Sharc: Managing Resources in Shared Clusters

Abstract

In this paper, we argue the need for effective resource control 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 control mechanisms such as reservations or shares and extends the benefits of such mechanisms to clustered environments. Our Sharc prototype (i) supports resource reservation for applications, (ii) enables dynamic resource allocation based on resource usages, (iii) provides performance isolation to applications, and (iv) handles a variety of failure scenarios. Our experimental evaluation has shown that Sharc can scale to 256 node clusters running 100,000 applications. Our 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 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 [18]. 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 of a *shared cluster*—a commodity cluster in which 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 [19], replicated web servers with a load balancing switch [2]) 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 control 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 allows the advantages of resource control mechanisms designed for a single node to be extended 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 control 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

[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.

web servers). The resource control 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.

We have implemented a prototype of Sharc on a cluster of Linux PCs and have demonstrated its efficacy using an experimental evaluation. Our results have shown that Sharc can (i) provide predictable allocation of resources, (ii) isolate applications from one another, and (iii) handle a variety of failure scenarios. Our experiments to quantify the overheads imposed by Sharc have shown that these techniques can 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 requirements for managing resources 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 the failure handling techniques employed by Sharc. Section 6 describes our prototype implementation, while Section 7 presents our experimental results. We present directions for future work and related work in Sections 8 and 9, respectively. Finally Section 10 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 borrow terminology from [18] and 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. Each capsule consists of one or more resource principals (processes, threads), all of which belong to the same application. 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 the 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, and ecommerce 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 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 that the throughput of any individual replica server. 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 a different resource need. 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 [3]. 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 an unified technique that integrates both tasks. Further, due to the po-

tential lack of trust among applications in shared clusters, especially in third-party hosting environments, such a technique will 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 prevent untrusted or misbehaving applications from affecting the performance of other applications. This could be achieved, for instance, by employing resource control techniques that also provide performance isolation.

High Availability. A shared cluster can experience a number of failures such as nucleus or capsule failure, node failure, link failure, and application failure. In addition, since resources on a node may be overbooked to extract statistical multiplexing gains, resource exhaustion due to the instantaneous demand exceeding capacity also constitutes a "failure"—the failure to meet resource guarantees. A shared cluster should provide high availability of resources by detecting common types of failures and recovering 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).

Scalability. 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, one of the goals of our work is to develop techniques that scale to clusters of these sizes. We are primarily interested in techniques that scale to moderate size clusters consisting of 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.

Compatibility with Existing OS Interfaces. Whereas the use of a middleware is one approach for managing resources in clustered environments [6, 7], 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 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 [13, 14] or shares [8, 11]. Many commercial and open-source operating systems such as Solaris [21], IRIX [20] and FreeBSD [5] already support such features.

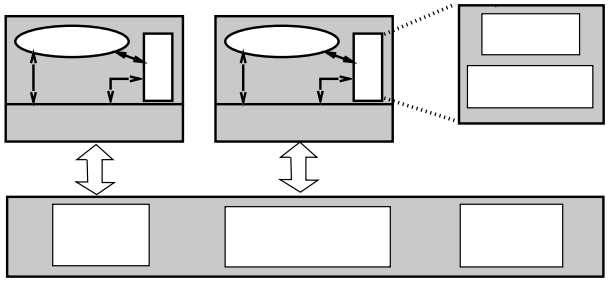


Figure 1: The Sharc Architecture.

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. 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 on the behalf of applications and determine appropriate reservations for applications and their capsules; the task of *enforcing* these reservations is left to the operating system kernel. This provides a clean separation of functionality between resource reservation and resource scheduling.

In this paper, we focus primarily on one resource, namely the CPU, and demonstrate how to allocate CPU bandwidth to applications in a shared cluster. Techniques proposed in this paper are easily extended to other cluster resources such as disk and network interface bandwidth.

3.1 The Control Plane

As shown in Figure 1, 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 node failures or

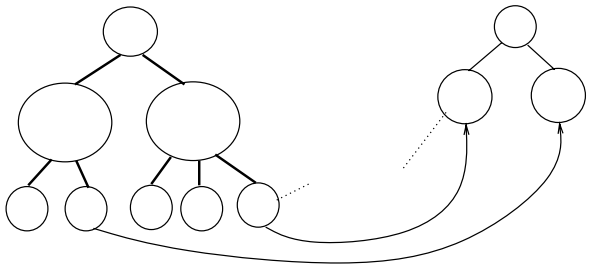


Figure 2: Sharc abstractions: A sample cluster-wide virtual hierarchy, a physical hierarchy on a node and the relationship between the two.

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 2(a)). 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 the 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 resource 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, 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. It also 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. The fault tolerance module is responsible for detecting and recovering from control plane failures and is described in Section 5.2

The nucleus uses the abstraction of a *physical* hierarchy to achieve these goals (see Figure 2(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 2, 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.

As explained earlier, the nucleus on a node is distinct from the operating system kernel. The nucleus, in conjunction with the control plane, determines the reservations for capsules and conveys them to the CPU scheduler; the CPU scheduler is entrusted with the task of actually enforcing these reservations. Sharc does not depend on a particular CPU scheduling algorithm; any scheduler suffices so long as it supports CPU reservations or CPU shares. Depending on the scheduler, the nucleus needs to map capsule requirements into the QoS parameters supported by the CPU scheduler. This mapping is trivial in case of reservation-based schedulers [13, 14]—capsule reservation translate directly to corresponding CPU reservations. The mapping becomes non-trivial if each capsule consists of multiple resource principals. In such a scenario, this mapping is simplified if the scheduler supports hierarchical reservations or aggregate reservations for groups of resource principals (e.g., resource containers [4]). In the event that the scheduler supports reservations only on a per-resource principal-basis, then the nucleus will need to partition the reservation of a capsule among its constituents and reserve CPU bandwidth for each individual resource principal.[2]

Sharc also supports CPU schedulers that employ shares instead of reservations [8, 11]. A proportional-share scheduler provides relative allocation of resources—the allocation of each resource principal is proportional to its weight and the fraction of the CPU bandwidth decreases as the number of tasks increases. Although a proportional-share scheduler by itself only provides relative guarantees, it has been shown that such schedulers can provide strong (deterministic) guarantees when combined with admission control—the admission controller limits the number of tasks in the system, enabling the scheduler to provide bounds on the allocations. Since the control plane in Sharc employs admission control, it is feasible to employ a proportional-share scheduler on each node and provide the desired resource guarantees. The mapping of capsule reservations to CPU shares is discussed in Section 4.1.

Next we describe the resource management mechanisms and policies employed by the control-plane and the nucleus

[2]To do so, the nucleus will need to be made aware of the resource principals within each capsule. In general, the nucleus only deals with capsules as a unit of allocation and is unaware of resource principals within a capsule or how the capsule further partitions its allocation among its constituents.

4 Sharc Mechanisms and Policies

In this section, we describe resource specification, admission control, and capsule placement policies employed by Sharc. We then 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.

The resource requirements of an application are specified using the notion of a *reservation*. Formally, a reservation is a pair (x, y) that requires x units of CPU time to be allocated to the capsule every y time units. To reduce the complexity of the allocation process, Sharc assumes that all reservations are made based on a fixed interval of duration τ. That is, $y = \tau$ and is fixed for all applications in the cluster. Consequently, a reservation in Sharc is a singleton R, $0 < R \leq 1$ that specifies the fraction of the interval τ reserved for a capsule (i.e., specifies that $R \cdot \tau$ units of CPU time be allocated to the capsule every τ time units). Let R_{ij} denote the reservation of the j^{th} capsule of application i, and let R_i denote the aggregate reservation of that application. Assuming that application i has C_i capsules, we have $R_i = \sum_{j=1}^{C_i} R_{ij}$. Observe also that while R_{ij} is a fraction between 0 and 1, the aggregate reservation R_i is, by definition, a quantity between 0 and C_i (since $0 < R_{ij} \leq 1 \Rightarrow 0 < \sum_{j=1}^{C_i} R_{ij} \leq C_i$).

Applications specify their resource requirements to Sharc using a simple resource specification language (see Figure 3). 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 specification 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 dont-care option for the mapping). An application is also allowed to specify if resource trading is permitted for its capsules. The allocation of capsules remains fixed if resource trading is prohibited; the allocation is adjusted based on usages when resource trading is permitted. Resource trading allows unutilized CPU cycles to be temporarily lent to other peer capsules under the condition that they are returned when needed.

Given such a resource specification, the admission control algorithm first performs checks to ensure that the request is feasible. A resource specification is said to be feasible if (1) each capsule requests no more than the total resources on a node; assuming a unit amount of resource on a node, we have $\forall j, \quad R_{ij} \leq 1$; and (2) the aggregate reservation does not exceed C_i: $R_i \leq C_i$. Whereas the former feasibility check is used when reservations for individual capsules are specified, the latter check is employed when the application specifies only an aggregate reservation. In the latter case, the reservation of each individual capsule is initialized to $R_{ij} = \frac{R_i}{C_i}$.

```
Application <id> <New | Modify | Terminate >
AggregateResv <req | *>
TradeResources <yes | no>
Capsules  <numCapsules>
Capsule 1 Resv <req | *> Node <nodeId | *>
Capsule 2 Resv <req | *> Node <nodeId | *>
...
```

Figure 3: The Sharc Resource Specification Language. A "$*$" denotes an unspecified value.

Assuming a feasible resource specification, the admission control algorithm proceeds as follows. First, the control plane needs to ensure that the total reservation for all applications does not exceed the capacity of the cluster. Assuming n nodes and m existing application, we have

$$\sum_{i=1}^{m} R_i + R_{m+1} \leq n \tag{1}$$

Next, the admission control algorithms determines if sufficient spare capacity exists on each individual node. If the resource specification specifies the desired mapping of capsules onto nodes, then the control plane only needs to check that the unused capacity on those nodes is larger than than the capsule reservations. That is,

$$\forall j, \;\; R_{ij} \leq S^k \tag{2}$$

where capsule j is mapped onto node k and S^k denotes the spare capacity on node k. If capsule placement is unspecified, then the capsule placement algorithm needs to determine a feasible mapping. To do so, the capsule placement algorithm sorts capsules in descending order of reservations and uses a best-fit strategy to map capsules to nodes in sorted order. For each capsule j, it finds a node using best-fit such that the spare capacity on that nodes exceeds the reservation R_{ij}. Sorting the capsules in descending order of reservations and using best-fit ensures that the algorithm will always find a mapping if one exists (otherwise a capsule with a smaller requirement might get mapped onto a node with a larger spare capacity, preventing a capsule with a larger resource requirement from finding such a node). Our current version of the control plane does not incorporate statistical overbooking of resources or take trust among applications into account while determining capsule placement—techniques for doing so are the subject of future research and are discussed briefly in Section 8.

If admission control is successful, the control plane creates a new application node in the virtual hierarchy and notifies all affected nuclei, which then update their physical hierarchies. Each nucleus in turn sets the appropriate reservation

or share for the capsule in the CPU scheduler. If the scheduler is reservation-based, then the reservation of the capsule is set to $(R_{ij} \cdot \tau, \tau)$.[3] If the scheduler supports CPU shares, then the reservation is mapped onto a weight by setting $w_{ij} = R_{ij}$; doing so guarantees at least $R_{ij} \cdot \tau$ time units to each capsule (a capsule may receive more than its allocation if there is unused bandwidth on a node, since unused bandwidth is fairly redistributed by a proportional-share scheduler).

4.2 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. Like the reservation R_{ij}, both A_{ij} and U_{ij} denote the fraction of CPU time allocated and used over an interval of duration τ; $0 \leq U_{ij} \leq 1$ and $0 < A_{ij} \leq 1$.

The nucleus on each node tracks the CPU 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 based on its usage, subject to aggregate reservation constraints for that application. The resource manager in the control plane uses usage vectors reported by various nuclei to compute new allocations every $\mathcal{I}$ time units. Before doing so, it first uses an exponential smoothing function to smooth the reported usage for each capsule.

$$U_{ij} = \alpha \cdot U_{ij}^{new} + (1 - \alpha) \cdot U_{ij} \tag{3}$$

where 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. Since nuclei on nodes are not synchronized, a nucleus might fail to report its usage vector within the allocated interval $\mathcal{I}$ (due to clock drift, failure or overload problems, all of which delay updates from the node). In the absence of usage information from a node, the resource manager conservatively assumes that the usages for capsules on that node equal their reservations (i.e., U_{ij}^{new} is set to R_{ij} in Eq. 3). As explained in Section 5, 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 CPU bandwidth 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 CPU cycles. (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.[4] Resource trading is only permitted between capsules of the same application, never across applications.

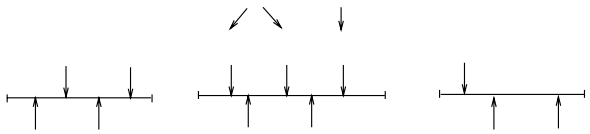


Figure 4: Various scenarios that occur while trading resources among capsules.

Our recomputation algorithm proceeds in three steps. First, capsules that lent resources to other peer capsules but need it back reclaim their allocations. Second, allocations of underutilized capsules are reduced appropriately. Third, this unutilized bandwidth is distributed (lent) to any capsules that could benefit from additional resources. Thus, the algorithm proceeds as follows, one application at a time.

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 CPU bandwidth*. 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 4, 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 4). The parameter ϵ ensures that the new allocation is slightly larger than the current usage, enabling the capsule to (gradually) reclaim lent resources.

[3] If there are multiple resource principals, say r, within a capsule and the scheduler does not support reservations for groups of resource principals, then the reservation of each resource principal is set to $(R_{ij} \cdot \tau / r, \tau)$.

[4] Accumulating credit for unused resources can cause starvation. For example, a capsule that could sleep for an extended duration of time and use its accumulated credit to continuously run on the CPU, thereby starving other applications. CPU schedulers that allow accumulation of credit need to employ techniques to explicitly avoid this problem [3].

Step 2: *Underutilized capsules give up CPU bandwidth*. 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 4, Case 2 depicts this scenario.

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 then the reservation (Case 2(a) in Figure 4), 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 the reservation (implying that the capsule is lending bandwidth), then any further reductions in the allocations are made gradually (case 2(b) in Figure 4). Thus,

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

where ϵ is a small positive constant.

After examining all capsules of the application in Steps 1 and 2, the resource manager can then allocate any unused bandwidth to the remaining capsules of that application.

Step 3: *Needy capsules are lent additional (unused) bandwidth*. A capsule signal 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 4, 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,

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

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 Equation 5 is to *reduce* the total allocation of the capsule; this is permissible since the capsule was already borrowing bandwidth which is returned back.[5] Thus, Equation 5 accounts for both positive and negative spare bandwidth in one unified step.

[5]For simplicity of exposition, we omitted one detail in Eq. 5. After computing A_{ij} in Eq. 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.

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. This requires one additional scan of all capsules so as to increase or decrease their allocations slightly using a simple heuristic (details omitted due to space constraints).

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 7, 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 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.

5 Failure Handling in Sharc

In this section, we describe the failure recovery techniques employed by Sharc. We consider 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—the virtual and the physical hierarchies replicate state information maintained by Sharc (see Figure 2); this replication is intentional and enables reconstruction of state lost due to a failure.

5.1 Nucleus Failure

A nucleus failure occurs when the nucleus on a node fails but the node itself remains operational. It is the responsibility of the control plane to detect a nucleus failure. If a nucleus fails to report usage statistics for two consecutive intervals of duration $\mathcal{I}$, then the fault tolerance module on the control plane is invoked to diagnose the problem. The fault-tolerance module first checks if the node is alive by sending echo messages to the node and then executing a remote script that examines the health of various operating system services. If the node is found to be healthy, the module then attempts to contact the nucleus. If the nucleus fails to respond, a nucleus failure is flagged.

The fault tolerance module then attempts to recover from the failure by starting a new nucleus (using a remote script that first cleans up the remnants of the previous nucleus and then starts

up a new one). The control plane then synchronizes its state with the nucleus by (i) examining the virtual hierarchy to determine all capsules residing on that node, and (ii) reconstructing the physical hierarchy using this information. Since the kernel is unaffected by the nucleus failure, the QoS parameters maintained by the CPU scheduler for individual capsules are also unaffected. Note that the control plane disables resource trading for capsules on that node until failure recovery is complete; this is done by setting $A_{ij} = U_{ij} = R_{ij}$ for all resident capsules in the absence of usage reports from the node.

5.2 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 the event of a control plane failure, all nuclei run a leader election algorithm [22] to elect a new node to host the control plane. This is achieved as follows. Upon detecting an unreachable control plane, the fault tolerance module on the nucleus invites all other nuclei, using a broadcast message, to participate in a voting process to elect a new control plane. Using a variant of the election algorithm described in [22], the nuclei then elect the node with the largest ID that has sufficient resources to run the control plane (the amount of resources required to run the control place is known a priori, since this is configured statically at system startup time based on the number of nodes and applications in the cluster). If the election fails due to the lack of sufficient resources on nodes to run the control plane, then the need for human intervention is signalled. If the election succeeds, then the nucleus on the elected node starts up a new control plane with the appropriate reservation. The control plane then tries to recover the state of the virtual hierarchy—this is achieved by polling each nucleus for the physical hierarchy and creating a union of the physical hierarchies.

Under rare circumstances, the cluster might have two concurrent control planes running. This happens if the node running the control plane experiences a transient link failure but the node itself remains operational during the failure. Before the restoration of the link, the other nuclei could vote and start up a new control plane. Each control plane broadcasts a periodic heartbeat message and listens for similar messages from other control planes. If a second control plane is detected then a simple election algorithm is run to choose between the two—typically a younger control plane (i.e., one that was started later) is always given preference and the older control plane terminates itself.

5.3 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 examine the virtual hierarchy and automatically reassign any capsule running on that node to other nodes in the cluster. The reassignment process involves admission control and capsule placement for the affected capsules. After determining the new mappings, the corresponding nuclei are notified and their physical hierarchies are updated. The affected application capsules can then be restarted on that node. Note that this process only helps determine a new set of nodes to run the capsules residing on the failed node; it does not help in recovering the state of the failed capsules—recovery of lost capsule state, if desirable, is left to the application and requires application-specific mechanisms such as checkpointing or logging.

6 Implementation Considerations and Complexity

We have implemented a prototype of Sharc on a cluster on Linux PCs. The entire Sharc system consists of around 4500 lines of code and is publicly available.[6] We chose Linux as the underlying operating system since a number of reservation-based and share-based CPU scheduler implementations are available for Linux, allowing us to experiment with how capsule reservations in Sharc map onto different QoS parameters supported by these schedulers. For the purposes of this paper, we chose a proportional-share CPU scheduler, since this allows us to demonstrate that Sharc can indeed interoperate with a non-reservation-based scheduler. We chose a weighted fair share-based scheduler, namely SFQ, available publicly from [17]. The CPU scheduler allows a weight to be associated with each resource principal and allocates processor bandwidth in proportion to their weights [11]. The scheduler also supports hierarchical allocation, allowing us to group resource principals into single capsule and allocate a share (weight) to the entire group.

Our Sharc prototype consists of two components—the control plane and the nucleus—that run as privileged processes in user space. 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. The nucleus can use system calls supported by the vanilla Linux kernel and the CPU scheduler to query CPU usage and set or modify shares

[6] The URL for download has been withheld for purposes of blind reviewing. Interested reviewers may contact us with the permission of the program chair.

for capsules. However, since these system calls operate on a single capsule at a time, to improve efficiency, we have implemented two new system calls that allow the nucleus to query CPU usages or set shares for multiple capsules simultaneously. This was the only change to the operating system interface; no other changes were made in line with our goal of backward compatibility with operating system interfaces and libraries.

The CPU and the nuclei communicate with one another using sockets with well-known port numbers. We chose raw sockets over remote procedure calls for reasons of efficiency. Each nucleus is required to register with the control plane at system startup time to enable this communication.

Next we describe the complexity of the mechanisms employed by the control plane and the nucleus.

Admission Control and Capsule Placement. For each new application, the control plane first sorts capsules in order of their resource requirements, which is an $O(c \cdot logc)$ operation for c capsules. The control plane is assumed to maintain a list of nodes sorted on their spare capacities (if not, this takes $O(n \cdot logn)$ time for n nodes). The control plane then uses a best-fit technique to match capsules with nodes. Since both capsules and nodes are in sorted order, this can be achieved in a single linear scan of the capsules and nodes (an $O(n+c)$ operation). Thus, the overall complexity of admission control and capsule placement is $O(n \cdot logn + c \cdot logc)$.

Resource trading. The resource trading algorithm described in Section 4.2 proceeds one application at a time; capsules of an application need to be scanned no more than twice to determine their new allocations (once for the first three steps and once in Step 4). Thus, the overall complexity is linear in the number of capsules and takes $O(mc)$ time in a system with m applications, each with c capsules (total of mc capsules). Each nucleus on a node participates in this process by determining CPU usages of capsules and setting new allocations; the overhead of these tasks is two system calls every $\mathcal{I}$ time units.

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 $\beta \cdot n + \beta' \cdot mc$, it reduces to $O(mc)$ bytes in practice, since $mc >> n$ in shared clusters (β, β' are constants).

7 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.

7.1 Experimental Setup

The testbed for our experiments consists of a cluster of five Linux-based PCs. Each PC is a 350 MHz Pentium II with 64MB RAM and runs Redhat Linux 5.2 with the 2.2.0 version of the Linux kernel. We used the publicly available implementation of the SFQ CPU scheduler from [17]; the process involved downloading a kernel patch and recompiling the patched kernel. All PCs are equipped with a 100Mb/s 3Com ethernet card (model 3c595) and are interconnected by a 3Com SuperStack II ethernet switch. Our experiments assumed a lightly loaded network, and unless specified otherwise, the Sharc control plane was 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) the Apache web server version 1.3.9, (ii) a home-grown streaming media server that streams variable bit-rate MPEG-1 files, (ii) the Quake game server (based on the publicly-available LinuxQuake version 2.30), and (iv) the mySQL database server version 3.23. 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.

7.2 Predictable CPU Allocation and Resource Trading

We conducted an experiment to demonstrate resource trading among capsules and predictable allocation of CPU bandwidth. We chose a research workgroup environment with the four applications listed in Section 7.1. The placement of various capsules and their reservations are listed in Table 1. 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 $t = 25$min, until $t = 38$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 $t = 36$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 $t = 37$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 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

Table 1: 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%

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 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.

7.3 Application Isolation in Sharc

We demonstrate application isolation in Sharc using a workload representative of a shared hosting platform. We use the following setup: (i) Node 1: mySQL server (50% reservation), (ii) Node 2: Quake server (15% reservation), and (iii) Node 3: streaming media server (15% reservation). We used the benchmark suite distributed with the mySQL server to emulate a heavy database workload. The Quake and streaming media servers are lightly loaded at all times. We ran a replicated web server (server A) with capsules on nodes 1 and 2; the aggregate reservation was set to 80% (40% per capsule) and resource trading was permitted. A second replicated web server (server B) was run on nodes 2 and 3 with a reservation of 20% per capsule; resource trading was turned off for this application. The following experiment demonstrates that Sharc can effective isolate applications from one another in the presence of bursty web workloads. We used the `httperf` tool [16] to send a burst of web requests to server A on node 1. The burst consists of requests for static web pages as well as dynamically generated web pages (Apache's PHP3 scripting language is used for dynamic web page generation). The burst causes the capsule on node 1 to borrow bandwidth from its peer on node 2, but does not affect the database server (see Figure 6(a)). Next we send a simultaneous burst to both capsules of server A; this causes the bandwidth borrowed on node 1 to be returned to node 2, but other applications are unaffected (see Figures 6(a) and (b)). Finally, we send a burst of web requests to the capsule of server B on node 3 (while maintaining a bursty workload on server A). Since resource trading is prohibited for server B, the capsule is unable to borrow bandwidth from its peer, even though the latter has bandwidth to spare. Again, the bursts do not affect other applications on the cluster (see Figure 6). This demonstrates that Sharc can effectively isolate applications from one another.

7.4 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.

7.4.1 Overheads Imposed by the Nucleus

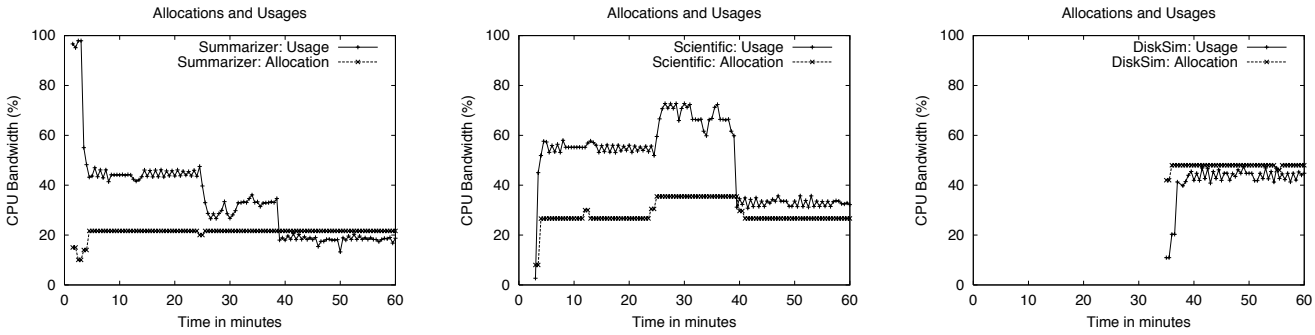
We first measured the CPU usage of the nucleus for varying loads; the usages were computed using the `times` system call and profiling tools such as `gprof`. We varied the number of capsules on a node from 10 to 10,000 and measured the CPU usage of the nucleus for different interval lengths. Figure 7(a) plots our results. As shown, the CPU overheads decrease with increasing interval lengths. This is because the nucleus needs to the query the kernel for CPU 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 $\mathcal{I} = 30$s. Even with 10,000 capsules, the CPU usage was 29% when $\mathcal{I} = 20$s and less than 10% when $\mathcal{I} = 30$s.

Figure 7(b) plots the system call overhead incurred by the nucleus for querying CPU 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 191μs and 102μs, respectively

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 640KB for 10,000 capsules (21.3 KB/s) and is less than 64KB per interval (2.1 KB/s) for 1000 capsules. Together these results show that the overheads imposed by the nucleus for most realistic workloads is small in practice.

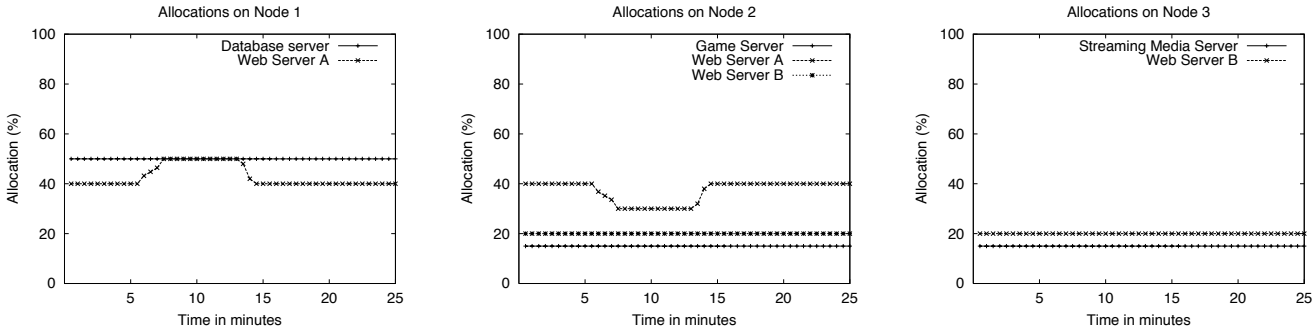
7.4.2 Control Plane Overheads

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 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.



(a) Summarizer (b) Scientific (c) Disksim

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



(a) Node 1 (b) Node 2 (c) Node 3

Figure 6: Application Isolation in Sharc. The allocations of all capsules on the three nodes are shown (due to space constraints, CPU usages of these capsules have been omitted).

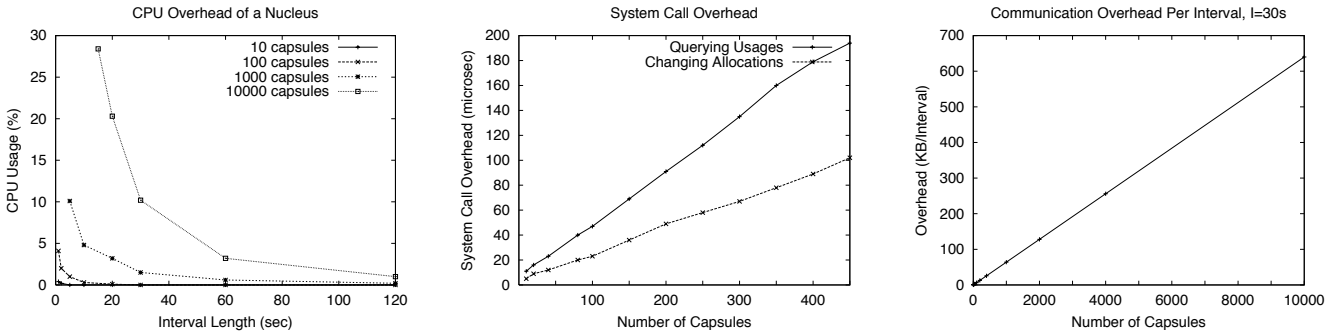
Figure 8(a) plots the CPU usage of the control plane for varying cluster sizes and interval lengths. The figure shows that a control plane running on a dedicated node can easily handle the load imposed by a 256 node cluster with 10,000 capsules (the CPU usage was less than 15% when $\mathcal{I} = 30$s). Figure 8(b) 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, 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 1.71Mb/s, which is less than 2% 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.

Table 2: Failure Handing 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

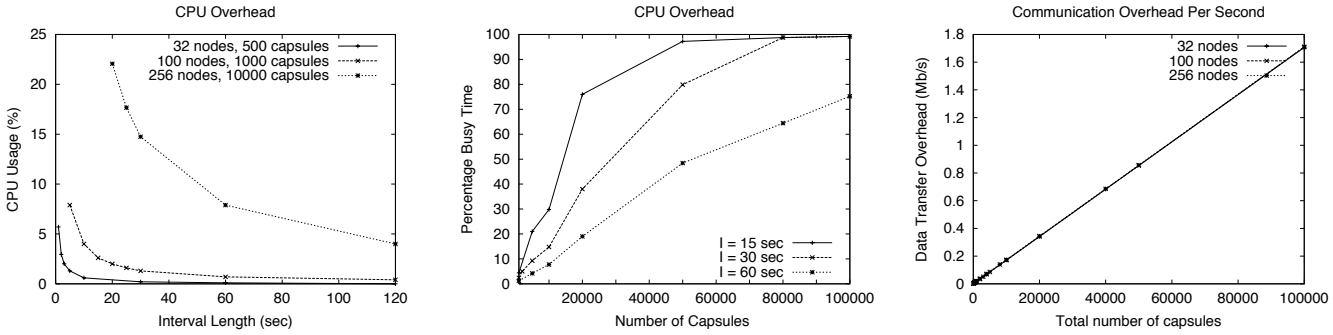
7.5 Handling Failures

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 2, 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



(a) CPU overhead (b) System call overhead (c) Communication Overhead

Figure 7: Overheads imposed by the nucleus.



(a) CPU overhead (b) Total Busy Time (c) Communication Overhead

Figure 8: Overheads imposed by the control plane.

(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 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 handles 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.

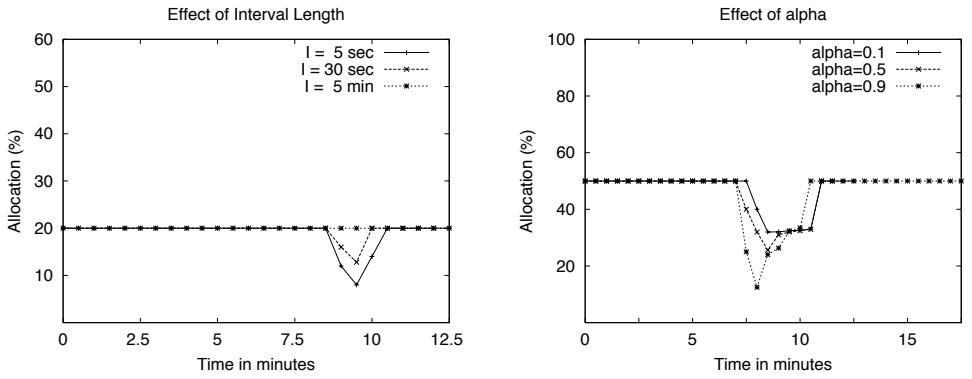
7.6 Effect of Tunable Parameters

To demonstrate the effect of tunable parameters $\mathcal{I}$ and α, we used the same set of workgroup applications described in Table 1. 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.

8 Limitations and 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 processor speeds or different number of processors. Our approach in-



(a) Effect of $\mathcal{I}$ (b) Effect of α

Figure 9: Impact of tunable parameters on capsule allocations.

volves modeling the slowest node in the system as a unit resources and modeling resources on other nodes relative to this node. The admission control, capsule placement, and resource trading techniques need to be modified accordingly. To illustrate, the current uniprocessor admission control that requires $R_{ij} \leq 1$ for each capsule generalizes to $R_{ij} \leq min(p, r)$ where p denotes the number of processors on that node and r denotes the number of resource principals in that capsule.

Managing other cluster resources: Although Sharc currently supports only cluster-wide allocation of CPU bandwidth, our techniques are applicable to other cluster resources such as disk and network interface bandwidth. We plan to enhance Sharc to manage all of these resources. Our design philosophy for these enhancements is similar—allow Sharc to determine per-capsule reservations for each such resource and let the OS scheduler enforce these allocations.

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 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.

9 Related Work

Several techniques for predictable allocation of resources within a single machine have been developed over the past decade [4, 8, 13, 14, 23]. 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 [15]. The design of scalable, fault-tolerant network services running on server clusters has been studied in [9, 12]. Use of virtual clusters to manage resources and contain faults in large multiprocessor systems has been studied in [10]. Scalability, availability and performance issues in dedicated clusters have been studied in the context of clustered mail servers [19] and replicated web servers [2]. Numerous middleware-based approaches for clustered environments have also been proposed [6, 7].

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 [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 that 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 accu-

mulation 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 [1]. 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 [4] and employs a linear programming formulation for allocating resources, resulting in polynomial time complexity. Sharc employs a simpler linear time algorithm to trade resources among capsules. Although our approach is more scalable due to its reduced complexity, Cluster Reserves can yield better allocation, especially in Step 4 of our trading algorithm where we revert to a heuristic. Finally, unlike Sharc, neither of these efforts have considered failure handling techniques.

10 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Corba Documentation. Available from http://www.omg.org.

[7] Distributed Computing Environment Documentation. Available from http://www.opengroup.org.

[8] K. Duda and D. Cheriton. Borrowed Virtual Time (BVT) Scheduling: Supporting Lantency-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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] D. Mosberger and T. Jin. httperf - A Tool for Measuring Web Server Performance. Technical Report HPL-98-61, HP Labs, 1998.

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

[18] 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.

[19] 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.

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

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

[22] A S. Tannenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[23] 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.