Developing and Evaluating Novel Network Protocols on Wide-Area Testbeds

Jeannie R. Albrecht
Master's Project Report

December 11, 2003

Abstract

As the popularity and availability of shared global testbeds continue to grow, researchers are placing less value on results obtained in simulation environments, and focusing more on the results obtained within wide-area testbeds such as PlanetLab. One consequence of this trend is that researchers are no longer able to control their experimental environment to the same degree that they can in simulation. Researchers on the wide-area can only hope to locate a set of nodes that possess the desired qualities at runtime. Currently, there is no service running on PlanetLab that supports global queries of this kind.

In this paper we present one solution to the problem of global resource querying on PlanetLab. Starting with an evaluation of TFRC, a novel network protocol, we describe some of the challenges that arise when testing experimental systems and protocols on wide-area testbeds. Focusing specifically on the problem of resource discovery, we propose a tool for use on PlanetLab that locates a set of machines based on a user-defined description of the desired testing environment. Users specify their request through an expandable XML query language. We describe the design and architecture of our tool and query language, and show the results from preliminary performance evaluations.

1 Introduction

When developing and evaluating novel network protocols, researchers typically go through several phases of experimentation. Initially, research is performed within a sophisticated simulation environment, such as ns2 [14]. Researchers have total control over the conditions their protocols are exposed to when using a simulator, and the same experiments are easily repeated. However these experiments often do not allow users to run real code, and they typically do not scale beyond a few hundred nodes. Many researchers use network emulators, like ModelNet [19], in addition to simulation, to further evaluate their systems. While not quite as controlled as the simulators, the latest network emulators give researchers many options to test their protocols. They offer the advantage over simulators of being able to run real code in real time. Emulators are also scalable to tens of thousands of nodes. Although experiments are not as easy to repeat in an emulator, the environment and test conditions are still well defined and fine tuned for specific research needs. The last round of testing involves running experiments on wide-area testbeds. Until recently it was difficult to find a large set of machines distributed across the wide-area that were open to experimental network research. However with the creation of shared global network testbeds, such as PlanetLab [16], researchers are now able run their code on machines spread around the world.

Before a new protocol can be publicly released, it must be thoroughly tested in an Internet-like environment to see how it will behave. One of the biggest criticisms of network simulators and emulators is that they do not accurately represent the Internet. There are currently no realistic models of the Internet available [7], which makes it difficult to simulate or emulate realistic network conditions. As a result, more and more researchers are turning to wide-area experimentation as a final test before distributing code to the public. Only in live wide-area testing do we know how a protocol will perform and interact with other systems on the Internet. There are many challenges that arise when using a shared wide-area testbed that do not exist in simulated and emulated environments. Most importantly, researchers no longer have total control over the tests they run or the machines on which they run. Changing network conditions, unconfigurable environments, and unrepeatable experiments make experimentation on the wide-area frustrating and tedious.

Researchers have responded to the instability of wide-area testbeds by devising ways to monitor their experimental environment. This involves measuring per node network and resource consumption periodically, and per-

haps even estimating future behavior based on past measurements. While this does not give them the control of simulation and emulation, it at least provides a way to know what the current conditions are on the testbed. If a specific set of conditions are desired for a given experiment, a researcher will attempt to hand-pick a subset of nodes that meet the desired criteria [4]. There currently are no system-wide services in place on PlanetLab that allow for automated resource discovery of this kind.

At the center of all of the problems that occur when running on the wide-area is the contention for resources among various users. In controlled testing environments, researchers do not have to be concerned that users at remote locations will affect their experimental results. This is not the case on shared wide-area testbeds. There are many users accessing the nodes simultaneously, each one desiring a different set of resources, creating a competitive environment. This has led to the proposal for several resource management and allocation schemes, which attempt to distribute the limited resources available throughout the testbed in a fair manner. These systems, combined with a way to efficiently discover resources on the testbed, are the first steps to creating a realistic, wide-area, testing environment that gives users the familiar control that is present in modern simulators and emulators.

In this paper, we discuss the development and evaluation of a novel network transport protocol that is an alternative to TCP. TFRC, or TCP Friendly Rate Control, offers a less drastic response to loss than TCP, which is desirable for applications that require smoother sending rates than TCP provides. When evaluating this protocol, we first briefly discuss our experiences within ModelNet, a network emulator. In this closed environment we were able to test the behavior of TFRC between one sender and receiver under very specific network conditions. Further, we were able to tune parameters to find the optimal configuration for the best performance. We then discuss our experiences of testing TFRC on the wide-area within the context of Bullet, a larger application designed for high bandwidth data dissemination.

Wide-area testing of TFRC and Bullet on PlanetLab motivated the latter part of our research. As we found ourselves repeatedly hand creating node lists based on our desire for a set of nodes with specific characteristics, we realized the benefits of an efficient resource discovery system for PlanetLab. In this paper we describe an architecture and query language for enabling resource discovery on PlanetLab. Using our tool, researchers are able to smoothly go from specifying resource constraints to automatic service instantiation on the correct set of nodes.

In constructing our resource discovery tool we are able to leverage the resource monitoring services that are publicly available on all PlanetLab nodes. By periodically gathering the measurement data from the nodes and storing it locally, we are able to perform queries for desired criteria across all PlanetLab machines. We also define an expandable query language that allows PlanetLab users to specify the exact set of resources needed for their experiments. In addition to a general query language, we allow users to make requests for groups of nodes with set sizes that satisfy all-pairs maximum latency requirements, as well as constraints for cross-group link latencies. We believe this tool gives users the flexibility they need to create a controlled testing environment.

The remainder of this paper proceeds as follows. Section 2 describes the development and evaluation of TFRC on ModelNet and PlanetLab. Section 3 illustrate the challenges associated with wide-area testbeds that do not exist in smaller, more controlled environments. In section 4, we discuss the problem of resource discovery on PlanetLab, and some of the existing services that help make the problem more tractable. Section 5 outlines the structure of our solution to resource discovery, and section 6 evaluates its performance. Future work and possible improvements to our resource discovery tool are shown in section 7, followed by a review of some related work in section 8. Finally, in section 9 we draw conclusions and summarize our experiences.

2 TCP Friendly Rate Control

This section discusses the development of a new transport protocol called TCP Friendly Rate Control, or TFRC. It is provided as an example of a novel network protocol that illustrates some of the difficulties encountered when building and performing evaluations on wide-area testbeds.

Although most traffic in the Internet today is best served by TCP, applications that require a smooth sending rate and that have a higher tolerance for loss often find TCP's reaction to a single dropped packet to be unnecessarily severe. TFRC targets unicast streaming multimedia applications with a need for less drastic responses to single packet losses [6]. TCP halves the sending rate as soon as one packet loss is detected. Alternatively, TFRC is an equation-based congestion control protocol that is based on loss events, which consist of multiple packets being dropped within one round-trip time. Unlike TCP, the goal of TFRC is not to find and use all available bandwidth, but instead to maintain a relatively steady sending rate while still being responsive to congestion. TFRC is less

aggressive than TCP, which is beneficial to applications where there might be multiple competing flows sharing the same links.

To guarantee fairness with TCP, TFRC uses the response function that describes the steady-state sending rate of TCP to determine the transmission rate in TFRC. The formula of the TCP response function [15] used in TFRC to describe the sending rate is:

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)}$$

This is the expression for the sending rate T in bytes/second, as a function of the round-trip time R in seconds, loss event rate p, packet size s in bytes, and TCP retransmit value t_{RTO} in seconds.

TFRC senders and receivers must cooperate to achieve a smooth transmission rate. The sender is responsible for computing the weighted round-trip time estimate R between sender and receiver, as well as determining a reasonable retransmit timeout value t_{RTO}. In most cases, using the simple formula $t_{RTO} = 4R$ provides the necessary fairness with TCP. The sender is also responsible for adjusting the sending rate T in response to new values of the loss event rate p reported by the receiver. The sender obtains a new measure for the loss event rate each time a feedback packet is received from the receiver. Until the first loss is reported, the sender doubles its transmission rate each time it receives feedback just as TCP does during slow-start.

The main role of the receiver is to send feedback to the sender once per round-trip time and to calculate the loss event rate included in the feedback packets. To obtain the loss event rate, the receiver maintains a loss interval array that contains values for the last eight loss intervals. A loss interval is defined as the number of packets received correctly between two loss events. The array is continually updated as losses are detected. A weighted average is computed based on the sum of the loss interval values, and the inverse of the sum is the reported loss event rate, p.

In our implementation of TFRC, we built an unreliable version of the protocol. The result is a transport protocol that is congestion aware and TCP friendly, but does not have the overhead of detecting and dealing with missing packets. This design is based on the assumption that lost packets are more easily recovered from other sources rather than waiting for a retransmission from the initial sender. Hence, we eliminate all retransmissions from TFRC.

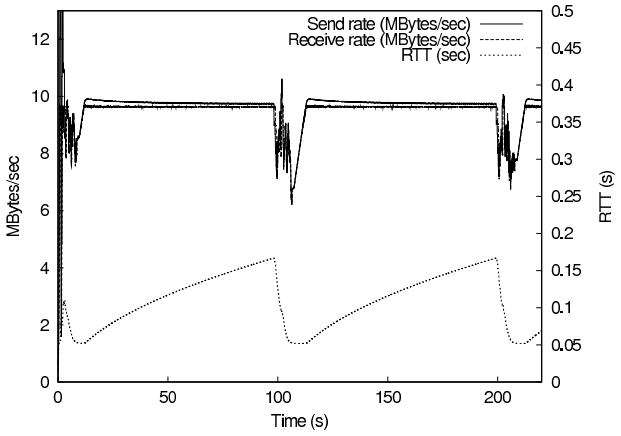

Figure 1: TFRC performance for a link with 50 ms delay between two end hosts in ModelNet.

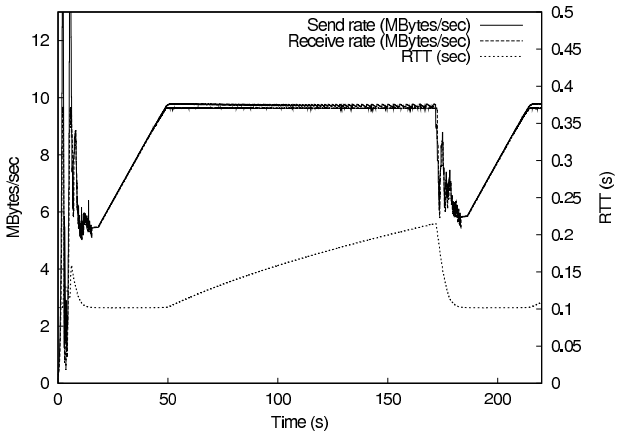

Figure 2: TFRC performance for a link with 100 ms delay between two end hosts in ModelNet.

2.1 Evaluation

When evaluating a network protocol such as TFRC, it is important to test the performance and behavior both in a standalone setting and in the context of an existing application. The following sections address both of these scenarios.

2.1.1 Standalone TFRC

Since the transmission rate of TFRC is based on the TCP throughput equation, the performance of TFRC under varying conditions is determined by the values of the variables in the equation. In particular, the achieved throughput of the receiver is greatly affected by the values of p, the loss event rate, and R, the round trip time estimate

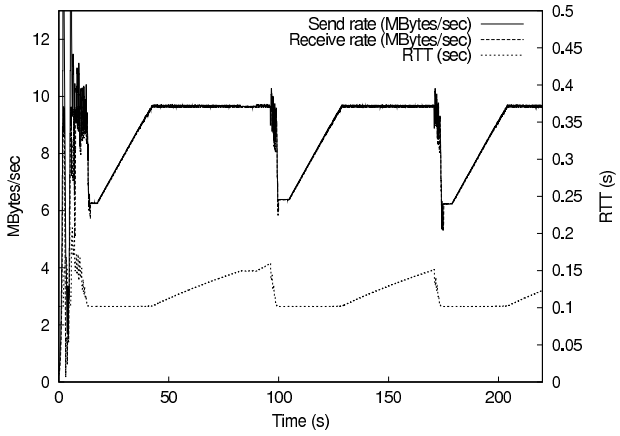

Figure 3: TFRC performance for a link with 100 ms delay between two end hosts in ModelNet. In this scenario we apply a lower smoothing factor to the RTT measurements.

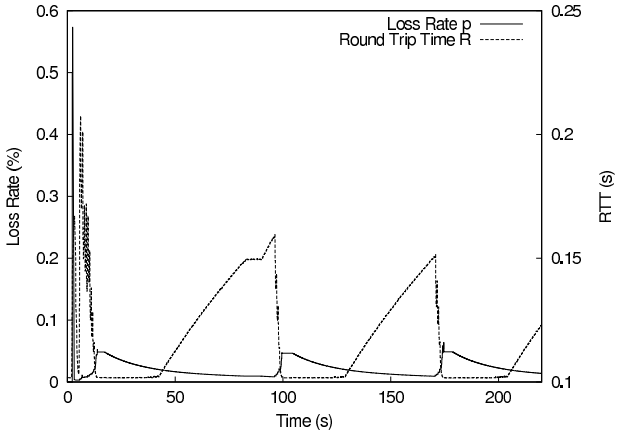

Figure 5: Loss rate versus Round trip time for the 100 ms link shown in Figure 3.

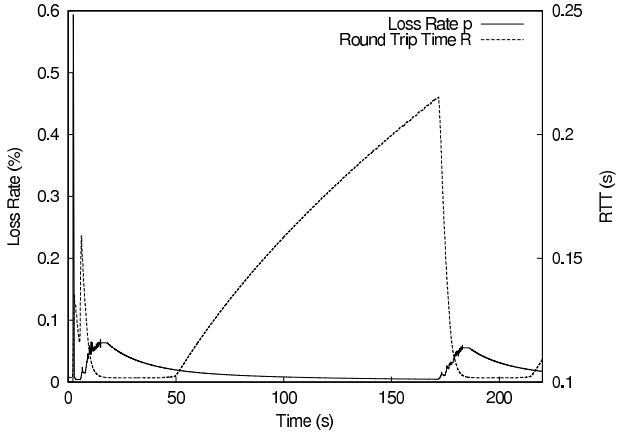

Figure 4: Loss rate versus Round trip time for the 100 ms link shown in Figure 2.

from sender to receiver. We will investigate the effects of both of these variables in this section. We used ModelNet [19], a network emulator, for the majority of the standalone testing. It provided a stable and controlled environment to evaluate the behavior of the system.

Figure 1 and Figure 2 show the effects of varying R (link latency) values. Notice that in Figure 2 the link delay is twice as high as the link delay of Figure 1, which means that feedback packets are received at a slower rate than in Figure 1. As a result, it takes about twice as long to reach the maximum transmission rate of approximately 10 MBps. Also shown in the graphs is effect of the congestion between the two ModelNet endhosts. As the maximum capacity of the link is reached, congestion causes the round trip time measurements to increase to more twice the link capacity, which eventually causes the sending rate to drop significantly in both graphs. These results show that the performance of TFRC is sensitive to changing values of R.

When computing the round trip times within TFRC, we use a smoothing factor in our measurements to help maintain a steadier transmission rate. We do this by using a weighted average of the RTT measurements. By placing a higher weight on previous measurements, we dampen the effect of sudden changes to R, causing the value of R to be smoother over time. Figure 3 shows the effect of this factor. In this graph, we placed a higher weight on new measurements, and ran the same experiment again, as is shown in Figure 2. The graph shows that by decreasing this smoothing factor to make changes to the value of R more drastic, the sending and receiving rates also become less smooth. The sending rate reacts to the change in R quicker than it did in Figure 2. This may be more desirable for some applications that must react quickly to congestion.

After evaluating the effect of changing values for round trip time R, we investigated the relationship between the loss rate p and R. Since both variables affect the sending rate, it is important to look at the behavior of both components. Also, we want to see if there was any relationship between the increasing RTT values and the loss rate that cause the sending rate to suddenly drop. Figures 4 and 5 show the correlation between p and R for the 100 ms links shown in Figures 2 and 3 respectively. Notice that as R increases and reaches its maximum, the loss rate p also rises. The is due to the fact that congestion affects both the loss rate and latency of a link. As the link becomes more congested, the round trip time continues to increase

until all packet queues are full. At this point packets are dropped, causing the loss event rate to increase. When both the value of R and p increase simultaneously, the sending rate drops significantly. Both variables have a significant impact on the sending rate.

When testing TFRC in a standalone setting, we initially tried running experiments on PlanetLab. However due to the lack of control, we were unable to single out specific parameters as we could in ModelNet. Other factors that were beyond our control, such as high CPU load, network congestion, and varying link latencies, made it difficult to perform accurate tests. The results we obtained were inconclusive. Fine-tuning performance factors and testing specific parameters are challenging tasks to complete in a dynamic, wide-area testbed like PlanetLab.

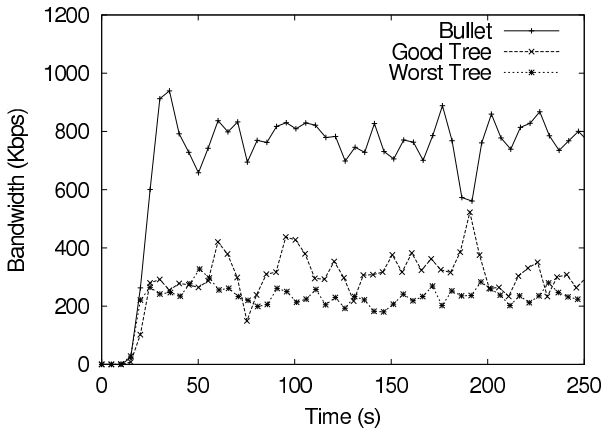

Figure 6: Achieved bandwidth over time for Bullet and TFRC streaming over different trees on PlanetLab with a root in Europe.

2.1.2 Using TFRC Within Bullet

When evaluating a new protocol like TFRC, in addition to performing evaluations as a standalone application, the protocol should be tested within the context of larger applications as well. Also, to fully evaluate the TCP friendliness when competing with other TCP flows, it must be run on the wide-area in a realistic Internet application. In this example, we chose Bullet [13] as the larger Internet application. Bullet is an algorithm for achieving high bandwidth data dissemination within an overlay mesh. Participants distribute deliberately disjoint data to their children, and the children leverage RanSub's [12] ability to pass random subsets of remote network state around the mesh to locate and retrieve missing data. The key insight is that the missing data can be received in parallel with the data from the parent node. Bullet uses TFRC as it's transport protocol, since there is no need for TCP retransmissions. We assume that it is quicker to get the missing data from someone else rather than wait for the sender to detect and correct any losses.

Unlike the standalone application where there is one sender and receiver, in Bullet you have many senders and receivers all using the same links. Congestion is expected, however as long as it is not too severe, TFRC can recover from it quickly without having a significant impact on the receiver's throughput. We evaluated TFRC's performance on 47 nodes on PlanetLab [16], a wide-area network testbed. Since we ultimately wanted to show that Bullet achieved higher bandwidth than traditional streaming over an overlay tree, we tested Bullet over TFRC against several hand-crafted trees. We also chose the source in the Bullet experiments to be constrained behind a high latency link. To do this, we used a source located in Italy, and the majority of the other nodes were in the United States.

To perform our experiment, we ran Bullet over a random overlay tree for 300 seconds while attempting to stream at a rate of 1.5 Mbps. We waited 50 seconds before starting to stream data to allow nodes to successfully join the tree. Then we compared this performance to data streaming over multiple hand-crafted trees. Figure 6 shows our results for two such trees. The tree labeled "good" in the graph was constructed by placing all nodes with high bandwidth and low latency high in the tree and close to the root. We used pathload [10] to measure the available bandwidth between the root and all other nodes to aid in this process. In this case, we are able to achieve a bandwidth of approximately 300 Kbps. The "worst" tree was created by setting the root's children to be the three nodes with the worst bandwidth characteristics from the root as measured by pathload. All subsequent levels in the tree were set in this fashion.

Performing this wide-area evaluation on PlanetLab was much more difficult than testing TFRC in a standalone environment using ModelNet. Results were inconsistent, and discrepancies among the nodes, such as clock skew, caused many problems in our analysis. Node failures were common, and widely varying network conditions made it hard to regenerate results for verification. This experience has led to the next part of our research which addresses some of the challenges associated with working on wide-area testbeds. In the next section, we will discuss some more of these problems in detail.

3 Challenges of Wide-Area Testbeds

Testing new protocols on wide-area testbeds introduces many more potential hazards and uncertainties than in other testing environments. Local-area testbeds, network emulators, and network simulators offer developers the ability to test their systems and protocols in controlled and manageable environments. On wide-area testbeds, many of these safety features are no longer present. Further, when running code across a large number of machines scattered around the world, several new challenges arise.

We now present some of the major challenges we had to address during our experimentation on PlanetLab. PlanetLab is a global network testbed for developing, deploying, and evaluating new protocols.

- There are no guarantees of reliability for data that is stored on remote machines on the wide-area. While some machines may employ some sort of daily snapshot, most do not. Storing anything irreplaceable on remote machines is a risk.
- Typically there is no global file system among machines on wide-area testbeds. Users often have to invent ways to efficiently keep their personal files up to date on all remote machines.
- Due to frequent and expected node outages, the testing environment is unstable. Further, since there are typically many users logged into each node on the testbed, resource availability may change drastically within short time intervals.
- While several proposals for distributed resource management on a wide-area testbed such as PlanetLab have been presented (see SHARP [8]) none are in place as of yet. Some initial per slice resource limits have been added to stop a small subset of the users from consuming large percentages of the resources. (A "slice" in this sense refers to a partition or share of global resources [8].)
- Wide-area testbeds are often heterogeneous testing environments. Every machine may be configured differently, and the hardware varies drastically from node to node. Each machine provides users with a different set of resources.
- Drastic clock skews are a common problem among PlanetLab nodes. Although there is a simple solution to this particular problem, it is often a low priority among system administrators.
- Since each machine is administered by a system administrator at a remote site, there is no consistency among nodes. Not all system administrators are as diligent as others about maintaining the nodes on the wide-area testbed. When a node goes down or needs maintenance, there is no guarantee that it will be repaired quickly.

In addition to these problems, on PlanetLab there is contention among the users for node and network resources. These resources include the CPU, bandwidth, disk space, physical memory, as well as many others. Since there is no easy way to determine what resources are available on each machine, some nodes are often extremely overloaded while others have resources to spare. Due to the constantly changing nature of this environment, it would be helpful to users if there was an efficient way to discover the available resources on all PlanetLab nodes. This would enable users to find a list of machines that meet the criteria they are looking for in an non-obtrusive manner. We address this task of building a resource discovery tool for PlanetLab in the next section.

4 Resource Monitoring Services on PlanetLab

The current techniques that many users employ while testing their applications and protocols on PlanetLab involve a variety of ad hoc methods. Researchers typically hand-pick nodes and links with characteristics that seem to fit their needs [4]. To further complicate the problem, as the number of users on PlanetLab continues to grow, so does the contention for resources. This makes finding a set of "good" machines increasingly more difficult. While testing Bullet (see Section 2.1.2), we needed a list of machines such that no more than 1 machine was located at each site, with 10 machines being located in Europe, and the others in North America. Maintaining this list was the most difficult aspect of testing. Machines would periodically go down, which meant we had to generate a new list and rerun all experiments. In other cases the load on a certain machine was so high that the program would barely run, again causing the list to be reconstructed.

By leveraging some of the monitoring services available on PlanetLab nodes, the problem of finding available resources is somewhat simplified. Here is a list of the monitoring data sets available on PlanetLab [2]:

- **All-pairs-ping**: The minimum, maximum, and average ping times measured between all pairs of Plan-

etLab nodes get stored in a text file that is updated every 30 minutes. Failed attempts also get noted in the file. (For more information, see `http://www.pdos.lcs.mit.edu/~strib/pl_app`.)

- **Ganglia sensors**: Ganglia collects node resource statistics on most PlanetLab nodes every 15 seconds. The data is available as a list of comma separated values on port 2841 at each node.
- **PLNetflow**: This service collects network statistics data every 5 minutes. It records the number of packets, number of bytes sent, network protocol used, IP addresses, and port for every slice on every PlanetLab node.
- **Scout**: Scout measures the number of bytes sent and received for each slice on every node. The information is updated every 5 minutes.
- **SliceStat**: This service measures the CPU, physical memory, and network bandwidth usage for each slice on each node. Data is collected every 5 minutes and stored locally on each node.

While all of these services provide useful information that is publicly available to researchers on PlanetLab, few users actually take the time to parse the information these monitors provide. This may be due to the fact that accessing the data on a local node does not provide any answers in terms of resource discovery. A user must know the measurements at all nodes at any given time to make a truly informed decision about resources. Even in this case, the decision is an estimate at best, since PlanetLab is a continuously changing environment. However, automated resource discovery using the monitoring data on PlanetLab to make a decision about available resources across all nodes would be more efficient and accurate than the common method of hand constructing node lists.

5 Our Solution to Resource Discovery

In this section, we describe the initial design of our resource discovery tool. Using the data gathered from the Ganglia monitoring service running on each node, we are able to generate a master list of machines that satisfy a user defined set of criteria for specific resources. We introduce an XML query language that allows users to specify their node resource requirements in an intuitive manner. In addition to resource discovery based on per node resource usage, we allow users to go one step beyond basic resource discovery and specify smaller groups of nodes that meet specific all-pairs latency requirements. The all-pairs-ping data service provides the needed data

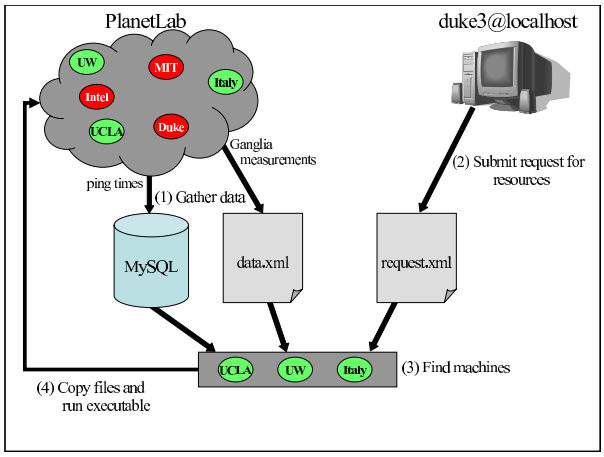

Figure 7: This diagram depicts how our resource discovery tool works. In step (1), latency measurements (ping data) and Ganglia resource consumption measurements are collected from all PlanetLab nodes and stored locally. Next, a PlanetLab user requests a set of machines with a specific set of resources (2). Upon receiving the request, our resource discovery tool finds the machines that meet the required criteria (3). In this case, the lighter colored nodes had the needed resources. Lastly (4), the necessary files are copied out to the desired nodes, and the executables are run.

to return results to these types of latency queries. Lastly, we enable users to request a minimum cross-group latency among their smaller groups of nodes. Note that while we focus on latency in our study, the techniques we present can be applied to any pairwise network metric.

For an example of how this process works, consider a user who is testing a new replicated service that has significant levels of load on the west and east coast of the United States. There are a set of replicated nodes on each coast, and they all must be able to communicate within the cluster with a low level of latency. At the same time, there is a requirement that synchronizing the replicas must occur within a certain time limit. To support this feature, each cluster periodically elects a leader who is responsible for transmitting data across the country with a maximum latency of 100 ms.

In this example, assume the user wants a list of nodes such that no more than 1 machine resides at each PlanetLab site. In addition, the user wants nodes who have at least 20% of their CPU unutilized, a load average of no more than "4.0" over the past 15 minutes, and a CPU speed of at least 1 GHz. For the west and east coast clusters, they want 2 groups of machines, such that the Group A (east coast nodes) has an all-pairs latency of no more

```
<?xml version="1.0" encoding="UTF-8"?>
<root date="Mon Dec  1 17:52:24 EST 2003">
 <node>
   <site>Carnegie Mellon University</site>
   <ip>128.2.198.196</ip>
   <hostname>PLANETLAB-2.CS.CMU.EDU</hostname>
   <boottime>1063759892</boottime>
   <bytes_in>82770.42</bytes_in>
   <bytes_out>445.61</bytes_out>
   <cpu_aidle>14.2</cpu_aidle>
   <cpu_idle>1.9</cpu_idle>
   <cpu_nice>0.7</cpu_nice>
   <cpu_num>1</cpu_num>
   <cpu_speed>1263</cpu_speed>
   <cpu_system>37.4</cpu_system>
   <cpu_user>61.4</cpu_user>
   <disk_free>11.129</disk_free>
   <disk_total>67.968</disk_total>
   <gexec>OFF</gexec>
   <load_fifteen>10.08</load_fifteen>
   <load_five>12.72</load_five>
   <load_one>12.43</load_one>
   <machine_type>x86</machine_type>
   <mem_buffers>80276</mem_buffers>
   <mem_cached>234568</mem_cached>
   <mem_free>6980</mem_free>
   <mem_shared>0</mem_shared>
   <mem_total>905012</mem_total>
   <mtu>1500</mtu>
   <os_name>Linux</os_name>
   <os_release>2.4.19-6_planetlab</os_release>
   <part_max_used>100.0</part_max_used>
   <pkts_in>106232272.00</pkts_in>
   <pkts_out>120.26</pkts_out>
   <proc_run>10</proc_run>
   <proc_total>792</proc_total>
   <swap_free>1152848</swap_free>
   <swap_total>2040244</swap_total>
   <sys_clock>1063760006</sys_clock>
 </node>
```

Figure 8: **data.xml**: Sample XML file describing the current Ganglia measurements on a PlanetLab node at CMU. The XML tags shown are identical to the metrics that Ganglia returns. On PlanetLab, each node returns a comma separated list of the measurements for each of these metrics.

than 50 ms, Group B (west coast nodes) has an all-pairs latency of no more than 70 ms, and at least one link between Groups A and B has a latency less than 100 ms.

The following sections describe the details of how this request would be satisfied. Figure 7 illustrates the entire process.

5.1 Gather Measurement Data

The first step is to gather all of the needed measurement data. This includes the Ganglia sensor measurements and the all-pairs-ping data. To collect the Ganglia sensor data, each node runs "curl http://127.0.0.1:2841/ganglia" and stores the output in a text file. We then collect all of these output files back to a local disk, combine the data, and parse it using JDOM (a Java representation of an XML

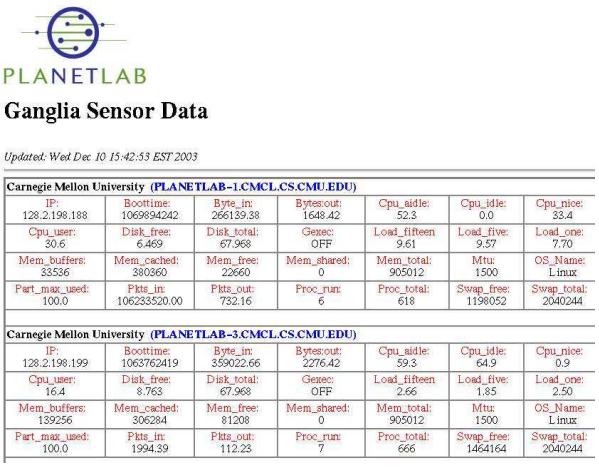

Figure 9: HTML document describing current Ganglia data for each PlanetLab node. This page is updated on an hourly basis. It is shown here to give an idea of what the page looks like. To read the actual values, see `http://www.cs.duke.edu/~albrecht/sensor.html`.

document) to create an XML document that describes each node's resource usage. Figure 8 illustrates a piece of a sample XML document that is created during this step. As an added convenience, we use XSLT, a language for translating XML to HTML, to create an HTML page that displays the current resource usage for each node. This page is updated and recreated hourly. A small screen shot of this page is shown in Figure 9.

The next step is to download the all-pairs-ping data and store it locally for future querying. This text file is retrieved every two hours. The data in the file is parsed and used to populate a table in a MySQL database using JDBC (Java Database Connectivity). Erroneous entries in the text file are ignored. The process of storing the ping data in the MySQL table and generating the XML document containing the Ganglia measurements is shown in step (1) in Figure 7.

5.2 Query Language

Once the data is gathered, we have the information needed to satisfy user requests for resources. In our tool, users describe their desired environment using an XML document. This is shown in step (2) of Figure 7. When defining the query language for requests, we propose a language that is expandable and easy to understand. It is very similar to the XML generated by the Ganglia data in the previous step. Figure 10 shows a sample request for the aforementioned example. We use JDOM again to parse the request and execute the query in the data file. Everything within the `<request>` tag

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <request>
    <numhosts>1</numhosts>
    <cpu_idle>20.0</cpu_idle>
    <cpu_speed>1000</cpu_speed>
    <load_fifteen>4.0</load_fifteen>
  </request>
  <group>
    <name>GroupA</name>
    <num_machines>4</num_machines>
    <latency>50</latency>
  </group>
  <group>
    <name>GroupB</name>
    <num_machines>5</num_machines>
    <latency>70</latency>
  </group>
  <constraint>
     <group_names>GroupA GroupB</group_names>
     <latency>100</latency>
  </constraint>
</root>
```

Figure 10: **request.xml**: Sample XML file describing a user's request for resources.

defines Ganglia metric requirements. In this particular case, <numhosts> refers to the number of machines located at each PlanetLab site. <cpu_idle> represents the percentage of unutilized CPU, which is 20 in this case. <cpu_speed> describes the speed of the processor, and <load_fifteen> is the average load experienced during the past 15 minutes. Notice how the tags and metrics in this request map directly back into the tags and data stored in data.xml in Figure 8.

The latter part of the request describes the groups needed for this experiment. The first <group> definition creates a group called "GroupA" with 4 machines (specified by <num_machines>) and a maximum latency of 50 ms. The second <group> creates a group called "GroupB" with 5 machines and a maximum latency of 70 ms. The final part of the request defines the cross-group <constraint>. In this case, GroupA and GroupB are the constrained groups, and they have a cross-group latency of 100 ms.

The request is not limited to the attributes, metrics, or groups shown in this example. Any of the Ganglia metrics recorded by the nodes and shown in data.xml can be used in a request for resources. The definition of groups and constraints is flexible to allow a variety of user specific scenarios to be created. Further, should additional services, measurements, or cross-group specifications be added in the future, the structure and simplicity of the XML used in our query language make it easy to support new features.

5.3 Master Node List and Group Creation

At this point in the process, we have the required data to fulfill the request, and the user has specified the exact resources needed. The first task is to generate the master node list, which consists of all nodes that have the required resources ignoring the group creation. To create this list, we start with a list of all PlanetLab nodes, and then eliminate nodes who do not meet the criteria specified. To determine which nodes meet these requirements, we can query the XML that was generated from the Ganglia data. At the conclusion of this process, we are left with a master node list that contains all nodes eligible for the group creation process.

Creating the groups of a specific size such that all links have a latency less than the stated maximum is an NP Hard problem. In fact, it is nothing more than an instance of the classic *k-clique* problem which asks whether or not a clique, or group, of a given size exists in a graph. More formally, the problem is defined as follows:

Given an undirected graph $G = (V, E)$, a clique is defined as a subset $V' \subseteq V$ of vertices that are fully connected such that each pair of vertices in V' are connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of the clique is the number of vertices it contains. The k-clique problem is the decision problem of determining if a clique of size k exists in the graph. In our case, the topology of PlanetLab is our graph, and we are looking for a subset of nodes (or vertices) of size k such that the pairwise latency between all k nodes is less than some threshold [18].

Assume we are looking for two groups of sizes m and n with the desired all-pairs latency property. Let N be the size of our master node list. In our example in Figure 10, m is 4 and n is 5. A naive algorithm for solving such a problem would be to enumerate all possible groups of size m, and all possible groups of size n, and and then systematically check the all-pairs latency in each group remembering that the two groups must be disjoint in the end. This algorithm is exponential. To add to the complexity of this already complicated problem, we have the extra constraint that the resulting groups must meet a cross-group latency requirement. Enumerating all possible solutions would be a very tedious and long running task.

Rather than check every possible combination, we try to approximate a solution to this problem. Although this approach does not guarantee that a solution will be found if it exists, it does find a solution in most cases, and it's running time is bounded by the size of the master list.

```
while(!done and counter<N) {
  foreach group {
  //1:  Find group of specified size
    newGroup=Checklatency(master_list, 0,
                          newVec, size, latency);
    if(newGroup.size()=size)
    //2a:  Group was found, so remove elements
      remove newGroup elements from master_list;
      groupList.add(newGroup);
    else
    //2b:  No group was found
       groupList.removeAllGroups();
       break;
  }

  if(groupList.size()!=0)
    //3:  Check for link between groups
    done=CheckCrossGroupLatency(groupList);
  if(!done)
    //4:  No link found. Shuffle list and retry
    restore_master_list();
    shuffle_master_list();
    groupList.removeAllGroups();
  counter++;
}

//Recursive algorithm for finding groups
nodeList CheckLatency(listNodes, count,
                      newVec, max, latency) {
  oldList=listNodes;   //Make copy of list
  first=oldList.pop(); //Get first element

  //Find links with desired latency in MySQl table
  nodes=execute(SELECT edges connected
                to first with ping < latency);
   newList.add(nodes);

  if(newList.size() < max-count-1) {
  //Not enough nodes to continue; restart
    if(oldList.size()==0)
      return newVec;
    else
      newVec.removeAllElements();
      return CheckLatency(oldList, count,
                          newVec, max, latency);
  } else {
    newVec.add(first); //Add element to final list
    count++;
    if(count==max) //We have found enough nodes
      return newVec;
    else  //Keep looking for more nodes
      return CheckLatency(newList, count,
                          newVec, max, latency);
  }
}
```

Figure 11: Pseudo code for our approximation to the full exponential search for groups that satisfy the given constraints.

The pseudo code for our algorithm is shown in Figure 11. We start by first finding a group of size m, and proceed to remove all nodes in the group from the master list of nodes. This step is shown in Comments 1 and 2a in Figure 11. Then we run the same procedure again, this time in search of a group of size n. If two groups are found, they are guaranteed to be two disjoint groups of size m and n. Otherwise (Comment 2b), we remove all groups, shuffle the master list, and try again.

Assuming two groups are found, next we check to see if any link exists between the two groups that is less than the maximum latency requirement, which is 100 ms in our case (Comment 3). If a link does exist, we have found a solution. If not, we add all nodes back to the master list, shuffle the list, increment our counter, and run the algorithm again. This is shown in Comment 4 of Figure 11. Due to the recursive nature of the group finding algorithm, shuffling the list increases the probability of finding two different groups than in the previous run. We continue this process until a solution is found, or until N (size of master node list) iterations have run unsuccessfully.

In practice, we find that this method works relatively well in a short amount of time. Most queries finish in a matter of minutes, and successful queries usually finish in under 10 seconds. Figure 12 shows an example of the output returned in this process.

Once the master node and group lists are created, the user is free to copy their code onto the PlanetLab nodes and run their experiment. We have a basic infrastructure set up for this purpose that uses scp to simultaneously copy files to all PlanetLab nodes designated in the master node list, and ssh to run the executable. At this point these steps are not integrated into the resource discovery process, although this may be added in the future.

6 Evaluation

To evaluate our tool, we analyze the running time of the two main components: the Ganglia data parsing and master list creation phase, and the group creation process. Also, since our tool measures resource consumption on PlanetLab nodes, we show an example of how we can evaluate the status of all nodes simultaneously for a specified resource. In this case, we look at CPU usage, which seems to be one of the most valuable resources on PlanetLab.

6.1 Ganglia Data Parsing Analysis

In Figure 13, we look at the time it takes our tool to find a master list of nodes when the size of PlanetLab is fixed at 139 machines. This allows us to analyze the overhead of adding a node to the master list. The majority of the data points lie between 650 and 700 ms, with some noise in the data causing minor oscillations in various spots. The slope of the line is slightly increasing, which means that as the number of nodes in the master list increases so does the completion time. This implies that there is a

```
> java FindHosts request.xml data.xml
Finding groups......done.

Master node list
------------------
  128.2.198.196
  150.135.65.3
  129.237.123.250
  160.36.57.174
  128.8.126.12
  12.46.129.23
  216.165.109.81
  171.64.64.217
  128.95.219.194
  158.130.6.253
  128.220.231.2
  128.84.154.71
  142.103.2.2
  141.213.4.202
  138.96.250.222
  131.243.254.35
  206.240.24.21
  69.28.151.3
  128.232.103.201
  198.133.224.145
  204.123.28.52
  128.42.6.144
Total nodes: 22


Groups
------------------
  Name: GroupA
  Latency 50.0 ms:
    128.2.198.196
    128.84.154.71
    158.130.6.253
    216.165.109.81

  Name: GroupB
  Latency 70.0 ms:
    150.135.65.3
    142.103.2.2
    141.213.4.202
    129.237.123.250
    128.95.219.194

Link that meets latency requirement
of 100.0 ms:

Source:  150.135.65.3
Destination:  128.2.198.196
Latency:  77.005
Reverse Latency:  88.686
```

Figure 12: Sample output after running the resource discovery tool for the request specified in Figure 10.

small overhead associated with adding nodes to the master list. However, since we must parse all of the data to check the resources of every node in PlanetLab for each request, the overhead is relatively small. We are essentially measuring the time it takes to add the hostnames to a list that is returned at the end of the procedure.

Figure 14 investigates the scalability of our resource discovery tool as more machines are added to PlanetLab. For this experiment, we maintained a constant master list size of 41 machines, and varied the total number of machines in PlanetLab. With a few exceptions, the slope of the line is approximately constant as we increase the number of

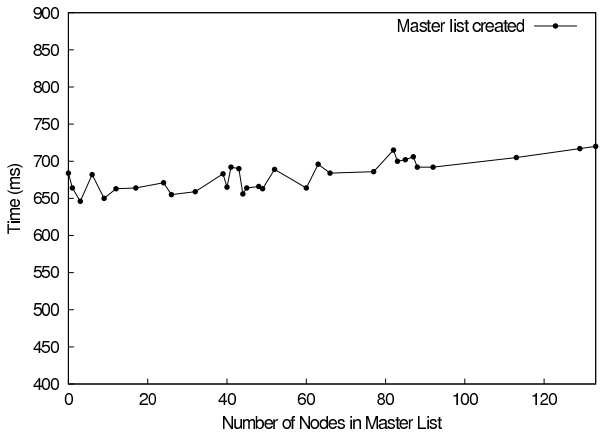

Figure 13: Time to locate a master node list versus the size of the completed master list. In this case the number of nodes in PlanetLab was fixed at 139 machines.

machines from 50 to 500. With 500 machines in PlanetLab, it takes 1 second to find a master list of nodes. We believe that it will continue to scale linearly in this fashion as the size of PlanetLab increases.

Another aspect of scalability that we must address is the bandwidth required to download the measurement data. To retrieve the Ganglia sensor data, we retrieve output files from all PlanetLab nodes once each hour. These files average 750 bytes in size. Currently, the Ganglia sensors run on 148 nodes in PlanetLab. This means that our file transfer requires 11 Kbytes of data to be sent over the network. Averaged over the hour between transfers, this works out to approximately 768 bps of network bandwidth. As we increase the size of PlanetLab, the Ganglia output file size will increase linearly. Each node will add approximately 750 bytes to the total size of the download. The current bandwidth required to retrieve the ping output is comparable to that of the Ganglia sensors. The most recent file size is 1.3 Mbytes, and this includes the data for about 250 nodes. This data is downloaded once every two hours, since the latency measurements tend to be more constant over time than the Ganglia resource statistics. Unlike the Ganglia sensor data, the size of the ping data file will increase quadratically with respect to the number of nodes on PlanetLab.

6.2 Group Creation Analysis

Figure 15 measures the time it takes our resource discovery tool to locate groups that meet the desired constraints. In this example, we varied the number of groups and the number of constraints in each request. Within the requests, we also varied the size of the groups and

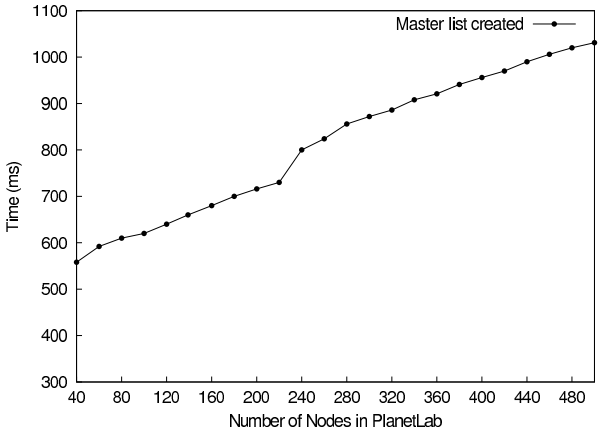

Figure 14: Time to find a master node list versus the total number of nodes in PlanetLab. Number of nodes in master list was fixed at 41 machines.

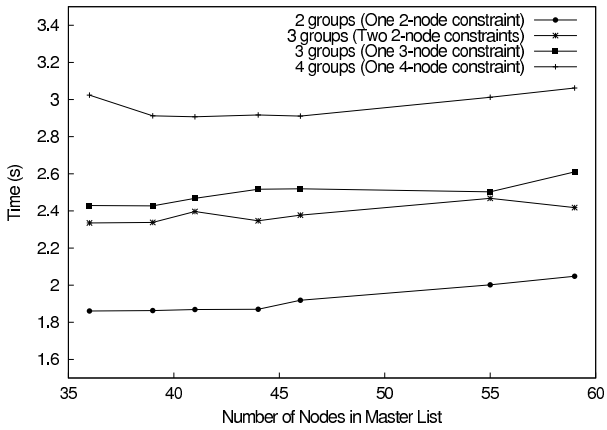

Figure 15: Time to find groups satisfying specific requests with varying numbers of groups and constraints versus the total number of nodes in the master list.

the requested latencies. For the line with 2 groups, we used the request shown in Figure 10, and varied the `<cpu_idle>` value to create master lists of different sizes. The line with 3 groups and two 2-node constraints requested 3 groups with 50 ms, 60 ms, and 70 ms latencies, and group sizes 3, 4, and 5 respectively. In this scenario, we requested that GroupA and GroupB have a cross group latency of 80 ms, and GroupB and GroupC have a cross group latency of 100 ms. For the third case, we requested the same three groups as in the previous case, however this time we requested a cross group latency among all 3 groups of less than 90 ms. In the final case, we added a fourth group to our request of size 4 and latency 80 ms, and asked for a cross group latency among all 4 groups of less than 150 ms.

The results show that overall, the maximum time to satisfy any of the requests shown is just over 3 seconds for all master list sizes. Increasing the number of groups causes the completion time to increase. This is due to the fact that more groups must be found, and therefore the algorithm must run for a longer period of time. Constraints that involve a greater number of nodes also take a longer time to satisfy than constraints with less nodes. The reason for this is because the number of links that must be found to meet the requirement increases at a rate greater than linear. For example, for a 3 node constraint, 3 links must be checked. For a 4 node constraint, 6 links must be checked. (The formula is $num_links = \sum_{i=1}^{k-1} i$, where k is the number of nodes in the constraint.) Thus it is easier to check two 2-node constraints with only 2 total links than to check one 3-node constraint with 3 links. The time to complete the requests rises slightly as the size of the master list increases in all four scenarios.

In order to avoid a full exponential search during the group creation process, we chose to implement an approximation that stops our algorithm after a specific number of iterations. Rather than enumerate all possible combinations of groups and nodes, our tool stops searching for groups after a maximum of N recursive attempts, where N is the number of nodes in the master list. To analyze the effect of our approximation, we look at the time it takes our tool to determine that no groups satisfying the given query can be found as a function of the master list size. We limit the experiments to cases where there is only 1 machine per site, since latency measurements between machines at the same site are approximately equal. The results to this experiment are shown in Figure 16. The curve is still exponentially increasing, however the rate of increase is smaller than a full exponential search would yield.

6.3 PlanetLab Node Analysis

Our tool gives PlanetLab users the ability to analyze the consumption of specific resources across all nodes simultaneously. In this section we give an example of how this can be used to evaluate the overall status of PlanetLab for a specific point in time. In this example, we chose to look at CPU usage. Figure 17 is a cumulative distribution function that shows the results of our study. In this graph, when searching for one machine per site, 75% of the nodes have resources available, while 25% percent have a CPU usage of 100%. If we look at 2 machines per site, only 60% of the nodes have a CPU usage of less than 100%. For 3 machines per site, the percentage drops even lower to 55% of the machines with less than 100% CPU usage. These results imply that while it is possible

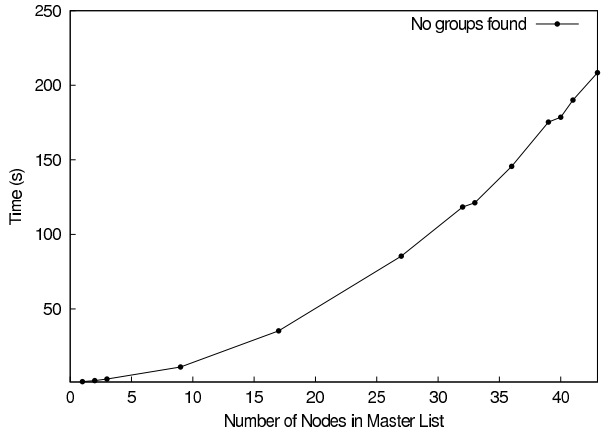

Figure 16: Time to determine that no groups satisfying a given request can be found versus the total number of nodes in the master list.

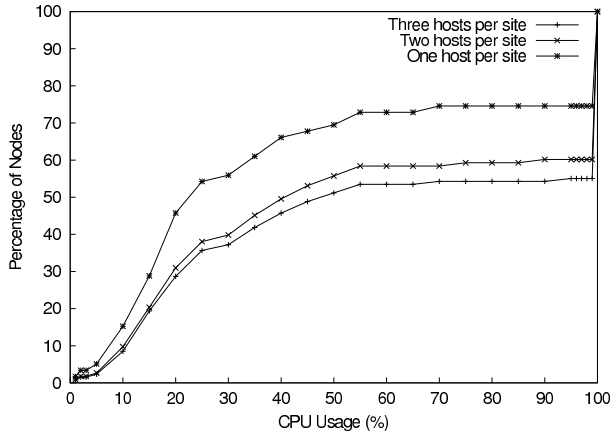

Figure 17: CDF of CPU usage for all machines on PlanetLab.

to find one node at each site with an available CPU, it is considerably harder to find two or three machines at each site that satisfy this constraint. This graph also shows us that the load across the machines is not equally balanced. In many cases, one machine at each site is significantly less loaded than the others. This is another reason why a tool for resource discovery is needed on PlanetLab. It provides a way to balance the load among all machines equally, since users will now be able to find the least loaded machine rather than randomly choosing one node per site.

7 Future Work

While the design of the resource discovery tool described in this paper solves some of the resource contention prob-

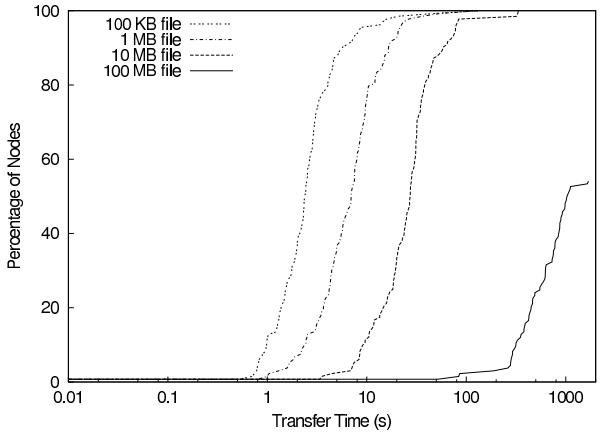

Figure 18: CDF of scp transfer time for varying file sizes across all machines on PlanetLab.

lems, there are many features that could be added to increase its effectiveness. In this section we discuss of few of the features we hope to add and improve upon in the future.

While latency measurements provide valuable information about link delays between PlanetLab nodes, perhaps an even more valuable metric is available bandwidth. Like the latency measurements, querying for pairwise bandwidth is also an NP Hard problem. However, the problem that must first be solved in this case is how to unobtrusively and accurately measure the available bandwidth between two nodes. Some solutions have been proposed, such as pathload [10] and Backbeat [17], and these services will certainly be considered as we continue to add features to our tool.

Another useful piece of information that would add to the flexibility of our resource discovery tool is the geographic coordinates of each node. It goal here is ultimately to support queries for x nodes in Europe and y nodes in North America. This would have been particularly useful in the Bullet experiments described in Section 2.1.2.

Our current solution to the problem of resource discovery is to download all measurement information to a central location and perform queries there. This is not a scalable solution. It would be better if we could decentralize the data in some way. One possible solution is to combine resource discovery with a resource management service such as SHARP [8]. In this case, agents running at each site could periodically pass resource consumption information to their neighbors, so that a particular request for resources would be forwarded through the network until a set of machine with the desired metrics are found. Another option is to employ the use a of distributed query

engine such as PIER [9].

As we mentioned before, our current techniques for copying data out to all nodes in the master node list and running executables is naive. It uses parallel scp connections to copy data to all nodes simultaneously, and parallel ssh sessions to run the executables in a similar fashion. A more elegant and quicker solution would be to use Bullet to disseminate the data to all nodes on the master node list. As described in Section 2, Bullet leverages the use of disjoint data and perpendicular downloads to send and receive large data files to all members of a group in an efficient manner. Figure 18 shows a CDF of file transfer times in Planetlab for files of varying sizes. While small files finish quickly, larger file have a much wider distribution of completion times. The longest time took over 30 minutes to complete, with almost half the nodes timing out after 10 minutes. Bullet will help make the transfer times across all nodes more equal, while also decreasing the total download time by using perpendicular bandwidth.

When finding groups that meet the specified criteria, we employ an approximation to the NP Hard k-clique problem. Other approximations to this problem exist given a specific network topology. Once we have a better idea of what the topology of PlanetLab looks like, we can implement more efficient algorithms to find groups that our current implementation provides.

8 Related Work

Sophia [20] is a network Information Plane that is current deployed on PlanetLab. Based on the ideas presented in [3] which describe an omniscient Knowledge Plan for networks, Sophia is a distributed system that stores, monitors, and adjusts to changing network conditions. There are three main components in Sophia. First, there are sensors distributed at each node that monitor node and network statistics. Second, Sophia uses a Prolog-like language to evaluate expressions and logic statements about the system. The third component is a set of actuators that perform local actions, such as killing processes, on the nodes. While this system is similar to our resource discovery tool, it only responds to requests about the condition of a specific resource at a given time. It does not handle user requests about multiple environmental conditions, and it does not support advanced group and cluster creation.

In [4], Considine, Byers, and Mayer-Patel argue that next generation wide-area testbeds should possess that same specifiable and repeatable behavior that is present in emulation and simulation. They go on to describe a constraint satisfaction method for finding a topology of nodes that meet a set of pairwise constraints, and note that this problem is NP Complete. The authors are essentially solving the same group creation problem that we are, however they do not address the per node resource consumption problem at all. They focus specifically on how to solve these types of problems using constraint satisfaction methods and intelligent search techniques on PlanetLab. In their example, they do mention that they eliminate all nodes with full file systems and CPU loads over 2.0, but they do not attempt to satisfy any other types of user specific resource constraints.

Grid technologies addressed the issue of resource discovery a few years ago as part of the Globus Grid toolkit [5]. The service that supports resource discovery in the Grid is called MDS2, which stands for Monitoring and Discovery Service. It includes an information provider framework called a Grid Resource Information Service (GRIS). Current implementations include static and dynamic host information, as well as network information. GRIS parses client requests and dispatches them to the appropriate information provider, who returns the results back to the client. This tool provides similar functionality to Grid users that we provide to PlanetLab users.

In [1], the authors present the design and implementation of INS, an Intentional Naming System. Using the scheme they propose, applications that use INS specify what they are looking for in the network, rather than specifying a specific location or hostname that describes where to find the needed resource. They propose a simple language based on attributes and values. To satisfy requests, INS request resolvers form an application level overlay network that discovers and monitors new services and resources. This system provides a way to locate services and resources based on what is being advertised and is known by the resolvers in the network. Sun's Jini [11] provides a similar framework for service discovery. It allows electronic components of all types to communicate and share their functions. Unlike our tool, it seems both INS and Jini have been designed more for locating physical devices in the network, such as a camera or printer, rather than node resources that are constantly changing.

9 Conclusion

Before a novel protocol can be released to the public, it must be tested in a realistic Internet-like environment. In the past, it was hard to find widely distributed machines that were open to experimental, network research. With the emergence of wide-area testbeds such as PlanetLab,

researchers now have the capability to expose their systems to live Internet conditions. However, these wide-area testbeds do not give users as much control over their experimental environment as simulation and emulation. When testing new protocols such as TFRC, there are several parameters that must be tuned for optimal performance. Without a way to control the testing environment, researchers resort to hand selecting a group of nodes that possess the needed resources at runtime. In this paper, we present a case study of a novel network protocol called TFRC, and describe the challenges that must be overcome when testing on wide-area testbeds. By leveraging the monitoring services available on PlanetLab, we propose a resource discovery tool and query language as a solution to the problem. Specifically, this paper makes the following contributions:

- We present a detailed analysis of TFRC. It is presented as an example evaluation of a novel network protocol that highlights the differences between emulators and wide-area testbeds, as well as motivates our work with resource discovery. We have also released a public version of TFRC that has been tested within Bullet and other large overlay systems.
- We outline the challenges involved with running experiments on wide-area testbeds like PlanetLab. Users cannot control their testing environments, and therefore must use resource discovery to locate the machines with the desired set of specifications before running any tests.
- We give a description of some of the monitoring services available on PlanetLab nodes. These services measure node and network resource consumption at regular intervals. The measurements are publicly available to all PlanetLab users.
- The design and performance of our resource discovery tool is described and evaluated. Resource discovery allows users to find nodes that have desired characteristics in an efficient manner. By leveraging the data provided in the monitoring services, our tool satisfies user requests for node resource statistics and network metrics.
- We define an expandable and generic XML query language that allows users to request specific node and network characteristics. Our language supports both per node resource usage characteristics, as well as pairwise network measurement specifications enabling the creation of smaller clusters of nodes. To support the latter feature, we provide an approximation to the NP Hard k-clique problem.
- An evaluation of the performance of the various components in our resource discovery tool is included. We also show how our tool can be used to analyze the status of all nodes on PlanetLab simultaneously.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Symposium on Operating Systems Principles*, December 1999.

[2] Brent Chun and Amin Vahdat. Workload and Failure Characterization on a Large-Scale Federated Testbed. Technical Report IRB-TR-03-040, Intel Research, November 2003.

[3] David Clark, Craig Patridge, J. Christopher Ramming, and John Wroclawski. A knowledge plane for the internet. In *Proceedings of ACM SIGCOMM*, August 2003.

[4] Jeffrey Considine, John Byers, and Ketan Mayer-Patel. A Constraint Satisfication Approach to Testbed Embedding Services. In *Proceedings of ACM HotNets-II*, November 2003.

[5] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing, 2001.

[6] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer. Equation-based congestion control for unicast applications. In *SIGCOMM 2000*, pages 43–56, Stockholm, Sweden, August 2000.

[7] Sally Floyd and Eddie Kohler. Internet research needs better models. In *Proceedings of ACM HotNets-I*, October 2002.

[8] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[9] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, September 2003.

[10] Manish Jain and Constantinos Dovrolis. End-to-End Available Bandwidth: Measurement methodology, Dynamics, and Relation with TCP Throughput. In *Proceedings of ACM SIGCOMM*, August 2002.

[11] Jini. http://java.sun.com/products/jini, 1998.

[12] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2003.

[13] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Streaming Using and Overlay Mesh. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[14] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[15] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, Vancouver, CA, 1998.

[16] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[17] Patrick Reynolds and Amin Vahdat. Backbeat: Coordinated Probes for Distributed Systems. In *Submission for NSDI*, 2004.

[18] Charles E. Leiserson Thomas H. Cormen, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter NP-Complete Problems. McGraw-Hill Book Company, 2001.

[19] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[20] Mike Wawrzoniak, Larry Peterson, and Timothy Roscoe. Sophia: An Information Plane for Networked Systems. In *Proceedings of ACM HotNets-II*, November 2003.