizontal portions of the plot) due to the bandwidth restrictions imposed by the rate regulator. Thus, the experiment demonstrates the utility of resource trading in improving application performance.

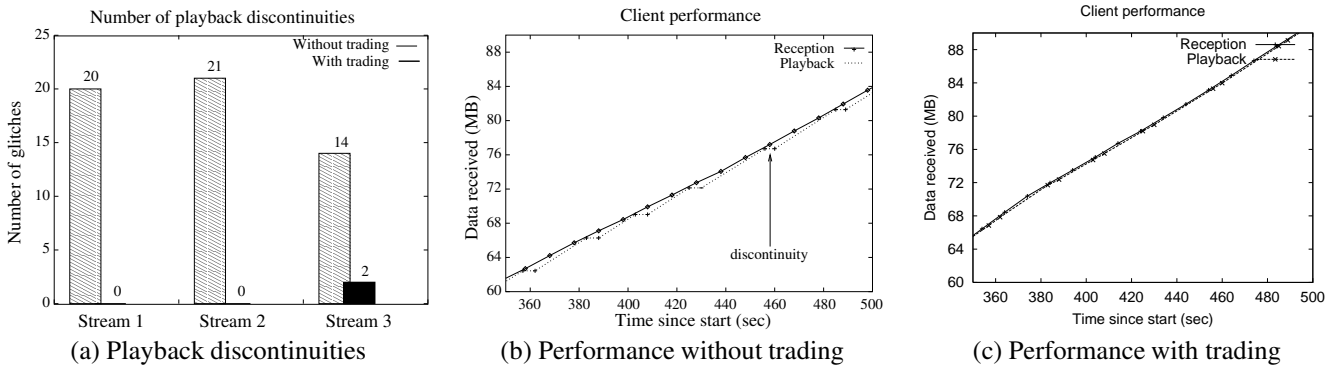


Figure 6: Impact of resource trading. Figure (a) shows the number of playback discontinuities seen by the three clients of the overloaded video server with and without the trading of network bandwidth. Figures (b) and (c) show a portion of the reception and playback of the second stream for the two cases.

6.5 Scalability of Sharc

To demonstrate the scalability of Sharc, we conducted experiments to measure the CPU and communication overheads imposed by the control plane and the nucleus. Observe that these overheads depend solely on the *number* of capsules and nodes in the system and are relatively independent of the *characteristic* of each capsule. The experiments reported in this section were conducted by running the control plane and the nuclei on 1 GHz Pentium III workstations with 256MB memory running RedHat Linux version 6.2

6.5.1 Overheads Imposed by the Nucleus

We first measured the CPU overheads of the nucleus for varying loads; the usages were computed using the `times` system call and profiling tools such as `gprof`. We varied the number of capsules on a node from 10 to 10,000 and measured the CPU usage of the nucleus for different interval lengths. Figure 7(a) plots our results. As shown, the CPU overheads decrease with increasing interval lengths. This is because the nucleus needs to query the kernel for CPU and network bandwidth usages and notify it of new allocations once in each interval \mathcal{I} . The larger the interval duration, the less frequent are these operations, and consequently, the smaller is the resulting CPU overhead. As shown in the figure, the CPU overheads for 1000 capsules was less than 2% when $\mathcal{I} = 5s$. Even with 10,000 capsules, the CPU usage was less than 4% when $\mathcal{I} = 20s$ and less than 3% when $\mathcal{I} = 30s$.

Figure 7(b) plots the system call overhead incurred by the nucleus for querying CPU and network bandwidth usages and for notifying new allocations. As shown, the overhead increases linearly with increasing number of capsules; the average overhead of these system calls for 500 capsules was only $497\mu s$ and $297\mu s$, respectively

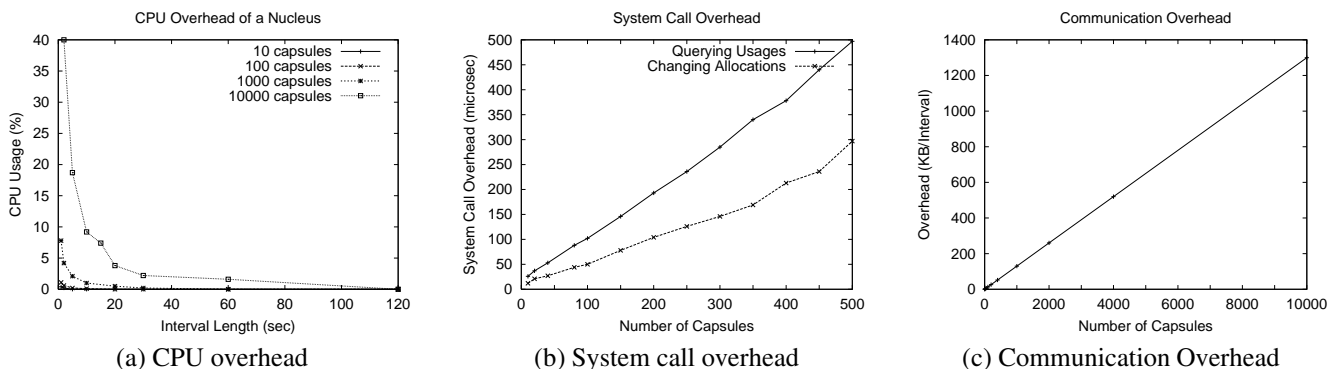


Figure 7: Overheads imposed by the nucleus.

Figure 7(c) plots the communication overhead incurred by the nucleus for varying number of capsules. The communication overhead is defined to be the total number of bytes required to report the usage vector to the control plane and receive new allocations for capsules. As shown in the Figure, when $\mathcal{I} = 30$ s, the overhead is around 1300KB for 10,000 capsules (43.3 KB/s) and is around 130KB per interval (4.3 KB/s) for 1000 capsules. Together these results show that the overheads imposed by the nucleus for most realistic workloads is small in practice.

6.5.2 Control Plane Overheads

Next we conducted experiments to examine the scalability of the control plane. Since we were restricted by a five PC cluster, we emulated larger clusters by starting up multiple nuclei on each node and having each nucleus emulate all operations as if it controlled the entire node. Due to memory constraints on our machines, we didn't actually start up a large number of applications but simulated them by having the nuclei manage the corresponding physical hierarchies and report varying CPU and network bandwidth usages. The nuclei on each node were unsynchronized and reported usages to the control plane every \mathcal{I} time units. From the perspective of the control plane, such a setup was no different from an actual cluster with a large number of nodes.

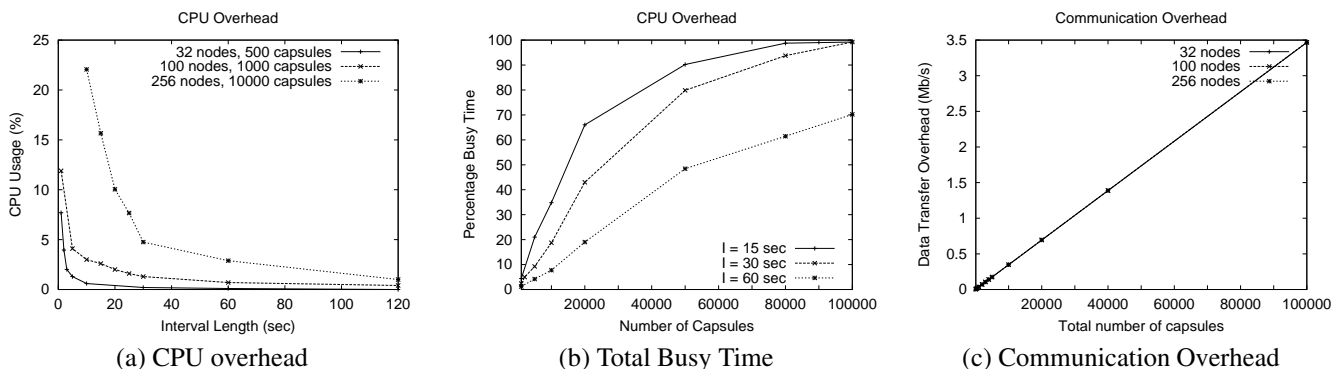


Figure 8: Overheads imposed by the control plane.

Figure 8(a) plots the CPU overhead of the control plane for varying cluster sizes and interval lengths. The figure shows that a control plane running on a dedicated node can easily handle the load imposed by a 256 node cluster with 10,000 capsules (the CPU overhead was less than 16% when $\mathcal{I} = 30$ s). Figure 8(b) plots the total busy time for a 256 node cluster. The busy time is defined to be the total CPU overhead plus the total time to send and receive messages to all the nuclei. As shown in the figure,

the control plane can handle up to 100,000 capsules before reaching saturation when $\mathcal{I} = 30$ s. Furthermore, smaller interval lengths increase these overheads, since all control plane operations occur more frequently. This indicates that a larger interval length should be chosen to scale to larger cluster sizes. Finally, Figure 8(c) plots the total communication overhead incurred by the control plane. Assuming $\mathcal{I} = 30$ s, the figure shows that a cluster of 256 nodes running 100,000 capsules imposes an overhead of 3.46Mb/s, which is less than 4% of the available bandwidth on a FastEthernet LAN. The figure also shows that the communication overhead is largely dominated by the number of capsules in the system and is relatively independent on the number of nodes in the cluster.

6.6 Effect of Tunable Parameters

To demonstrate the effect of tunable parameters \mathcal{I} and α , we used the same set of workgroup applications described in Table 2. We put a capsule of the scientific application to sleep for a short duration. We varied the interval length \mathcal{I} and measured its impact on the allocation of the capsule. As shown in Figure 9(a), increasing the interval length causes the CPU usage to be averaged over a larger measurement interval and diminishes the impact of the transient sleep on the allocation of the capsule (with a large \mathcal{I} of 5min the effect of the sleep was negligibly small on the allocation). Next we put a capsule of Disksim to sleep for a few minutes and measured the effect of varying α on the allocations. As shown in Figure 9(b), use of a large α makes the allocation more sensitive to such transient changes, while a small α diminishes the contribution of transient changes in usage on the allocations. This demonstrates that an appropriate choice of \mathcal{I} and α can be used to control the sensitivity of the allocations to short-term changes in usage.

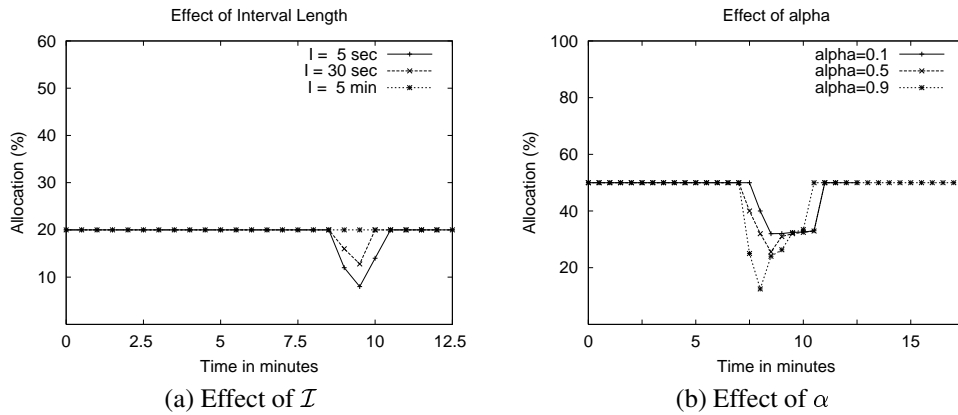


Figure 9: Impact of tunable parameters on capsule allocations.

6.7 Handling Failures

We used fault injection to study the effect on failures in Sharc. We ran 100 capsules of our workgroup applications on each of the four nodes and ran the control plane on a dedicated node and set $\mathcal{I} = 30$ s. We killed the nucleus on various nodes at random time instants and measured the times to detect and recover from the failure. As shown in Table 3, the control plane was able to detect the failure in 80.7s (around $2.5 * \mathcal{I}$). Once detected, starting up a new nucleus remotely took around 11.13 sec, while reconstructing the 100 node physical hierarchy and resynchronizing state with the nucleus took an additional 54ms (total recovery time was 11.18s). Next we studied the effect of node failures by halting the OS on nodes at arbitrary time instants. Detecting a node failure took around 79.27s; the control plane then attempted to reassign the 100 capsules on the failed node to other nodes. The resulting admission control, capsule placement and sending updates to nuclei took 55.1ms. In one case, we used a heavily loaded system, and as expected, the control plane signalled its inability to reassign capsules to other nodes due to

Table 3: Failure Handling Times (with 95% Confidence Intervals)

Failure type	Time to detect	Time to recover
Nucleus	80.7s \pm 5.91	11.18s \pm 0.45
Node	79.27s \pm 5.79	55.1ms \pm 3.89
Control plane	19.85s \pm 5.89	17.41s \pm 1.99

lack of sufficient resources. Finally, we studied the impact of control plane failures. The control plane was run on a dedicated cluster node and was killed at random instants. The nuclei were able to detect the failure in 19.8s; running the election algorithm took 16.63s, starting up a new control plane took 9.45ms, while reconstruction of the 400 capsule virtual hierarchy took another 294.9ms (total recovery time was 17.41s). Our current prototype can only handle the case where a control plane running on a dedicated node fails; handling the failure of a control plane that runs on a node with active capsules is more complex and is not currently handled.

7 Directions for Future Work

In this section, we present some limitations of our current design and discuss directions for future research.

Heterogeneous clusters: Our current design assumes that all nodes in the cluster are homogeneous. We are currently enhancing Sharc to accommodate nodes with different characteristics (e.g., different number of processors, different processor speeds, multiple network interfaces). Our approach involves modeling the slowest node in the system as a unit resource and modeling resources on other nodes relative to this node. The admission control, capsule placement, and resource trading techniques will need to be modified accordingly to account for this heterogeneity.

Security considerations: We are examining various security implications of running untrusted applications on shared clusters. One of our goals is to prevent malicious applications from sending fake messages by masquerading as the nucleus or the control plane (communication between the nucleus and the kernel requires root privileges and is more difficult to compromise). Public key cryptography is one possible approach to address this issue—all communications between the nucleus and the control plane is encrypted using the public key of the recipient and digitally signed using the private key of the sender. We also plan to study the performance implications of using encryption on the scalability of the system.

Resource overbooking and trust: Currently the control plane does not support overbooking of resources on a node or take trust among applications into account during capsule placement. Overbooking of resources to extract statistical multiplexing gains requires more sophisticated admission control techniques as well as enhanced techniques to map reservations on an overbooked node to underlying CPU reservations or shares. Similarly, taking trust among applications into account, in addition to resource availability, during capsule placement makes placement a multi-dimensional optimization problem with several constraints. The design of such techniques is the subject of future research.

8 Related Work

Several techniques for predictable allocation of resources within a single machine have been developed over the past decade [7, 11, 18, 19, 27]. A key contribution of Sharc is to extend the benefits of such single node resource management techniques to clustered environments.

Research on clustered environments has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [21]. The design of scalable, fault-tolerant network services running on server clusters has been studied in [12, 16]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in [13]. Scalability, availability and performance issues in dedicated clusters

have been studied in the context of clustered mail servers [24] and replicated web servers [4]. Numerous middleware-based approaches for clustered environments have also been proposed [9, 10]. Finally, gang scheduling and co-scheduling efforts have investigated the issue of coordinating the scheduling of tasks in distributed systems [5, 17]; however, neither approach supports resource reservations, which is a particular focus of our work.

Two recent efforts have focused on the specific issue of resource management in shared commodity clusters. A proportional-share scheduling technique for a network of workstations was proposed in [6]. Whereas there are some similarities between their approach and Sharc, there are some notable differences. The primary difference is that their approach is based on fair *relative* allocation of cluster resources using proportional-share scheduling, whereas we focus on *absolute* allocation of resources using reservations (reservations and shares are fundamentally different resource allocation mechanisms). Even with an underlying proportional-share scheduler, Sharc can provide absolute bounds on allocations using admission control—the admission controller guarantees resources to applications and constrains that the underlying proportional-share scheduler to fair redistribution of *unused* bandwidth (instead of fair allocation of the *total* bandwidth as in [6]). A second difference is that lending resources in [6] results in accumulation of credit that can be used by the task at a later time; the notion of lending resources in Sharc is inherently different—no credit is ever accumulated and trading is constrained by the aggregate reservation for an application. The Cluster Reserves work has also investigated resource allocation in server clusters [3]. The work assumes a large application running on a cluster, where the aim is to provide differential service to clients based on some notion of service *class*. The approach uses resource containers [7] and employs a linear programming formulation for allocating resources, resulting in polynomial time complexity. In contrast, techniques employed by Sharc have complexity that is linear in the number of capsules. Further, Sharc can manage both CPU and network interface bandwidth, whereas Cluster Reserves only support CPU allocation (the technique can, however, be extended to manage network interface bandwidth as well). Finally, unlike Sharc, neither of these efforts have considered failure handling techniques.

9 Concluding Remarks

In this paper, we argued the need for effective resource control mechanisms for sharing resources in commodity clusters. To address this issue, we presented the design of Sharc—a system that enables resource sharing in such clusters. Sharc depends on resource control mechanisms such as reservations or shares in the underlying OS and extends the the benefits of such mechanisms to clustered environments. The control plane and the nuclei in Sharc achieve this goal by (i) supporting resource reservation for applications, (iii) providing performance isolation and dynamic resource allocation to application capsules, and (iv) providing high availability of cluster resources. Our evaluation of the Sharc prototype showed that Sharc can scale to 256 node clusters running 100,000 capsules. Our results demonstrated that a system such as Sharc can be an effective approach for sharing resources among competing applications in moderate size clusters.

References

- [1] Anonymous. Capsule Placement Strategies for Shared Hosting Platforms. Technical report, August 2001.
- [2] Anonymous. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. Technical report, October 2001.
- [3] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference, Santa Clara, CA*, June 2000.
- [4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA*, June 2000.
- [5] A. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Trans. on Computer Systems*, 19(3), August 2001.
- [6] A. Arpaci-Dusseau and D.E. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, NV*, June 1997.
- [7] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the third Symposium on Operating System Design and Implementation (OSDI'99), New Orleans*, pages 45–58, February 1999.

- [8] J. Blanquer, J. Bruno, M. McShea, B. Ozden, A. Silberschatz, and A. Singh. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the FreeBSD'99 Conference, Berkeley, CA*, October 1999.
- [9] Corba Documentation. Available from <http://www.omg.org>.
- [10] Distributed Computing Environment Documentation. Available from <http://www.opengroup.org>.
- [11] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 261–276, December 1999.
- [12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France*, pages 78–91, December 1997.
- [13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'99), Kiawah Island Resort, SC*, pages 154–169, December 1999.
- [14] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of Operating System Design and Implementation (OSDI'96), Seattle*, pages 107–122, October 1996.
- [15] P. Goyal, S. S. Lam, and H. M. Vin. Determining End-to-End Delay Bounds In Heterogeneous Networks. *ACM/Springer-Verlag Multimedia Systems Journal*, 5(3):157–163, May 1997.
- [16] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA*, pages 319–332, October 2000.
- [17] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of Gang Scheduling on a Workstation Cluster. In *Proceedings of the IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 27–40, 1996.
- [18] M. B. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pages 198–211, December 1997.
- [19] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, September 1996.
- [20] C. Lin, H. Chu, and K. Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium, Seattle, WA*, August 1998.
- [21] M. Litzkow, M. Livny, and Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.
- [22] QLinux Software Distribution. Available from <http://lass.cs.umass.edu/software/qlinux>, 1999.
- [23] T. Roscoe and B. Lyles. Distributing Computing without DPEs: Design Considerations for Public Computing Platforms. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, September 2000.
- [24] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: A Highly Available, Scalable Cluster-based Mail Service. In *Proceedings of the 17th SOSP, Kiawah Island Resort, SC*, pages 1–15, December 1999.
- [25] REACT: IRIX Real-time Extensions. Silicon Graphics, Inc., <http://www.sgi.com/software/react>, 1999.
- [26] Solaris Resource Manager 1.0: Controlling System Resources Effectively. Sun Microsystems, Inc., <http://www.sun.com/software/white-papers/wp-srm/>, 1998.
- [27] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of ASPLOS-VIII, San Jose, CA*, pages 181–192, October 1998.