Sharc: Managing CPU and Network Bandwidth in Shared Clusters

Bhuvan Urgaonkar and Prashant Shenoy, *Member*, *IEEE*

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 1) support reservation of CPU and network interface bandwidth for distributed applications, 2) dynamically allocate resources based on past usage, and 3) 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.

Index Terms—Shared clusters, dedicated clusters, Sharc, capsule, nucleus, control plane, CPU and network bandwidth, Linux, hosting platforms.

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 [25]. 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 [26], 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.

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 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

• *The authors are with the Department of Computer Science, University of Massachusetts Amherst, 140 Governor's Drive, Amherst, MA 01003. E-mail: {bhuvan, shenoy}@cs.umass.edu.*

Manuscript received 12 Aug. 2002; revised 24 Apr. 2003; accepted 28 Apr. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 117124.

1. As 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.

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 1) provide predictable allocation of CPU and network interface bandwidth, 2) isolate applications from one another, and 3) handle a variety of failure scenarios. A key advantage of our approach is its efficiency—unlike previous approaches [2] 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 related work in Section 7. Finally, Section 8 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.

2.1 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, e-commerce, and peer-to-peer applications. Similarly, shared clusters in workgroup environments will run a mix of scientific applications, simulations, and batch jobs. In addition to heterogeneity across applications, there could be heterogeneity *within* each application. For instance, an e-commerce 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.

2.2 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 timescale.

Resources could be reserved either based on the aggregate needs of the application (e.g., for a replicated Web server where the total throughput is of greater concern than the throughput of any individual replica) or based on the needs of individual capsules (e.g., an e-commerce application consisting of capsules performing different tasks and having diverse requirements). 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.

2.3 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.

2.4 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.

2.5 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

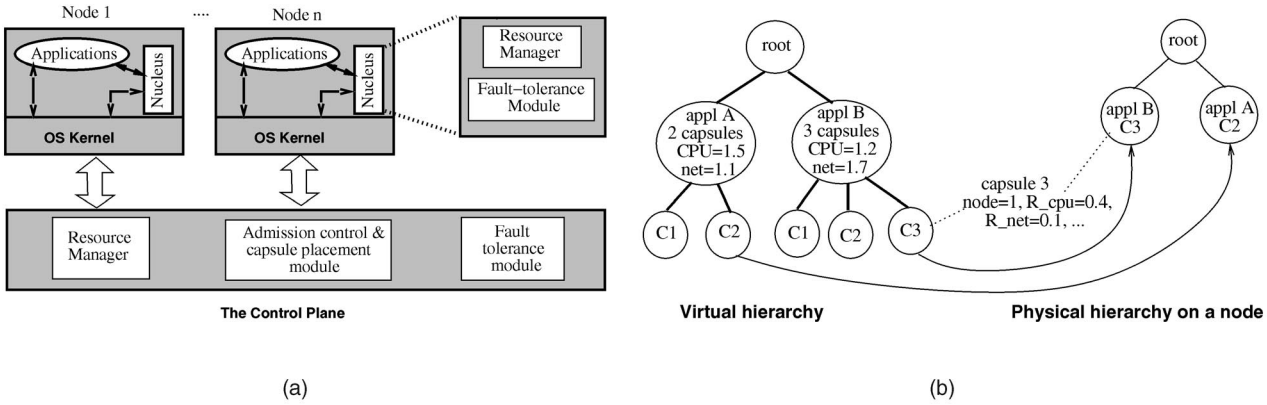


Fig. 1. Sharc architecture and abstractions. (a) shows the overall Sharc architecture. (b) shows a sample cluster-wide virtual hierarchy, a physical hierarchy on a node and the relationship between the two.

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.

2.6 Compatibility with Existing OS Interfaces

Whereas the use of a middleware is one approach for managing resources in clustered environments [7], [8], this approach typically requires applications to use the interface exported by the middleware 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 nonstandard 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 supports some notion of quality of service such as reservations [16], [17], or shares [12]. Many commercial and open-source operating systems such as Solaris [28], IRIX [23], and FreeBSD [5] 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 Fig. 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 Fig. 1a). 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.[2] 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 Fig. 1a, 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: 1) it ensures that sufficient resources exist for each new application, and 2) 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

2. Note that it is not mandatory for applications to reserve resources with Sharc before they are started on the cluster. An application may choose not to reserve any resources. Different policies are possible for allocating resources to such applications. In our Sharc protoype, resources on each node are first assigned to the capsules that explicitly reserved them; the remaining resources are distributed equally among capsules that did not reserve any resources.

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 Fig. 1b). 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 Fig. 1a, 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 Fig. 1b). 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 Fig. 1b, 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 the case of the CPU, an application that needs x units of CPU time every y time units specifies a reservation of (x, y). In the 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 jth 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 (see Fig. 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 (the latter is specified using a do not care option for the mapping). 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. Applications are also allowed to specify lower bounds on the CPU and network bandwidth allocations for their capsules. Finally, applications may specify a parameter ϵ (labeled "Epsilon" in Fig. 2) in the RSL for every capsule; this parameter is used by the resource trading algorithm described in Section 4.3.

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:

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

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

```
Application <id> <New | Modify | Terminate >
AggregateResv <req | *>
TradeResources <yes | no>
Capsules  <numCapsules>
Capsule 1 Resv <req1 | *> Node <nodeId1 | *> LB <lb1 | *> Epsilon <e1 | *>
Capsule 2 Resv <req2 | *> Node <nodeId2 | *> LB <lb2 | *> Epsilon <e2 | *>
...
```

Fig. 2. The Sharc Resource Specification Language. A "$*$" denotes an unspecified value.

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 studied the capsule placement problem in detail in separate pieces of work. In [30], we present an experimental comparison of some online capsule placement strategies. In [31], we describe some theoretical properties of both the online and the offline versions of the capsule placement problem.

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.

Sharc allows resources to be traded among the capsules of an application, but not among the applications themselves. The reason for prohibiting interapplication resource trading is that Sharc has been developed primarily for environments such as commercial hosting platforms where applications negotiate contracts with the cluster seeking guarantees on resource availability. The application providers pay the cluster in return for these resource guarantees. Failure to meet these resource guarantees may imply loss in revenue for the cluster. By not allowing interapplication resource trading, we ensure that when an application needs to utilize all the resources that it had reserved (e.g., when a high number of requests arrive at a news site), it gets them. Systems that allow interapplication resource trading may fail to ensure this. We will see in Section 6 that, although we do not allow interapplication resource trading explicitly, the use of *work conserving* resource schedulers on the nodes in the Sharc cluster ensures that any idle resources are automatically given to applications that need them in addition to their own shares (from this perspective, interapplication trading is "automatic" and implicit in Sharc).

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. Due to space constraints, we only show how the nucleus maps capsule requirements to OS-specific QoS parameters for proportional-share and lottery schedulers. Techniques for reservation-based schedulers and leaky-bucket regulators are described in an extended version of this paper [29].

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). 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 [22]. 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.

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 jth 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 $< U_{i_1 j_1}, U_{i_2 j_2}, \ldots >$ 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.

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

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 underutilizing 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.[3]

Resource trading is only permitted between capsules of the same application, never across applications.

Our recomputation algorithm proceeds in three steps. First, capsules that lent resources to other peer capsules, but need them back, reclaim their allocations. Second, allocations of underutilized 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.

3. Accumulating 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 [3].

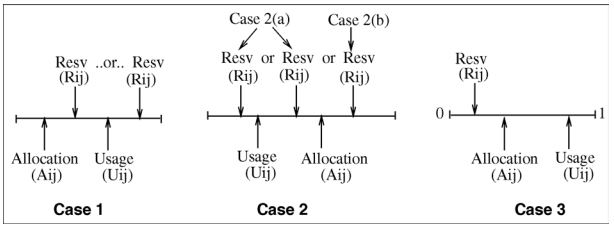


Fig. 3. Various scenarios that occur while trading resources among capsules.

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}). In Fig. 3, Case 1 pictorially depicts this scenario.

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

$$A_{ij} = min(R_{ij}, (1 + \epsilon_{ij}) \cdot U_{ij}), \tag{3}$$

where ϵ_{ij}, $0 < \epsilon_{ij} < 1$ is a per-capsule positive constant that may be specified in the RSL and takes a default value if unspecified. In our experiments, we use a value of 0.1 for this parameter.

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 Fig. 3). The parameter ϵ_{ij} 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 underutilizing resources if its current usage is strictly smaller than its allocation (i.e., usage $U_{ij} <$ allocation A_{ij}). In Fig. 3, Case 2 depicts this scenario.

Since the allocated resources are underutilized, 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 Fig. 3), then the new allocation is set to the usage (i.e., the allocation of a capsule that borrowed bandwidth, but did not use it is reduced to its actual usage). On the other hand, if the current allocation is smaller the reservation (implying that the capsule is lending bandwidth), then any further reductions in the allocations are made gradually (Case 2(b) in Fig. 3). Thus,

$$A_{ij} = \begin{cases} U_{ij} & \text{if } A_{ij} \geq R_{ij} \\ (1 - \epsilon_{ij}) \cdot A_{ij} & \text{if } A_{ij} < R_{ij}, \end{cases} \tag{4}$$

where ϵ_{ij} is a small positive constant, $0 < \epsilon_{ij} < 1$.

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.

It is possible to have two different values for the parameter ϵ_{ij} in Steps 2 and 3. Also, capsules that need to reclaim lent resources fast may be assigned a large ϵ_{ij} value.

In this paper, we report results from experiments in which all the capsules had the same ϵ_{ij} value of 0.1.

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 should not already be lending bandwidth to other capsules ($A_{ij} \geq R_{ij}$), else it would have been considered in Step 1. In Fig. 3, Case 3 depicts this scenario.

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

$$A_{ij} = A_{ij} + min\left(\frac{1 - \sum_{j \in node} A_{ij}}{\mathcal{N}_1}, \frac{R_i - \sum_{j=1}^{C_j} A_{ij}}{\mathcal{N}_2}\right), \quad (5)$$

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 (5) is to *reduce* the total allocation of the capsule; this is permissible since the capsule was already borrowing bandwidth which is returned back.[4] Thus, (5) 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, $\sum_{j \in node} A_{ij} \leq 1$ 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. It can be shown that the bin-packing problem, which is NP-hard [10], reduces to the resource allocation problem that the resource manager solves using Steps 3 and 4. Consequently, the resource manager has to resort to the following heuristic: It performs a small, constant number of additional scans of all capsules to increase or decrease their allocations slightly. This heuristic has been found to perform well in practice, yielding total allocations within 5 percent of the aggregate reservations for applications throughout our experimental study.

The newly computed allocations are then conveyed as an allocation vector $< A_{i_1 j_1}, A_{i_2 j_2}, \ldots >$ 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.

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 recomputation 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 smooths 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

Sharc can handle three types of failures—nucleus failure, control plane failure, and node and link failures. In this section, we provide a brief overview of the mechanisms Sharc employs for recovering from nucleus failures and control plane failures. The details of these mechanisms can be found in the extended version of this paper [29]. 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.

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 $O(k \cdot logk)$ operations for k capsules. Capsules are then linearly ordered in approximately decreasing order

4. For simplicity of exposition, we have omitted a couple of details in (3), (4), and (5). First, these steps also involve ensuring that any lower bounds specified by the application on the capsules' CPU and network bandwidth allocations are maintained. Second, after computing A_{ij} in (5), 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.

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 an $O(n \cdot k)$ operation for a cluster of n nodes. Thus, the overall complexity of admission control and capsule placement is $O(n \cdot k + k \cdot logk)$.

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 [2].

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 $mk >> n$ in shared clusters (β, β' are constants).

Implementation details. We have implemented a prototype of Sharc on a cluster on Linux PCs. 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 multithreaded and is based on Posix threads. The control plane consists of threads for 1) admission control and capsule placement, 2) resource management and trading, 3) communication with the nuclei on various nodes, and 4) 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 [21] and the leaky bucket rate regulator for allocating CPU and network interface bandwidth, respectively. This allows us to demonstrate that Sharc can indeed interoperate 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 microbenchmarks, we demonstrate that Sharc:

1. provides predictable allocation of CPU based on the specified resource requirements,
2. can isolate applications from one another,
3. can scale to clusters with a few hundred nodes running 100,000 capsules, and
4. 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:

1. an e-commerce application consisting of a front-end Web server and a back-end relational database,
2. a replicated Web server that uses Apache version 1.3.9,
3. a file download server that supports download of large audio files, and
4. a home-grown streaming media server that steams 1.5 Mb/s MPEG-1 files.

Our second workload is representative of a research workgroup environment and consists of

1. *Scientific*, a compute-intensive scientific application that involved matrix manipulations,
2. *Summarize*, an information retrieval application,
3. *Disksim*, a publicly-available compute-intensive disk simulator, and
4. *Make*, an application build job that compiles the Linux 2.2.0 kernel using GNU make.

In all the experiments, the best-fit-based placement algorithm described in Section 4.1 was used for placing the applications. A value of 0.1 was used for the parameter ϵ in all the experiments.

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

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 e-commerce 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 e-commerce site). Our file download application emulates a music download site

TABLE 1
Capsule Placement and Reservations

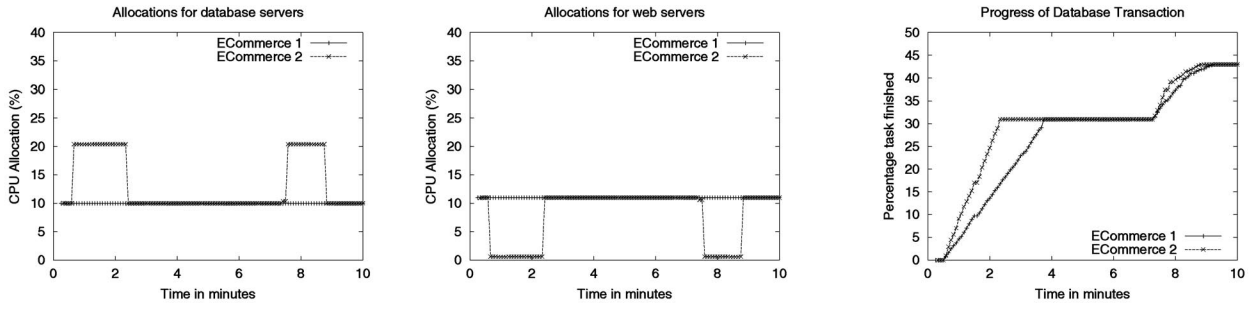
Applications	Capsules & % (cpu,net) Reservations				Workload	Resource trading
	N1	N2	N3	N4		
Ecommerce 1 (EC1)	(10,5)	(10,5)	–	–	Mixed	no
Ecommerce 2 (EC2)	–	(10,5)	(10,5)	–	Mixed	yes
File download (FD)	(5,10)	–	(5,10)	(5,10)	I/O intensive	yes
Streaming (S1)	–	–	(5,5)	(5,5)	I/O intensive	no
Streaming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

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 e-commerce 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 e-commerce 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. Fig. 4 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 $t = 70$ seconds 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 underloaded capsules to the overloaded capsule. We then send a second similar burst at $t = 160$ seconds and observe a similar behavior. We send a third burst at $t = 300$ seconds that is skewed toward 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 $t = 500$ seconds, a similar simultaneous burst is sent to the two capsules again with similar results. Fig. 5 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. Figs. 4 and 5 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



(a) (b) (c)

Fig. 4. Predictable CPU allocation and trading. (a) and (b) show the CPU allocation for the database server and the Web server capsules. (c) shows the progress of the two bursts processed by these database servers.

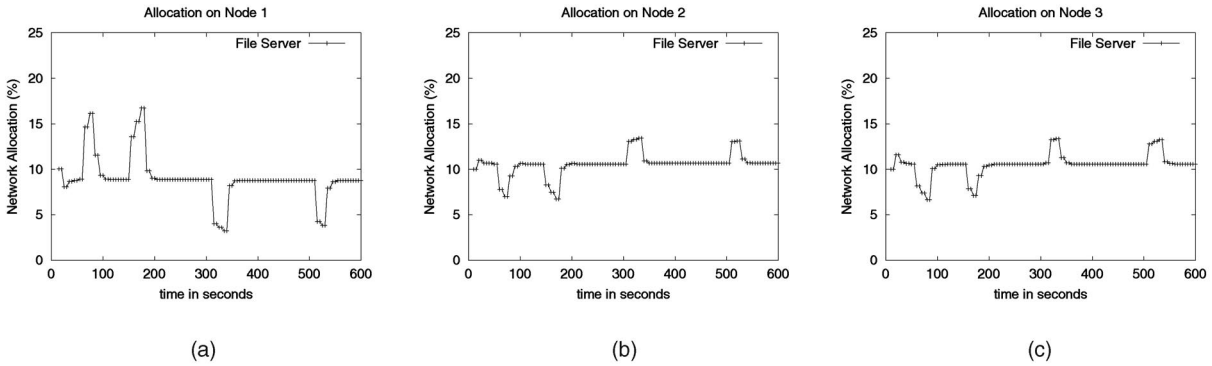


Fig. 5. Predictable network allocation and trading. (a) File download 1. (b) File download 2. (c) File download 3. (a), (b), and (c) depict network allocations of capsules of the File download application.

guaranteed resource share, which is unaffected by the bursty workloads of the e-commerce and file download applications.

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 $t = 25$ minutes until $t = 38$ minutes. 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 $t = 36$ minutes and the bandwidth borrowed on node 3 by the scientific application has to be returned (since the total allocation on the node reaches 100 percent, there is no longer any spare capacity on the node, preventing any further borrowing). Finally, two kernel builds startup at $t = 37$ minutes and are allocated their reserved shares. We measured the CPU allocations and the actual CPU usages of each capsule. Since there are 10 capsules in this experiment, due to space constraints, we only present results for the three capsules on node 3. As shown in Fig. 6, 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 $t = 36$ minutes, the total allocation reaches 100 percent; at this point, there in 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.

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%

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 $t = 5$ minutes, 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). Fig. 7a 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 recomputation period $\mathcal{I}$, which was set to 5 seconds in our experiment). In contrast, lack of resource trading causes a significant degradation in performance. Figs. 7b and 7c show a 150

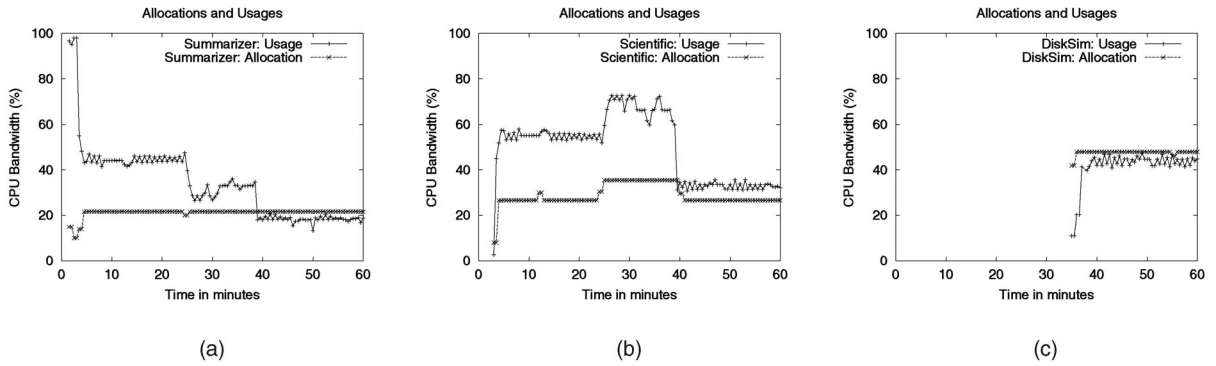


Fig. 6. Predictable allocation and resource trading. (a) Summarizer, (b) scientific, and (c) disksim. (a), (b), and (c) depict DPU usages and allocations of capsules residing on node 3.

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 Fig. 7b, 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.

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 256 MB 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. Fig. 8a 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 1,000 capsules was less than 2 percent when $\mathcal{I} = 5$s. Even with 10,000 capsules, the CPU usage was less than 4 percent when $\mathcal{I} = 20$s and less than 3 percent when $\mathcal{I} = 30$s.

Fig. 8b 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 497μs and 297μs, respectively.

Fig. 8c plots the communication overhead incurred by the nucleus for a 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 1,300 KB for 10,000 capsules (43.3 KB/s), and is around 130 KB per interval (4.3 KB/s) for 1,000 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

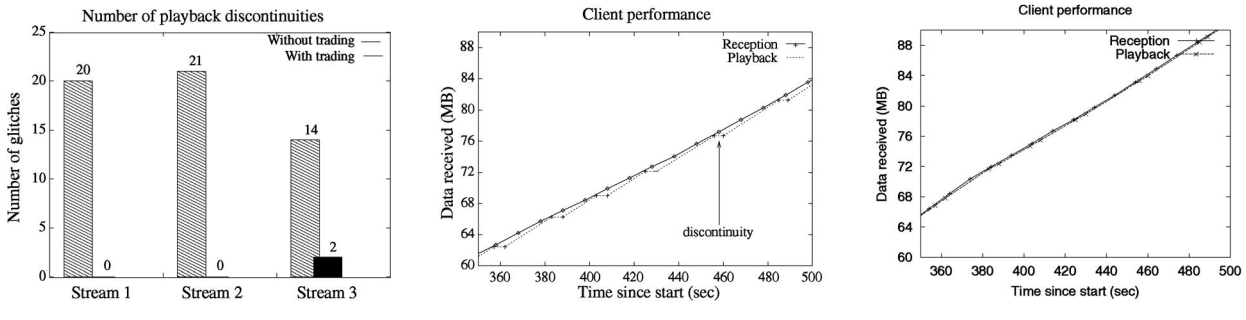


Fig. 7. Impact of resource trading. (a) shows the number of playback discontinuities seen by the three clients of the overloaded video server with and without the trading of network bandwidth. (b) and (c) show a portion of the reception and playback of the second stream for the two cases.

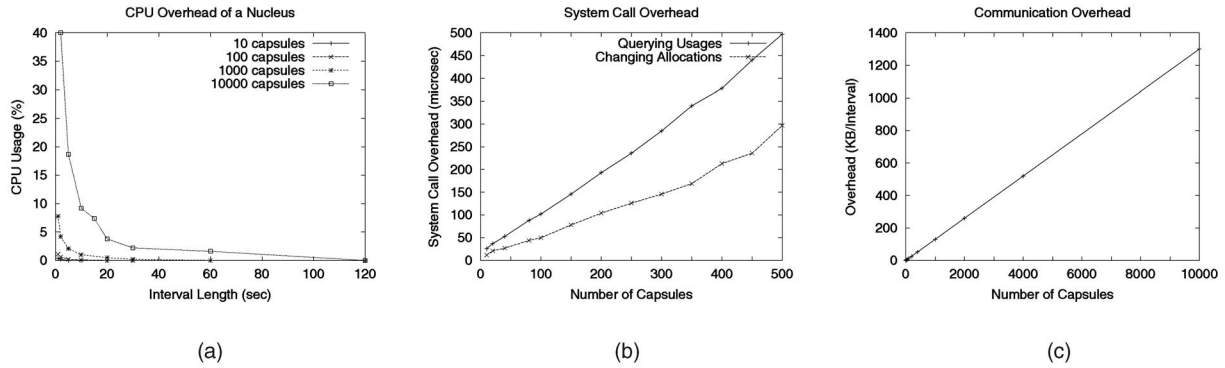


Fig. 8. Overheads imposed by the nucleus. (a) CPU overhead, (b) system call overhead, and (c) communication overhead.

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 did not 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.

Fig. 9a 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 percent when $\mathcal{I} = 30$s). Fig. 9b plots the total busy time for a 256 node cluster. The busy time is defined to 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, Fig. 9c 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 percent 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 Fig. 10a, 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 five minutes, 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 Fig. 10b, 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.

6.7 Handling Failures

We used fault injection to study the effect on failures in Sharc. Due to space constraints, we only present a summary of our experimental evaluation of the three kinds of failure handling that Sharc supports in Table 3. For a detailed description, please refer to an extended version of this paper [29].

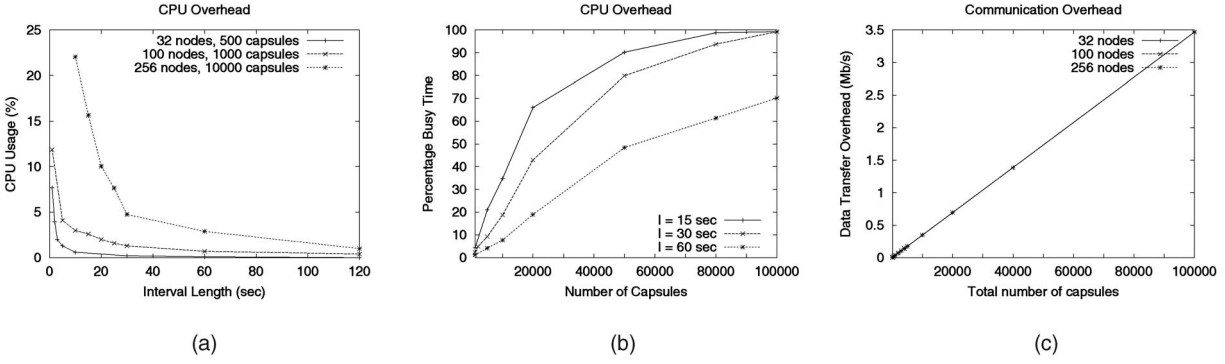


Fig. 9. Overheads imposed by the control plane. (a) CPU overhead, (b) total busy time, and (c) communication overhead.

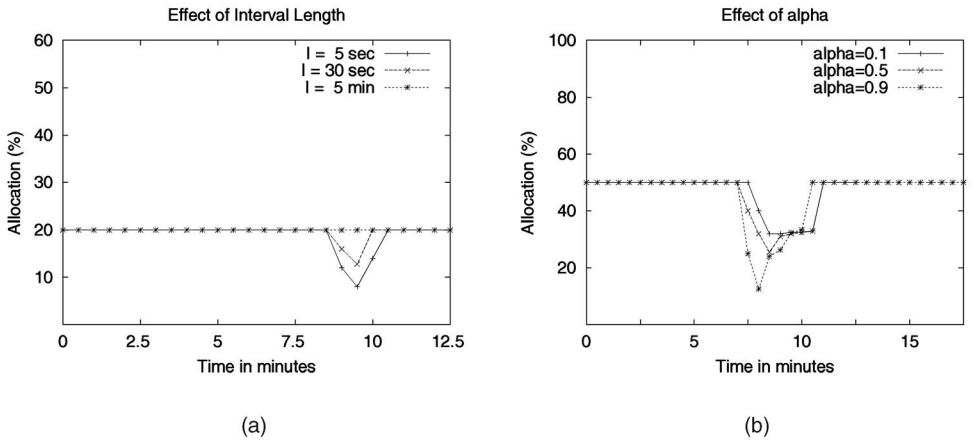


Fig. 10. Impact of tunable parameters on capsule allocations. (a) Effect of $\mathcal{I}$. (b) Effect of α.

7 Related Work

7.1 Resource Management in a Single Machine

Several techniques for predictable allocation of resources within a single machine have been developed over the past decade. New ways of defining resource principals have been proposed that go beyond the traditional approach of equating resource principals with entities like processes and threads. Banga et al. [4] provide a new operating system abstraction called a *resource container*, which enables fine grained allocation of resources and accurate accounting of resource consumption in a single server. *Scheduling domains* in the Nemesis operating system [17], *activities* in Rialto [16], and *Software Performance Units* in [32] are other examples. Numerous approaches have been proposed for predictable scheduling of CPU cycles and network bandwidth on a single machine among competing applications. These include proportional-share schedulers such as *Start-time Fair Queuing* [12], and reservation-based schedulers as in *Rialto* [16] and *Nemesis* [17].

A key contribution of Sharc is to extend the benefits of such single node resource management techniques for CPU and network bandwidth to clustered environments. As discussed in Section 4.2, Sharc can work with a variety of scheduling algorithms. Moreover, Sharc's model of an application as composed of a set of capsules is general enough to allow it to work with any of the resource principal abstractions mentioned above.

There has also been work on predictable allocation of memory, disk bandwidth, and shared services in single servers. Verghese et al. [32] address the problem of managing resources in a shared-memory multiprocessor to provide performance guarantees to high-level logical entities (called *software performance units (SPUs)*) such as a group of processes that comprise a task. Their resource management scheme, called "performance isolation," has been implemented on the Silicon Graphics IRIX operating system for three system resources: CPU, memory, and disk bandwidth. Of particular interest is their mechanism for providing isolation, with respect to physical memory, which works by having dynamically adjustable limits on the number of pages that different SPUs are entitled to based on their usage and importance. They also implement some mechanisms for managing shared kernel resources such as spinlocks and semaphores. Waldspurger [33] introduces several novel mechanisms for managing memory in the VMWare ESX Server. These mechanisms allow the server to support overcommitment of memory to achieve better scalability than simple static partitioning of memory would allow. An algorithm based on a combination of proportional allocation of memory and an idle memory tax is used to provide memory isolation guarantees, while achieving high memory utilization. The *Cello* disk scheduling framework [27] was proposed by Shenoy and Vin to meet the diverse performance requirements of applications supported by modern general purpose file and operating systems. Cello assigns weights to application classes and allocates disk bandwidth to them in proportion to their weights. It derives a schedule that balances the trade off between the desire to align the service provided to applications to their needs and the desire to minimize disk latency overheads. Reumann et al. [24] propose an OS abstraction called *Virtual Service (VS)* to eliminate the performance interference caused by shared services such as DNS, proxy cache services, time services, distributed file systems, and shared databases.

TABLE 3
Failure Handing Times (with 95 Percent Confidence Intervals)

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

We plan to enhance Sharc to also manage disk bandwidth and memory. For disk bandwidth, we intend to use the Cello disk scheduling framework [27] that has now been implemented in the Linux kernel [21]. Our design philosophy for managing disk bandwidth is similar—allow Sharc to determine per-capsule reservations let the Cello scheduler enforce these allocations. For managing memory, we are designing a virtual memory manager based on the idea of dividing memory among capsules so that the miss rates they experience are inversely proportional to their weights. Developing mechanisms to extend this to manage memory on a cluster-wide basis is part of ongoing work.

All these techniques focus on resource allocation for applications running on a single server. Sharc, on the other

hand, is concerned with multitiered applications with components distributed across multiple nodes.

7.2 Resource Management in Shared Clusters

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 [19]. Numerous middleware-based approaches for clustered environments have also been proposed [7], [8]. Finally, gang scheduling and coscheduling efforts have investigated the issue of coordinating the scheduling of tasks in distributed systems [15]; however, this approach does not support resource reservation, which is a particular focus of our work.

Some 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 [3]. 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 the underlying proportional-share scheduler to fair redistribution of *unused* bandwidth (instead of fair allocation of the *total* bandwidth as in [3]). A second difference is that lending resources in [3] 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.

Chase et al. present the design and implementation of *Muse*, an architecture for resource management in a hosting center [6]. Muse uses an economic model for dynamic provisioning of resources to multiple applications. In the model, each application has a utility function which is a function of its throughput and reflects the revenue generated by the application. There is also a penalty that the application charges the system when its goals are not met. The system computes resource allocations by solving an optimization problem that maximizes the overall profit. Muse puts emphasis on energy as a driving resource management issue in server clusters. Like Sharc, Muse uses an exponential smoothing-based predictor of future resource requirement. There are some important differences between Muse and Sharc. Muse allows resources to be traded between applications, whereas Sharc does not. Sharc manages both CPU and network bandwidth. Muse manages only CPU, although we note that its resource management mechanism can be easily extended to manage network bandwidth.

The *Cluster Reserves* work at Rice University has also investigated resource allocation in server clusters [2]. 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*. This is achieved by providing fixed resource shares to application spanning multiple nodes and dynamically adjusting the shares on each server based on the local resource usage. The approach uses resource containers [4] 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).

7.3 Other Resource Management Issues in Shared Clusters

The design of shared clusters involves several problems that are complementary to the one of trading resources presented in this paper. In [30], we present techniques to infer the CPU and network requirements of applications. This information is then used to overbook cluster resources in a controlled fashion such that the cluster can maximize its revenue while providing probabilistic QoS guarantees to applications. We also present several capsule placement strategies and investigate their impact on the cluster's revenue.

7.4 Resource Management in Dedicated Clusters

Alongside research on shared hosting centers, work has appeared on resource management in hosting centers based on a dedicated model in which the applications are assigned resources at the granularity of a single server. Ranjan et al. [22] address the issue of allocating resources in such a model. Their algorithm determines the number of servers to be allocated to each class periodically. It assumes a G/G/N model with FCFS scheduling on each server in the cluster. Response time is assumed to be linearly related to utilization. The algorithm works by trying to maintain a target utilization level. *Oceano* [1] is a management infrastructure developed at IBM for a server farm. This work is mainly concerned with the implementation issues involved in building such a platform and is low on algorithmic details. The *Cluster-On Demand (COD)* [20] work presents an automated framework to manage resources in a shared hosting platform. COD introduces the concept of a *virtual cluster*, which is a functionally isolated group of hosts within a shared hardware base. A key element of COD is a protocol to resize virtual clusters dynamically in cooperation with pluggable middleware components.

7.5 Resource Management in Highly Distributed Clusters

In [34], the authors consider a model of hosting platforms different from that considered in our work. They visualize future applications executing on platforms constructed by clustering multiple, autonomous distributed servers, with resource access governed by agreements between the owners and the users of these servers. They present an architecture for distributed, coordinated enforcement of resource sharing agreements based on an application-independent way to represent resources and agreements.

In this work, we have looked at hosting platforms consisting of servers in one location and connected by a fast network. However, we also believe that distributed hosting platforms will become more popular and resource management in such systems will pose several challenging research problems.

7.6 Scalability and Reliability in Cluster-Based Systems

The design of scalable, fault-tolerant network services running on server clusters has been studied in [9], [14]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in [11]. Scalability, availability, and performance issues in dedicated clusters have been studied in the context of clustered mail servers [26].

8 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

1. supporting resource reservation for applications,
2. providing performance isolation and dynamic resource allocation to application capsules, and
3. 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.

Acknowledgments

The authors would like to thank the reviewers for their insightful comments. This research was supported in part by the US National Science Foundation grants CCR-9984030 and EIA-0080119.

References

[1] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger, "Oceano—SLA Based Management of a Computing Utility," IBM Research, year?

[2] M. Aron, P. Druschel, and W. Zwaenepoel, "Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers," *Proc. ACM SIGMETRICS Conf.,* June 2000.

[3] A. Arpaci-Dusseau and D.E. Culler, "Extending Proportional-Share Scheduling to a Network of Workstations," *Proc. Conf. Parallel and Distributed Processing Techniques and Applications,* June 1997.

[4] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. Third Symp. Operating System Design and Implementation,* pp. 45-58, Feb. 1999.

[5] J. Blanquer, J. Bruno, M. McShea, B. Ozden, A. Silberschatz, and A. Singh, "Resource Management for QoS in Eclipse/BSD," *Proc. Free BSD Conf.,* Oct. 1999.

[6] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers," *Proc. 18th ACM Symp. Operating Systems Principles,* pp. 103-116, Oct. 2001.

[7] Corba Documentation, http://www.omg.org, 2003.

[8] Distributed Computing Environment Documentation, http://www.opengroup.org, 2003.

[9] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 78-91, Dec. 1997.

[10] M.R. Garey and D.S. Johnson, *Computer and Intractability: A Guide to the Theory of NP-Completeness.* year?

[11] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum, "Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors," *Proc. ACM Symp. Operating Systems Principles,* pp. 154-169, Dec. 1999.

[12] P. Goyal, H.M. Vin, and H. Cheng, "Start-Time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *Proc. ACM SIGCOMM,* Aug. 1996.

[13] P. Goyal, S.S. Lam, and H.M. Vin, "Determining End-to-End Delay Bounds In Heterogeneous Networks," *ACM/Springer-Verlag Multimedia Systems J.,* vol. 5, no. 3, pp. 157-163, May 1997.

[14] S.D. Gribble, E.A. Brewer, J.M. Hellerstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction," *Proc. Fourth Symp. Operating System Design and Implementation,* pp. 319-332, Oct. 2000.

[15] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda, "Implementation of Gang Scheduling on a Workstation Cluster," *Proc. Workshop Job Scheduling Strategies for Parallel Processing,* pp. 27-40, 1996.

[16] M.B. Jones, D. Rosu, and M. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 198-211, Dec. 1997.

[17] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE J. Selected Areas in Comm.,* vol. 14, no. 7, pp. 1280-1297, Sept. 1996.

[18] C. Lin, H. Chu, and K. Nahrstedt, "A Soft Real-Time Scheduling Server on the Windows NT," *Proc. Second USENIX Windows NT Symp.,* Aug. 1998.

[19] M. Litzkow, M. Livny, and M. Mutka, "Condor—A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. Distributed Computing Systems,* pp. 104-111, June 1988.

[20] J. Moore, D. Irwin, L. Grit, S. Sprenkle, and J. Chase, "Managing Mixed-Use Clusters with Cluster-on-Demand," Cluster-on-Demand Draft, Internet Systems and Storage Group, Duke Univ., year?

[21] QLinux Software Distribution, http://lass.cs.umass.edu/software/qlinux, 1999.

[22] S. Ranjan, J. Rolia, H. Fu, and E. Knightly, "QoS-Driven Server Migration for Internet Data Centers," *Proc. 10th Int'l Workshop Quality of Service,* 2002.

[23] REACT: IRIX Real-Time Extensions, Silicon Graphics, Inc., http://www.sgi.com/software/react, 1999.

[24] J. Reumann, A. Mehra, K. Shin, and D. Kandlur, "Virtual Services: A New Abstraction for Server Consolidation," *Proc. USENIX Ann. Technical Conf.,* June 2000.

[25] T. Roscoe and B. Lyles, "Distributing Computing without DPEs: Design Considerations for Public Computing Platforms," *Proc. Ninth ACM SIGOPS European Workshop,* Sept. 2000.

[26] Y. Saito, B. Bershad, and H. Levy, "Manageability, Availability and Performance in Porcupine: A Highly Available, Scalable Cluster-Based Mail Service," *Proc. 17th Symp. Operating Systems Principles,* pp. 1-15, Dec. 1999.

[27] P. Shenoy and H. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," *Proc. ACM SIGMETRICS Conf,* pp. 44-55, June 1998.

[28] Solaris Resource Manager 1.0: Controlling System Resources Effectively, Sun Microsystems, Inc., http://www.sun.com/software/white-papers/wp-srm/, 1998.

[29] B. Urgaonkar and P. Shenoy, "Sharc: Managing CPU and Network Bandwidth in Shared Clusters," Technical Report TR01-08, Dept. of Computer Science, Univ. of Mass., Oct. 2001.

[30] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource Overbooking and Application Profiling in Shared Hosting Platforms," *Proc. Fifth Symp. Operating Systems Design and Implementation,* Dec. 2002.
[31] B. Urgaonkar, P. Shenoy, and A. Rosenberg, "Application Placement on a Cluster of Servers," Dept. of Computer Science, Univ. of Mass., year?
[32] B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *Proc. ASPLOS-VIII,* pp. 181-192, Oct. 1998.
[33] C.A. Waldspurger, "Memory Resource Management in VMWare ESX Server," *Proc. Fifth Symp. Operating Systems Design and Implementation,* Dec. 2002.
[34] T. Zhao and V. Karmacheti, "Enforcing Resource Sharing Agreements among Distributed Server Clusters," *Proc. 16th Int'l Parallel and Distributed Processing Symp.,* April 2002.

Bhuvan Urgaonkar received the BTech (Honors) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1999, and the MS degree in computer science from the University of Massachusetts, Amherst, in 2001. He is currently a PhD candidate in the Department of Computer Science at the University of Massachusetts, Amherst. His research interests include distributed systems, operating systems, computer networks, and algorithms.

Prashant Shenoy received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1993, and the MS and PhD degrees in computer science from the University of Texas, Austin, in 1994 and 1998, respectively. He is currently an assistant professor in the Department of Computer Science, University of Massachusetts, Amherst. His research interests are multimedia systems, operating systems, computer networks, and distributed systems. Dr. Shenoy is a member of the Association for Computing Machinery (ACM). He has been the recipient of the US National Science Foundation Career Award, the IBM Faculty Development Award, the Lilly Foundation Teaching Fellowship, and the UT Computer Science Best Dissertation Award. He is a member of the IEEE.