Lessons from Resource Allocators for Large-Scale Multiuser Testbeds

Robert Ricci[†] ricci@cs.utah.edu
David Oppenheimer[‡] doppenhe@cs.ucsd.edu
Jay Lepreau[†] lepreau@cs.utah.edu
Amin Vahdat[‡] vahdat@cs.ucsd.edu

[†]*University of Utah, School of Computing*
[‡]*University of California San Diego, Department of Computer Science and Engineering*

Abstract

Resource allocation is a key aspect of shared testbed infrastructures such as PlanetLab and Emulab. Despite their many differences, both types of testbed have many resource allocation issues in common. In this paper we explore issues related to designing a general resource allocation interface that is sufficient for a wide variety of testbeds, current and future. Our explorations are informed by our experience developing and running Emulab's "assign" resource allocator and the "SWORD" resource discoverer, our experience with the PlanetLab and Emulab testbeds, and our projection of future testbed needs.

1 Introduction

PlanetLab's [4] principles include "unbundled management," [18] where key functionality for creating, configuring, and managing experiments and services is provided not by PlanetLab itself, but by *Infrastructure Services*. Each Infrastructure Service has responsibility for a different part of the application life cycle, and the goal is to design the interface for each type of Infrastructure Service to allow a variety of designs and implementations. Different services can then be built with different features and goals, and these can coexist and compete in an open environment.

The Infrastructure Service we concentrate on is the Resource Allocator. As shown in Figure 1, this service takes as input an abstract description of desired resources from a user and a description of resource status from a Resource Discoverer. It produces an *allocation* of concrete resources to be instantiated by a Service Deployment service.

In this proposed model, a Resource Discovery service collects and maintains information about the state of nodes (i.e., location, CPU and memory usage) and the paths between them (such as latency, bandwidth, and packet loss.) Resource Allocation services will use this information to control access to and scheduling of those resources.

Currently, the dominant way of choosing nodes on PlanetLab is ad-hoc. Users choose from all available PlanetLab nodes a subset to host their application. While resource monitoring services [15] are available to guide this decision process, many users assign nodes to their applications in an arbitrary and static manner. Slices created this way have lease times of eight weeks, but studies have shown [17] that conditions on PlanetLab are very likely to change over time periods as small as half an hour, leaving users with mappings that no longer meet their requirements. Because picking the right nodes is a major factor in the success of an experiment, a better way to select nodes is needed.

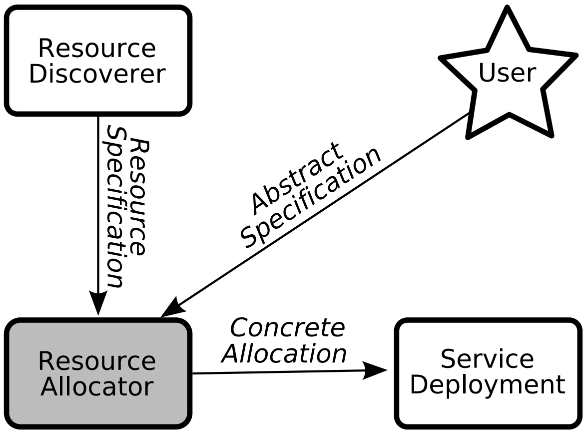

Figure 1: The Resource Allocator's relationship to other Infrastructure Services

This better way is a Resource Allocator. Its goal is to let users specify characteristics of their slice in higher-level terms and find resources that match their requirements. For example, a user might request three nodes that are within 15 milliseconds network latency of one another, but are at different sites. In response to this request, the allocator may give the user nodes at MIT, Harvard, and Boston University. Or, the user may ask for a set of "server" nodes that have high bandwidth between them, and a set of "client" nodes on slower links. The allocator will have to take into account the current status of candidate nodes, such as the availability of CPU time, memory, and disk space, as well as whether or not the user is allowed access to the node.

While conceptually simple, there are many imaginable types of resource allocators. Some might attempt to provide users hard guarantees about their allocations, while others might give best-effort service. Some may implement "first come, first serve" scheduling, while others may support preemption and reservations. Some may support priorities among users, while others might use complex models such as an economic marketplace where users pay for resources with some abstract credits or real currency [3]. Some may allocate resources immediately or not at all, while others may support some sort of scheduling. Allocators with access to historical data about path characteristics may choose to schedule an experiment to run at a time of day or day of the week at which it predicts the desired conditions will be present. Through all of this, the Resource Allocator should attempt to make *efficient* use of the resources it controls, to maximize the utility delivered to end users and applications.

Obviously, the Resource Allocator encapsulates not just mechanism, like some Infrastructure Services, but potentially complex policies. These policies are visible and crucial to the individual user and to PlanetLab as a whole. These are among the reasons that designing a general Resource Allocator interface is especially challenging.

In this paper, we draw on our combined experience with two very different resource allocators and discoverers, both of which have been used on PlanetLab. We outline them in Section 2. We do not claim that either can satisfy all of the needs of PlanetLab users—instead, we use our experience to suggest features that may be present in the next generation of resource allocators. In doing so, we define some common needs of resource allocators, and in Section 3, suggest some features that should be present in an interface that can accommodate a reasonably diverse set of allocators. We discuss the interaction between the Resource Discovery and Resource Allocator services in Section 4, and between multiple allocators in Section 5. We finish by making a call to the designers of Resource Allocation services to promote responsible resource use.

2 Relevant Work

2.1 Emulab

Emulab [23] is software that runs shared testbeds for development, debugging, and evaluation of networked systems and applications. It began as a facility for emulating Internet-like conditions in predictable, repeatable environments, but has now expanded to encompass other types of experimental resources, such as static and mobile wireless nodes, simulation, and real-Internet resources including PlanetLab and the RON testbed [2]. Emulab uses the exported PlanetLab API [7] as a backend, and performs the role of all Infrastructure Services in a tightly-coupled manner.

Emulab offers an interface to PlanetLab and RON resources that allows users to ask for basic constraints on nodes, such as available CPU, memory, and disk space. It allows users to ask for nodes that are all at one site, or that are spread across as many sites as possible. On the RON testbed, it also allows users to ask for latency, packet loss, and bandwidth constraints on wide-area links; it should be fairly simple to adapt link constraints to PlanetLab once we have good path measurements, a problem which we and others [9] are working on. Emulab's node-constrained and link-constrained mappings are performed by different mappers, so it is not currently possible to do both in the same experiment.

2.1.1 What Emulab Has Today

Emulab's resource allocation is currently tied to its resource discovery phase. That is, it is the discoverer/assigner (called `assign`) [20] that both tracks currently available resources and solves the combinatorial optimization/constraint satisfaction problem of giving users resources that match their request. In practice, this works well, because we have a single goal (maximum throughput), and a single program attempting to achieve that goal.

For scheduling, `assign` does only admission control—either the submitted experiment is allocated resources immediately, or it is rejected. This works well for the dominant type of experiment on Emulab, interactive experiments. Emulab's fast instantiation allows users to use their resources within a few minutes. Interactive experiments can be seen as part of a larger class of experiments, *immediate experiments*, in which the user is not scheduling the experiment for some future time, but instead requesting that instantiation begin as soon as possible.

Emulab also allows *queued experiments*. If the queue flag is set for an experiment, then the experiment is placed in a scheduling queue, and is periodically run through `assign` to discover when sufficient resources are available for it. At this point, the experiment is instantiated and the user is notified via email. This is a best-effort service—no guarantee is made as to when the experiment will be instantiated.

When Emulab experiments are submitted, they are given a duration—a time bound after which the resources used for that experiment will be revoked. Users may change or remove this duration, but they must justify any change. Thus, experimenters are encouraged not to hold resources for longer than necessary. This procedure promotes fair sharing among users, which is important since Emulab resources are in high demand.

2.1.2 What Emulab is Building Now

Emulab has had, up to this point, a first-come-first-serve policy. We have started designing a hybrid system that allows:

- Immediate experiments
- Queued experiments
- Resource reservations

The first two types of experiments are described above. The third is straightforward in its semantics—a user will get some level of guarantee from Emulab that their experiment will be able to run during a given window in the future. However, it will likely require changes to the semantics of the other types of experiments, particularly immediate ones. For example, when a user tries to instantiate an immediate experiment, if some of the nodes they are given are reserved in the near future, then the duration of the experiment may have to be reduced with the user's consent (the alternative would be to not instantiate).

Emulab's way of dealing with scheduling during periods of high use in the past has relied on non-technical methods. Here, Emulab operators manually design a schedule or encourage the affected experimenters to contact each other directly to work out a schedule. The resulting schedule has been enforced only by nagging those who go over their allotted time. This method has worked surprisingly well. However, it seems clear that it will not scale to many users or long time periods, and more technical means will have to be employed to enforce schedules. Thus, the Emulab project is working to support resource reservations with automatic scheduling. However, this adds a new dimension, time, to an already NP-hard allocation problem, so it will be the subject of future research.

Emulab will also introduce pre-emption of experiments, in which a higher priority experiment may take resources from a lower-priority job. This pre-emption will likely be able to take three forms:

- Revoking in-use resources immediately
- Shortening the duration of an instantiated experiment (the first is a special case of this)
- Revoking or postponing a future reservation

In all cases, the resource allocator must inform affected users of the change.

The current Emulab "swapout" process does not preserve nodes' disk or memory state, so Emulab experiments have a "swappable" bit that indicates if they will lose critical state if pre-empted. Non-swappable experiments cannot be pre-empted; policy determines which experimenters are allowed to set this bit. In the future, Emulab will have stateful swapout so that all experiments can be pre-empted without data loss, allowing more aggressive pre-emption.

2.2 SWORD

SWORD is a resource discovery infrastructure for shared wide-area platforms such as PlanetLab. It has been running on PlanetLab for approximately one year. A user wishing to find nodes for their application submits to SWORD a resource request expressed as a topology of interconnected groups. A group is an equivalence class of nodes with the same per-node requirements (e.g., free physical memory) and the same inter-node requirements (e.g., inter-node latency) within each group. Supported topological constraints within and among groups include required bandwidth and latency. For example, a user might request one group of two nodes, each with load less than some value and that are at a particular site; and a second group of two nodes, each with load less than some value and that are within a required network latency of a particular reference node. The user might further request that there be at least one cross-group network link with at least 1 Mb/s bandwidth. An example of a SWORD resource request is shown in Figure 2.

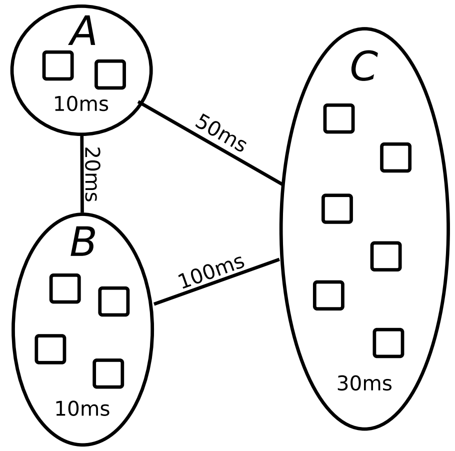

Figure 2: An example SWORD abstract specification. Three groups, A, B, and C are given constraints on maximum intra-group latency and constraints for the latencies between groups.

In addition to specifying absolute requirements, users can supply SWORD with per-attribute "penalty functions" that map values of an attribute within the required range but outside an "ideal" range to an abstract penalty value. This capability allows SWORD to rank the quality of the configurations that meet the application's requirements, according to the relative importance of each attribute to that application.

Architecturally, SWORD consists of a distributed query processor and an optimizer. The distributed query processor uses multi-attribute range search built on top of a peer-to-peer network to retrieve the names and attribute values of the nodes that meet the requirements specified in the user's query. SWORD's optimizer then attempts to find the lowest-penalty assignment of platform nodes (that were retrieved by the distributed query processor) to groups in the user's query—that is, the lowest-penalty embedding of the requested topology in the PlanetLab node topology, where the penalty of an embedding is defined as the sum of the per-node, inter-node, and inter-group penalties associated with that selection of nodes.

SWORD is an advisory service: it provides resource discovery, but not resource allocation. In particular, because PlanetLab does not currently support resource guarantees, a set of resources that SWORD returns to a user may (and probably will) no longer best meet the resource request at some future point in time. In light of this fact, SWORD supports a "continuous query" mechanism where a user's resource request is continuously re-matched to the characteristics of available resources, and a new set of nodes returned to the user. The user can then choose to migrate one or more instances of their application away from nodes that have become unsuitable.

2.3 Other Related Work

Researchers have developed, and in some cases deployed, a number of systems for resource discovery and resource allocation in wide-area distributed systems. Virtual Grids [11], Condor [12], Redline [13], and Network Sensitive Service Selection (NSSS) [10] provide resource discovery through a centralized architecture, while Ganglia [15] and MDS-2 [24] use a hierarchical architecture, and XenoSearch [22] and

SWORD [16] employ a decentralized architecture. All of these systems allow users to find nodes meeting per-node constraints, while Virtual Grids, NSSS, SWORD, and `assign` also consider network topology.

Resource allocation for wide-area platforms has also received recent attention. Such systems include SHARP [8] and Sirius [21]. Bellagio [3] is modeled after a virtual marketplace where users spend virtual currency in exchange for combinations of system resources.

3 Requirements for an Allocator Interface

We now apply our experiences to the interface for the Resource Allocator service. Rather than trying to define an API, we sketch the general shape of the interface, identifying properties we believe are essential.

3.1 Asynchronous Interface

We can describe the resource allocator as taking a set of jobs and returning a schedule of allocations. In practice, since the set of all jobs is not known in advance, the allocator must produce allocations for jobs as they are submitted. An allocator must consider the current state of PlanetLab, the resources it has allocated to other jobs, and the job or jobs it is attempting to schedule. We will call a scheduling for a particular job an "offer," and we suggest that the allocator should return zero or more offers for each job. Returning no offers is equivalent to refusing the job; we will discuss later why it might be advantageous to return more than one offer.

We require a callback mechanism between the submitter of the job and the resource allocator to support some of the job types we have defined: Queued jobs require notification when the job is ready to be instantiated, and pre-emption requires notification when a reservation changes.

Thus, we suggest an asynchronous interface to the allocator. Rather than returning an allocation immediately, the allocator may promise to call back the user within some specified amount of time with its decision. This allows Emulab-type scheduling where reservations are short and answers need to be quick. It accommodates scheduling more in line with the current PlanetLab model where applications may be long-lived, and thus the scheduler has longer to answer. It gives the allocator the option of considering more than one job at a time—it may accumulate requests over some period of time, then consider them simultaneously.

3.2 Resource Requests

A resource request is the specification submitted to the resource allocator by the user or an agent acting on a user's behalf. Here, we discuss some of the required content:

- An *abstract specification* of the user's requested resources
- A *resource specification* of the resources discovered by the Resource Discoverer
- A *time specification* from the user, indicating when they need the requested resources

3.2.1 Abstract Specification

The *Abstract Specification* is the representation submitted by the user, describing the desired resources. Our experience with Emulab and SWORD shows that relatively simple primitives are powerful enough to express a wide variety of constraints.

On Emulab, the user defines virtual nodes and links, and places constraints on them. Emulab has a simple type system, which allows the user to place a type constraint on any node or link, and each concrete resource to have a list of types that it can satisfy. This enables flexible specifications, allowing the user to request, for example, a generic PC, a class of PCs (such as those with 64-bit AMD processors), or a specific type of PC (say, a Dell PowerEdge 2850). This type system also allows Emulab to represent non-PC hardware such as routers and wireless sensor nodes. Links can also be typed (wired, Ethernet, wireless, 802.11, etc.), and constraints placed on their delay, bandwidth, and packet loss rate.

Emulab represents special node properties with what it calls *features* and *desires*. *Desires* are properties of nodes in the abstract specification, which are matched up to *features* in the concrete specification. Some examples of desires are:

- **Generic node resources** such as a specified threshold of available CPU or memory.
- **Special node hardware** such as SSL accelerators, extra processors, extra disks.
- **Node software features** such as operating system or available libraries.
- **Other node attributes** such as location (site, city, country, continent) or last-hop connectivity (Internet2, DSL, etc.).

In SWORD, the abstract specification takes the form of a topology of *groups*, with constraints on the intra-group and inter-group path properties. Per-node attributes such as CPU load and free memory can also be specified, in a similar manner to Emulab features and desires. These constraints can be both hard and soft, and users can supply utility functions that SWORD uses to rank alternative configurations that meet the soft requirements.

The units of specification of resources will be largely determined by the units that the underlying PlanetLab substrate supports. For example, PlanetLab's "shares" map roughly to process priorities. It is difficult to abstract this—if a user wishes to ask for, say x CPU cycles per second and y MB of memory, we cannot map this onto shares because the quantity of resources that a share represents varies across nodes and over time. PlanetLab is in the process of supporting CPU and memory guarantees. Once the transition is complete, Resource Allocators will be able to export higher-level abstractions to their users.

If a particular allocator requires the user to expend credits, real dollars, etc., the user's corresponding supply will be known internally in the allocator, or can be passed along with the authorization tokens. If it turns out that credit-based allocators become dominant, then it may be appropriate to include a maximum budget in the request that the user is willing to spend.

3.2.2 Resource Specification

A resource specification should list, unambiguously, the resources found by the resource discoverer. Identity of nodes in Emulab is straightforward; in PlanetLab, there are questions about how nodes should be identified (i.e., hostname, IP address, some GUID, etc.) We think that any sufficiently stable identifier should be sufficient for this purpose. The identifier should be tied to the allocation state for that node—as long as it has the same identifier, PlanetLab should attempt to preserve slivers on it. If a node is assigned a new identifier, then it should start with no slivers. This will help Resource Allocators keep track of their allocations.

Links should be specified at least in terms of type (i.e., wired vs. wireless), bandwidth, latency, and packet loss. Emulab uses all three of these. Other metrics such as jitter may also be valuable. All link metrics should be specified in each direction to support asymmetric link properties.

3.2.3 Time Specification

We suggest a time specification that consists of three parts: The start time, the end time, and a duration. Obviously, there is redundancy if all three are specified, but we think that including all of them gives flexibility, allowing for specification of all job types that we have identified.

An "immediate job" could be requested by giving only a start time, *now*, but no end time or duration. Or, it could include a duration or end time if the user knows how long they need the resources.

A "queued job" could be requested by giving no start time, but giving a duration and/or an end time. The duration would be taken to mean the amount of time the job is to run for, and the end time would be taken to mean the time at which failure should be reported to the user if the job has not been able to be instantiated.

Finally, a "reserved job" can be fully specified with a start time in the future. This also enables implementing "fuzzy ranges," where the user specifies all three criteria. In this case, if the interval between the start and end times is greater than the duration, the assigner is free to give resources at any point during the interval, with the specified duration.

If any fields are left empty, the allocator may choose to supply values for them in its return value to the user. For example, consider the case where the user did not specify an end time, but the allocator knows of future reservations for the same resources. In that case, we want the allocator to inform users that they may have the resources starting at the time they requested, but that those resources will later be revoked to satisfy another reservation.

3.3 Handling Multiple Potential Offers

We suggest that the interface allow allocators to return multiple offers for a job. The rationale behind this suggestion is that in complicated allocators, such as the ones based on economic models, there are value judgments that must be made. For example, is it worth x extra credits to the user to get their resources during the daytime rather than overnight? Is it worth y extra dollars to get within 2% of the desired latency instead of within 5%? An allocator might find multiple ways to schedule a job that fit the user's requirements, and in many cases, it makes sense to let the user choose between them.

We suggest three different ways that an allocator can handle multiple possible offers. The simplest, and perhaps the default behavior, would be for the allocator to choose a single allocation on behalf of the user.

Some allocators, however, might want to offer a chance for the user to be "in the loop" regarding which allocation is chosen. Such an allocator could make multiple offers to the user, who could choose from among them—interactive environments such as PluSH [1] and Emulab could easily present possible offers to the user.

More complex allocators might choose to let the user be "in the loop" by supplying an optimization function to the allocator to choose between multiple offers. While some work has already been done in this area (such as Condor's ClassAds [19], and SWORD's penalty function), it is clearly an open area for future work. Thus, we believe that for the time being, it is best to leave the language for optimization functions unspecified, allowing each allocator to pick its own.

4 Relationship to Resource Discoverers

A Resource Allocator's ability to make effective and efficient use of the available resources will be governed in part by how much information is available to it. Thus, its interaction with Resource Discoverers is key. An important factor in this interaction is how much data must be passed back and forth between the two. While node information should be manageable, path information is $O(n^2)$ in the number of nodes, and so passing information about all paths will scale poorly. It may be feasible to pass around simple measurements of latency, bandwidth, etc., for a testbed of up to thousands of nodes, but the problem becomes much worse when one considers allocators that wish to take into account more path metrics, or even time-series data about these metrics.

SWORD's current strategy for dealing with this wealth of information is to query for resources in two stages. First, it performs a query to select a set of *candidate nodes* which have attributes that satisfy the user's request. This may, for example, filter out overloaded nodes or those with full disks. Then, SWORD queries for the properties of paths only between pairs of candidate nodes. This means that it must consider much less data than if it were looking at all pairs of nodes.

While the discoverer can filter the data supplied to the allocator, ultimately it is still the allocator's job to decide which resources to use. As PlanetLab and the experiments and services that run on it grow large, complicated allocators will require more data than can easily be passed back and forth.

It seems inevitable, then, that one of two outcomes will occur. First, the resource discoverers may have to choose only to provide allocators with some subset of the available data. We believe this is problematic, for reasons discussed below. The alternative is to couple allocators more tightly with discoverers. We see this as an attractive option, for two reasons. First, resource discovery is an inherently decentralized process. As the work required of an allocator grows, so too will its need for processing power. One way of addressing this is to distribute at least some of the processing to the nodes that are collecting data. Second, all-pairs data collection simply does not scale, so resource discoverers of the future [14, 5] will have to use techniques to reduce the number of measurements required. This reduced set of measurements could be communicated efficiently to the assigner, but will require discoverer-specific knowledge to expand. Since the best way of taking these measurements is still an open research question, it is unlikely we will come to a consensus regarding the best way to represent them. In the meantime, we are left keeping resource discoverers and resource allocators tightly coupled.

If a discoverer chooses to give only some subset of its data to an allocator, that set must be chosen carefully, because the allocator will have specific goals in mind, and to get efficient and effective resource usage, the subsetting process must have at least some knowledge of the allocator's goals. If the Resource Discoverer and Resource Allocator are working at cross purposes, inefficiencies are virtually guaranteed.

While two nodes may be roughly equivalent in the view of the discoverer, the allocator may know that it has already reserved one of them, or that the user has not been granted permission to use another. When we consider large topologies, the chances that the discoverer happens to pick a set of free nodes goes down when resources are tight, leading to a high false negative rate. The problem is in fact worse than this—the resource discoverer and the resource allocator may be considering entirely different data (i.e., CPU load and economic cost), making the chances that the two decide on the same sets of nodes and times very small. Since one of the largest problems on PlanetLab today is resource usage during periods of contention, it is important to maximize the effectiveness of the discovery/allocation combination.

Thus, it seems valuable not to enforce a separation between these two components, allowing implementations that encapsulate both in a single component. Emulab, for example, may choose this approach, as it does currently. In the interests of efficient resource use, the community may eventually find that it is rarely if ever desirable to separate discovery and allocation. Hopefully, experience with a first version of this interface will be able to inform our decision about this in future versions.

We also suggest giving allocators the option of exporting some version of their current schedule. Before submitting resource requests, users prefer to have some idea of what resources are available, so that they know if they are likely to be scheduled soon. It also gives them a chance to consider revising their request if resources are tight.

5 Coexisting Allocators

A challenging issue for resource allocation on PlanetLab is how multiple resource allocators can operate simultaneously without interfering with one another. Except for PlanetLab resources, this issue does not currently arise for Emulab. It has a single resource allocation system with exclusive control over all non-PlanetLab resources. The allocator can perform admission control by refusing to schedule a user's job. In contrast, PlanetLab's design philosophy advocates "unbundled management" where multiple platform management services can coexist. In the current proportional-share model in PlanetLab, allocators do not have guaranteed control over any resources. Under this system, admission control is impossible, and each resource allocator can only make scheduling decisions based on the jobs submitted to it and the measured platform resource usage characteristics (which are induced by the jobs scheduled both by all resource allocators attempting to control the same set of underlying resources).

Thus PlanetLab currently supports only "best effort" resource allocators. SWORD, although billing itself as only a resource discovery system, is an example of such a system. Such best-effort techniques are also subject to oscillations. Lightly loaded nodes will tend to be noticed by many allocators, all of which may assign work to them, causing the nodes to become overloaded. Once the allocators notice the overload, all may migrate jobs off of nodes, causing them to become lightly loaded again. Coordination between allocators can reduce, but not totally eliminate, this effect.

PlanetLab is in the process of augmenting the base platform with a single platform-wide reservation system that will allow users or resource allocation systems to pre-reserve guaranteed fractions of a node's CPU cycles and physical memory). Although providing a single reservation mechanism "bundles" a small amount of management functionality into the PlanetLab platform, such a reservation system will allow resource allocators to offer better than best-effort service. In the absence of such a base allocator, there is a disincentive for people to use resource allocators that exercise admission control. For example if one allocator decides PlanetLab is too full, and as a result does not provide sufficient resources, then the user is likely to simply use a different allocator or deploy their experiment directly, even if this serves to overload PlanetLab. Users thus have a disincentive to using allocators that attempt to achieve the global good of not overloading PlanetLab.

Once resource guarantees are technically feasible, the policy decision of how the capacity they represent should be divided between allocators must be addressed. One model that has been suggested is that each allocator would receive some fraction of the total PlanetLab capacity, and allocators that prove to be more popular could have their fraction increased. When making this decision, it will be important to look not only at how "full" the allocator's allotment is, but also at how efficiently it is allocating resources. If an allocator has given out all of its guarantees, but users are not actually using them, it is not in the interest of the global good to give that allocator more resources. Instead, resources should be given to allocators that will use them efficiently. We think that one major point of competition between allocators will be how efficiently they allocate resources during peak times of contention; users will tend to prefer allocators able to deliver the highest level of utility. During times of peak demand, this will be allocators that are either efficient or sufficiently over-provisioned.

6 Responsible Resource Use

As PlanetLab moves to a scheme that guarantees some level of performance, it will be important for the allocators to provide a disincentive to holding resources that a user does not currently need. For example, while a user is learning the PlanetLab environment, or doing initial measurements, it is often unnecessary to hold resources around the clock or on many nodes. In another common usage mode, jobs that are "experiments" and not long-running services can often gather their data within a few days or a week, and then should be terminated.

The PlanetLab share semantics do not "charge" users for the number of nodes they allocate or the length of time nodes are held. Thus, this resource management solution has a fundamental problem in that it does not discourage users from grabbing shares on all nodes and holding them for months at a time, even if the user only needs a few nodes for a few days. Since shares do not represent guarantees, PlanetLab can hand out as many as it likes, and idle shares consume few node resources. However, as PlanetLab begins offering guarantees, every idle slice represents capacity that cannot be promised to new jobs that wish to use it.

This is a problem that Emulab has witnessed in practice. Before we implemented appropriate countermeasures to resource hogs, use of the Emulab cluster resources was inefficient. The testbed was often too full to admit new experiments. What was required were *incentives* for users to use resources responsibly, or *disincentives* for irresponsible use.

Our current solution to this is twofold. First, Emulab monitors nodes for "idleness," and reclaims resources after an experiment has been idle for a predefined period. Second, all submitted experiments contain a default expiration time, after which the experiment's resources will be reclaimed whether or not idle. It is easy for a user to disable or override both of these timeouts—both are presented on the job submission and modification interfaces. But, to do so, they must offer an explanation, usually a single sentence. If either timeout is about to expire, the user is notified to give them a chance to request the resources for longer. Occasionally adjusted in response to resource pressure, the inactivity timeout is set to a few hours, and the expiration timeout is set to approximate a workday. In our experience, this works reasonably well, as requiring a user to justify their node use discourages abuse of the system, and resources tend to be reclaimed quickly even if users forget to release them.

Allocators with an economic model, such as Bellagio [3], provide a different kind of incentive for responsible resource use. Through the use of abstract credits, a user budget throttles resource usage. To run a large experiment over a long time period, users carefully manage their budget. After completing the experiment, the user budget is likely to be sufficiently depleted to discourage unduly holding resources.

PlanetLab has an advantage over Emulab when it comes to reclaiming resources—it is easier to reclaim CPU guarantees than it is to reclaim whole nodes. An allocator could temporarily revoke a slice's CPU or memory guarantees without tearing down that slice's slivers. The allocator could change those guarantees to the minimal levels, or even move the slice to share-based scheduling. Thus, when the user is ready to use them again, it is a simple operation to restore previous guarantees (assuming there is capacity to hold them). Of course, the virtual machine that constitutes a sliver itself uses up some disk space, so slices that are expected to be idle for extended periods of time should be torn down, unless disk space is never an issue. There are "sensors" running on PlanetLab that track sliver use, such as CoMon [6], to assist in this process.

The high-level lesson we take from these experiences is that resources will not be used efficiently unless there is some incentive for users to use them responsibly. In our experience, the easier it is for a user to gain access to resources, the less of a sense they have that the available resource pool is finite across both space and time, and the less aware they are that their holding resources prevents others from using them. Simple mechanisms, even those that rely on social rather than technical means, can encourage responsible use, and need not be onerous for users who have a legitimate need for large, long-running jobs. We strongly recommend that all allocators contain some provisions to promote responsible use; doing so is in the allocator's best interest as well as the global interest of any shared computing infrastructure.

7 Conclusions

We take from our experiences with Resource Allocators and Resource Discoverers for PlanetLab five main lessons:

- The Resource Allocator interface should be **asynchronous**.
- The interface should allow for allocators supporting the **immediate, queued, and reserved** styles.
- **Resource guarantees** are necessary, both to allow new types of allocators, and to share fairly among allocators.
- Allocators should offer incentives for **responsible resource use**.
- Separation of Resource Allocators and Resource Discoverers may not be desirable. This separation **should be optional**.

Acknowledgments

We would like to thank the PlanetLab team for providing this valuable resource to the community. Our work was supported by the NSF under grants ANI-0205702, CNS-0335296, and CNS-0519879, as well as the Center for Networked Systems.

References

[1] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (SIGOPS-OSR)*, 40(1), January 2006.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Experience with an Evolving Overlay Network Testbed. *ACM Computer Communications Review*, 33(3):13–19, July 2003.

[3] A. AuYoung, B. N. Chun, A. C. Snoeren, and A. Vahdat. Resource Allocation in Federated Distributed Computing Infrastructures. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, October 2004.

[4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, Mar. 2004.

[5] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proceedings of ACM SIGCOMM*, 2004.

[6] CoMon. http://comon.cs.princeton.edu/.

[7] M. Fiuczynski, M. Huang, A. Klingaman, S. Muir, L. Peterson, and M. Wawrzoniak. PlanetLab Version 3.0. Technical Report PDN–04–023, PlanetLab Consortium, Jan. 2005.

[8] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *SOSP*, October 2003.

[9] N. Hu and P. Steenkiste. Exploiting Internet Route Sharing for Large Scale Available Bandwidth Estimation. In *ACM/USENIX Internet Measurement Conferece (IMC 2005)*, Oct. 2005.

[10] A.-C. Huang and P. Steenkiste. Network-Sensitive Service Discovery. In *USITS*, 2003.

[11] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. Chien. Efficient Resource Description and High Quality Selection for Virtual Grids. In *Proceedings of the IEEE Conference on Cluster Computing and the Grid (CCGrid 2005)*, 2005.

[12] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *ICDCS*, 1988.

[13] C. Liu and I. Foster. A Constraint Language Approach to Matchmaking. In *International workshop on Research Issues on Data Engineering (RIDE 2004)*, 2004.

[14] Y. Mao and L. Saul. Modeling Distances in Large-Scale Networks by Matrix Factorization. In *Proceedings of IMC*, 2004.

[15] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[16] D. Oppenheimer, J. Albrecht, D. A. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, 2005.

[17] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service Placement in Shared Wide-Area Platforms. Poster at SOSP 2005 poster session, 2005.

[18] L. Peterson and T. Roscoe. The Design Principles of PlanetLab. Technical Report PDN–04–021, PlanetLab Consortium, June 2004.

[19] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[20] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review (CCR)*, 32(2):65–81, Apr. 2003.

[21] Sirius. https://snowball.cs.uga.edu/˜dkl/pslogin.php.

[22] D. Spence and T. Harris. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *Proceedings of HPDC*, 2003.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

[24] X. Zhang and J. Schopf. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proceedings of the International Workshop on Middleware Performance (MP 2004)*, April 2004.