Management Authority: Reference Implementation

GDD-06-39

GENI: Global Environment for Network Innovations

January 2, 2007

Version 0.2

Authors:

Larry Peterson, *Princeton University*

Aaron Klingaman, *Absaroka Software*

Steve Muir, *Princeton University*

Mark Huang, *Princeton University*

Andy Bavier, *Princeton University*

Vivek Pai, *Princeton University*

KyongSoo Park, *Princeton University*

This work is supported in part by NSF grants CNS-0519885, CNS-0335214, ANI-0335214 and CNS-0439842

Revision History:

Version	Changes log	Date
v0.1	Original version posted	12/06/06
v0.2	Original contents consolidated into section 2. New sections 3 and 4 created for Auditing and Monitoring.	01/02/07

Table of Contents

1 Introduction

This note describes the PlanetLab *management authority* (MA), and the role it plays in operations and maintenance control for PlanetLab. It serves as an example implementation of an MA for GENI, with a particular focus on mechanisms to (1) securely boot and configure nodes, (2) audit Internet traffic, and (3) monitor node health. We make no claims that this is the only way to implement this functionality, or even the best way. However, the following does describe a system that has proven robust across a wide-range of circumstances, and in doing so, hopefully identifies some of the issues that any MA will face. We also note that working code that corresponds to this system is available at www.planet-lab.org.

Note that the terminology used in this note is PlanetLab-specific, but the mapping onto GENI terminology should be obvious from the context.

2 Booting

PlanetLab Central (PLC), acting as a *management authority* (MA), is responsible for installing and updating software (e.g., the VMM and Node Manager) running on the nodes it manages. This section describes the process by which nodes boot and join PlanetLab.

2.1 Boot Server

PLC runs a *boot server* that nodes contact to retrieve the executable programs they are required to run to be a part of PlanetLab nodes. These include the a Linux-based VMM and the Node Manager control program.

PLC also maintains a database of registered nodes. Each node is affiliated with an organization (owner) and is located at a site belonging to the organization. The current implementation includes a database with the following tuples:

principal = (name, email, org, addr, keys, role)

org = (name, address, admin, sites[ ])

site = (name, tech, subnets, lat_long, nodes[ ])

node = (ipaddr, state, node_key, node_id)

where

state = (install | reinstall | boot | debug)

role = (admin | tech)

The admin field of each org tuple is a link to a principal with role = admin; this corresponds to the primary administrative contact for the organization. Similarly, the tech field in the site tuple is a link to a principal with role = tech; this is the person that is allowed to define the node-specific configuration information used by the node's slice creation service when the node boots.

The node state indicates whether the node should (re)install the next time it comes online, boot the standard version of the system, or come up in a safe (debug) mode that lets PLC inspect the node without allowing any slices to be instantiated or any network traffic to be generated. The MA inspects this field to determine what action to take when a node contacts it. Nodes are initially marked (in the MA database) as being in the install state.

The management authority supports a public interface that is used by node owners to register their nodes with PLC. An organization (node owner) enters into a management agreement with PlanetLab through an offline process, during which time PlanetLab learns and verifies the identities of the principals associated with the organization: its administrative and technical contacts. These principals are then allowed to use this interface to upload their public keys into the MA database, and create database entries for their nodes. These operations include:

```
node_id = AddNode(auth, node_values)
UpdateNode(auth, node_id, node_values)
DeleteNode(auth, node_id)
```

where node_id is a unique identifier for the node and node_values is a structure that includes the node's name and IP address, among other information.

This public interface also supports operations that allows users to learn about the set of nodes it manages, so that they know the set of nodes available to deploy slices on:

```
node_ids[ ] = GetManagedNode(auth)
node_values[ ] = GetNodes(auth, node_id)
```

Today, this operation returns the DNS name and IP address of a set of nodes managed by PLC. At some point, it makes sense to publish a broader set of attributes for available nodes. These attributes would most naturally be represented as an RSpec.

2.2 Nodes

A node boots with three pieces of state in a persistent, write-protected, store: a bootfile, a network configuration file named plnode.txt, and a public key for PLC. All three pieces of information are created through an offline process involving the node owner and PLC, and installed in the node by the owner.

The bootfile is an executable image that the node is configured to run whenever it boots. This program causes the node to interact with PLC to download all necessary software modules. plnode.txt is a node-specific text file that is read by the executable booted from the bootfile. It gives various attributes for the node, including its node_id, DNS name, and IP address, along with a unique node_key generated by PLC and assigned to the node. The following is an example plnode.txt file:

```
IP_METHOD = "DHCP"
IP_ADDRESS = "128.112.139.71"
HOST_NAME = "planetlab1.cs.foo.edu"
NET_DEV = "00:06:5B:EC:33:BB"
NODE_KEY = "79efbe871722771675de604a2..."
NODE_ID = "121"
```

Today, the bootfile is available on a CD (called the PlanetLab BootCD), and plnode.txt is available on either a floppy or USB device. In general, they could be combined on a single device.

The node uses the node_key from plnode.txt during the boot process to authenticate itself to PLC. PLC trusts that the node is physically secure. Note that PLC also assumes that there is a means by which PLC can remotely reboot a node. The preferred implementation is an on-machine reboot capability (e.g., HP's Lights-Out product), although other implementations are possible. The mechanism should also support console logging.

2.3 Boot Manager

Nodes use a private interface to contact a PLC boot server when they come up. Specifically, the bootfile available on each node contains a minimal Linux system that initializes the node's hardware, reads the node's network configuration information from plnode.txt, and contacts PLC. (The bootfile actually identifies a sequence of potential boot servers for the node to contact, which are tried in order until the note successfully boots.) The boot server returns an executable program, called the *boot manager* (approximately 20KB of code), which the node immediately invokes.

The boot manager, now running on the node, reads the node_key from plnode.txt, and uses HMAC [1] to authenticate itself to the MA with this key. Each call to the MA is independently authenticated via HMAC. The MA also makes sure the source address corresponds to the one registered for the node, to ensure that the right plnode.txt has been put in the right machine, but this is only a sanity check, as the server trusts that the node is physically secure.

The first thing the node learns from the MA is its current state. If the state = install, the boot manager runs an installer program that wipes the disk and downloads the latest VMM, NM, and other required packages from the MA, and chain-boots the new kernel. The downloaded packages are also cached to the local disk. A newly installed node changes the node's state at the database to boot so that subsequent attempts do not result in a complete re-install.

If the node is in boot state, the boot manager generates a public-private key pair to be used in authenticating service slices once the node has booted, and requests that the MA generate a public key certificate that binds the public key to this particular node. It then contacts the MA to verify whether its cached software packages are up-to-date, downloads and upgrades any out-of-date packages, and finally chain-boots the most current kernel. Otherwise, if the boot manager learns that the node is in debug state, it continues to run the Linux kernel it had booted from the bootfile, which lets PLC operators ssh into and inspect the node. Both operators at PLC and the site technical contacts may set the node's state (in the MA database) to debug or install, as necessary.

In addition to boot-time, there are two other situations in which the node and MA synchronize. First, running nodes periodically contact the MA to see if they need to update their software, as well as to learn if they need to reinstall. Each node currently does this once a day. Second, whenever the node state is set to debug in the MA database, the MA contacts the node to trigger the boot process, making it possible to bring the node into a safe state very quickly.

Figure 1 gives a flow chart for the boot manager.

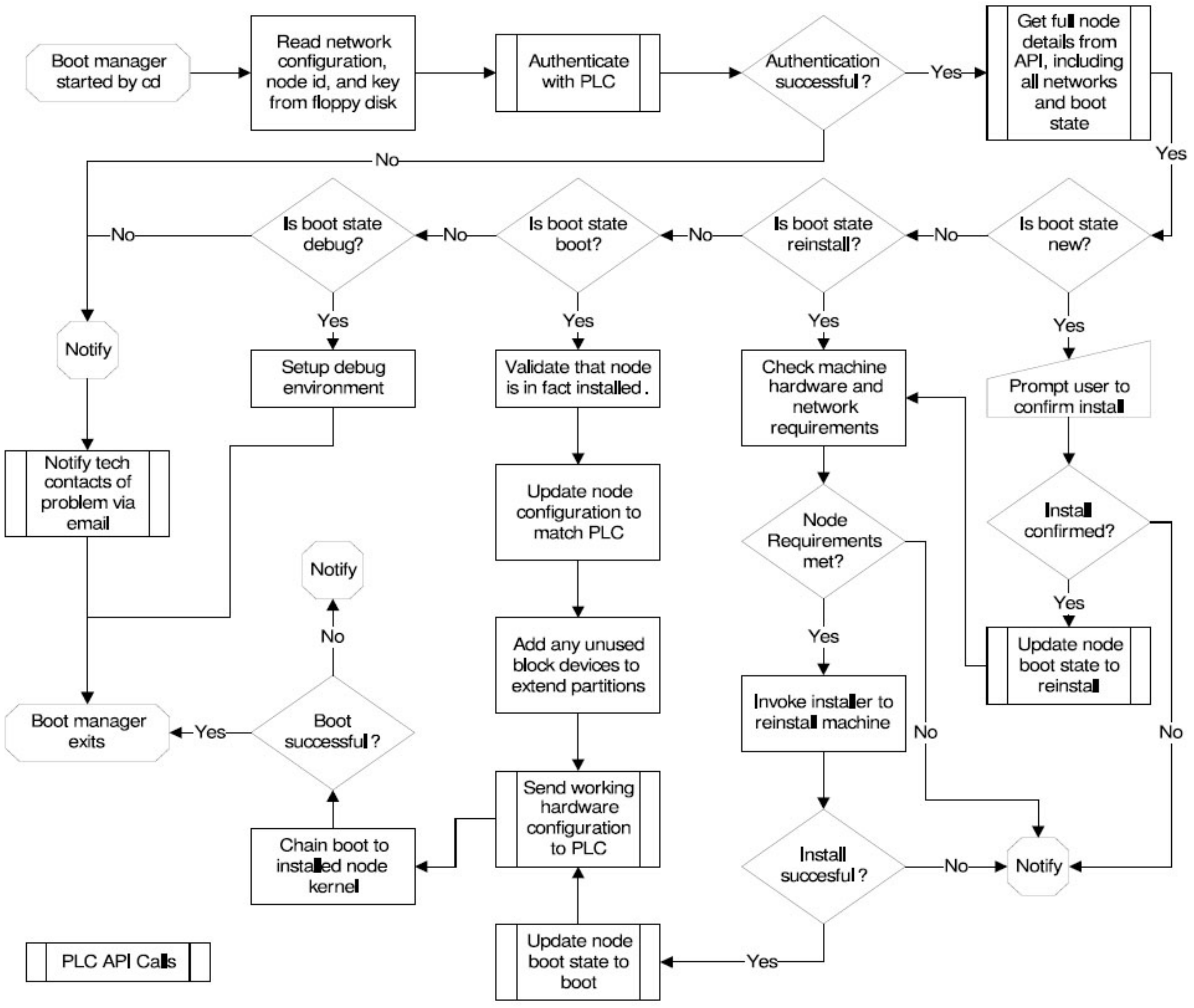

Boot manager started by cd
Read network configuration, node id, and key from floppy disk
Authenticate with PLC
Authentication successful ?
Yes
Get full node details from API, including all networks and boot state
Yes
No
Is boot state new?
No
Is boot state reinstall?
No
Is boot state boot?
No
Is boot state debug?
No
Yes
Yes
Yes
Yes
Notify
Setup debug environment
Validate that node is in fact installed.
Check machine hardware and network requirements
Prompt user to confirm install
Notify tech contacts of problem via email
Update node configuration to match PLC
Node Requirements met?
Install confirmed?
Yes
Yes
Notify
No
Add any unused block devices to extend partitions
Update node boot state to reinstall
Invoke installer to reinstall machine
No
No
Boot manager exits
Yes
Boot successful ?
Send working hardware configuration to PLC
Chain boot to installed node kernel
Install succesful ?
No
Notify
Update node boot state to boot
Yes
PLC API Calls

Figure 1. Boot Manager Flow Chart

2.4 Example Session

The following gives an example session of the boot manager for a new node being installed and then booted.

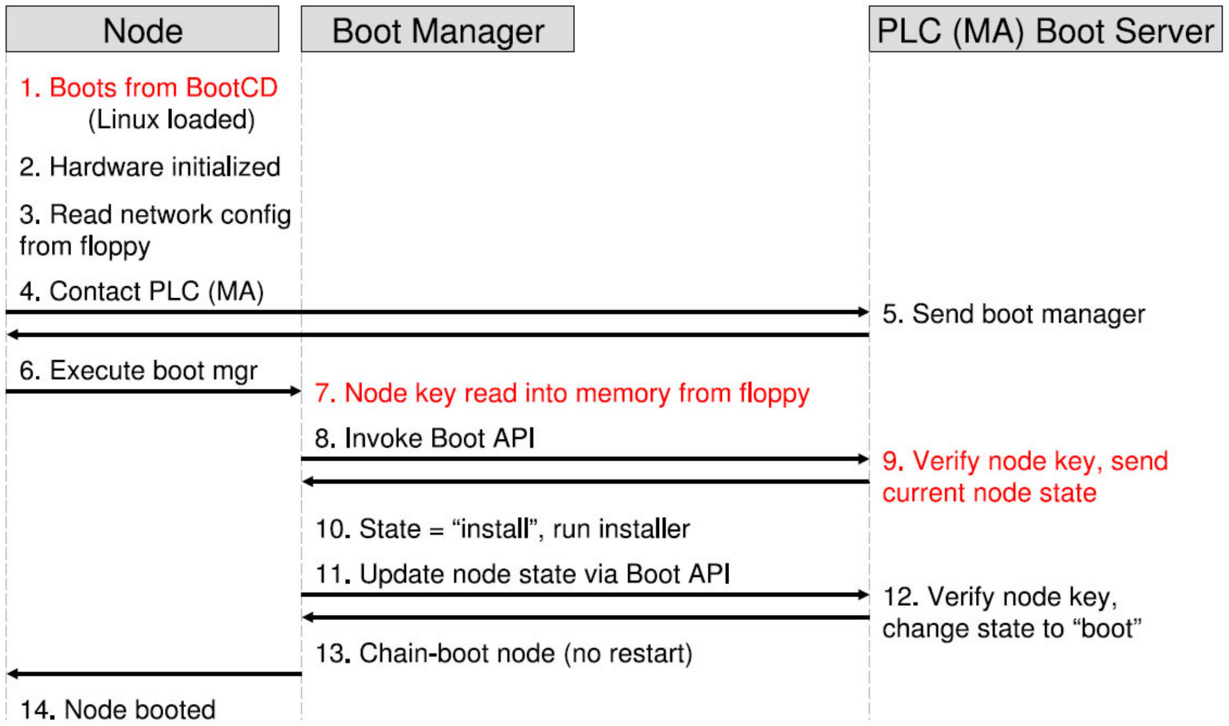

Node
Boot Manager
PLC (MA) Boot Server
1. Boots from BootCD (Linux loaded)
2. Hardware initialized
3. Read network config from floppy
4. Contact PLC (MA)
5. Send boot manager
6. Execute boot mgr
7. Node key read into memory from floppy
8. Invoke Boot API
9. Verify node key, send current node state
10. State = "install", run installer
11. Update node state via Boot API
12. Verify node key, change state to "boot"
13. Chain-boot node (no restart)
14. Node booted

Figure 2. Example Boot Manager Session

3 Auditing

PLC runs a network auditing service, called PlanetFlow, that provides comprehensive accountability for all traffic that leaves the PlanetLab network [2]. PlanetFlow provides a public interface as easy to use as the WHOIS database for quickly determining accountability, thereby enforcing the chain of responsibility that exists between PlanetLab, its researchers, and its hosting sites. When and end-user or the administrator of an outside network notices unwanted or unrequested traffic from a PlanetLab node and wants to file a complaint, PlanetFlow makes it possible to map information about the traffic into the responsible slice, and hence, the responsible researchers.

PlanetFlow runs in a regular, unprivileged slice on every PlanetLab node and performs all privileged operations (such as reading packet headers generated by other slices, an operation not normally allowed) through the Proper service [3]. PlanetFlow consists of four primary components:

- a flow collector that classifies outgoing packets into IP flows;
- a database that stores the flows;

- web and administrative interfaces for querying the database; and
- a central server for storing, querying, and archiving flow data from all nodes.

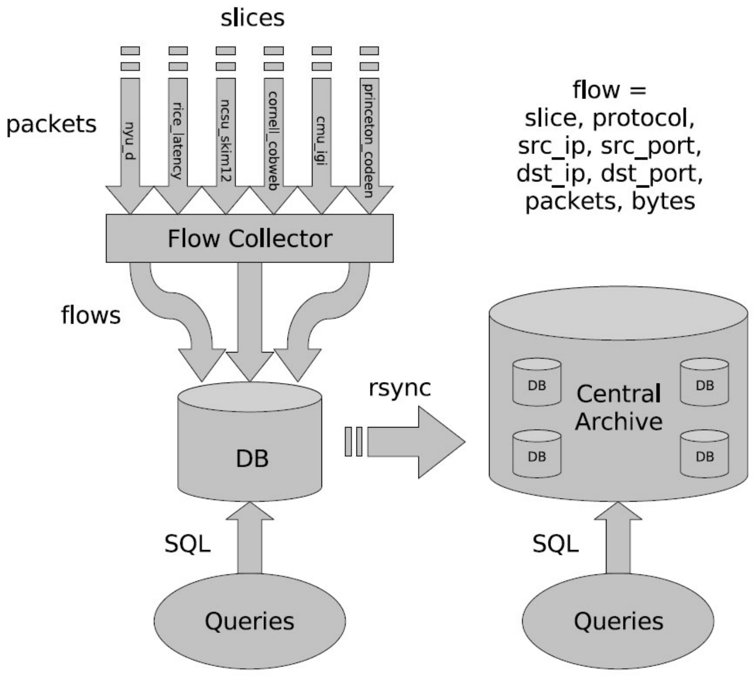


Figure 3: PlanetFlow components.

The current implementation of PlanetFlow is optimized for low overhead, fast insertion time, reasonable query time, and archivability. Each component of the implementation was chosen carefully to meet these specific goals. Ease of implementation and flexibility in both maintenance and usage were also considered. PlanetFlow typically adds less than 1% CPU overhead to a system; in the worst case, the overhead reaches 3%.

3.1 Flow Collector

PlanetFlow captures the IP headers of all outgoing packets, as well as their associated meta-data (e.g., originating slice and timestamp), with the ulogd program. To conserve space, packet headers are not recorded individually, but are instead classified into flows for which daily packet and byte counts are kept. The advantage of using ulogd instead of a program like tcpdump is that ulogd reads packets from the kernel in batches through a Netlink socket [4] in order to reduce context switching. A single recvmsg() call on a Netlink socket can transfer a batch of 64 or more packet headers. To transfer the same number of packet headers through a raw packet socket, such as one that tcpdump would open, recvmsg() would have to be called 64 times. Additionally, because Netlink data can be multicast, it is possible for multiple flow collectors to run simultaneously in different slices, without incurring significant additional overhead.

We extended ulogd to perform flow accounting for PlanetFlow. ulogd now classifies every outgoing IP packet into a unique flow keyed on the following attributes:

- Slice ID
- IP protocol number
- IP source address
- TCP or UDP source port, ICMP Echo ID, GRE key, or PPTP Call ID
- IP destination address
- TCP or UDP destination port

ulogd maintains a bounded cache of active flows, and stores a cumulative packet and byte count for each flow. Every five minutes, ulogd empties its cache into a database. Flows spanning multiple five minute time intervals are aggregated daily. Previous versions of PlanetFlow did not aggregate flows, which caused busy nodes to generate excessive amounts of usually redundant data. Periodic database updates and daily aggregation render PlanetFlow data adequate for auditing purposes, but unsuitable for certain types of real-time decision-making.

It is likely that the CPU overhead of PlanetFlow would increase with higher throughput rates, but because of flow aggregation, storage overhead would not. The storage requirement is directly related to the number of individual flows generated by a node. Mapping experiments that contact a large number of peers require the greatest amount of storage to audit, but higher link speeds would not necessarily increase the number of peers contacted in such experiments.

3.2 Flow Database

Previous versions of PlanetFlow used flat text file to store flow information, which made it difficult to perform any but the most simple queries on the auditing data. Even these queries could take several minutes to complete. To reduce query times and improve the usability of the interface, the current implementation now uses the open-source database MySQL to store flow information. MySQL is a fast, general-purpose database that, like all SQL servers, supports a standard and flexible query language. MySQL data files are particularly easy to archive and manage.

Two indices into the database optimize insertion operations and common-case queries. The first index, by flow, ensures that each flow in the database is unique and reduces insertion time. The second index, by IP destination address, speeds the most common queries when the destination is known. Using indices in a relational database to access only a few fields of data, does introduce significant storage overhead. Since the indices duplicate nearly all of the information in the database, storage requirements are increased by 150% or more. A busy node can generate 50 MB of compressed MySQL data and index files per day; an average node generates about 15 MB per day. Previous versions of PlanetFlow that stored flow data in a flat text file generated only 5-10 MB of compressed data per day, but could occasionally end up generating more because they did not aggregate flows.

Ease of management was another key reason that MySQL was chosen to implement the flow database. Because MySQL tables are just files and MySQL databases are just directories, flow data can be made immediately available on the archive server by simply copying the raw table files to it with a tool such as rsync [5].

Other database systems cannot merge tables or databases together without some amount of overhead. Using MySQL enables PlanetFlow data to be centrally aggregated and archived with no import overhead, improvingthe scalability of the system.

3.3 Query Interface

A Web interface to PlanetFlow runs on the default HTTP port of every PlanetLab node. The first instinct of many complainants is to type the IP address of an unknown peer into their Web browsers. The Web interface asks complainants to search the database for the offending traffic that they received, and to report their concerns directly to the maintainers of the service that generated it. In most cases, complaints can be quickly resolved through a simple explanation of the service, without the intervention of PlanetLab Support.

Most PlanetFlow queries now take seconds rather than minutes to return, and the Web interface has become highly usable as a result. If the number of complaints led and ultimately resolved is any indication of success of the new implementation, then more than half of the approximately 100 traffic complaints received through the Web interface from May 2004 to July 2005 were led since April 2005, when the current version of PlanetFlow was deployed. Complaints received via e-mail rather than through the Web interface are almost always resolved by PlanetLab Support by accessing theWeb interface and putting complainants in direct contact with the responsible researchers. Much of the burden of resolving traffic complaints has thus been shifted to the actual users of PlanetLab who generate traffic.

The Web interface also offers an aggregate summary of all flows sent by each active slice. This information may be used to estimate the bandwidth requirements of each slice, as well as to identify the most active users of a node.

A stand-alone administrative interface to the database called pfgrep is available to PlanetLab administrators to perform text-based searches similar to those that can be performed through the Web interface. In many cases, pfgrep is faster to use than the Web interface.

A streaming sensor interface [6] is available for other slices to subscribe to at:

```
http://localhost/flows
```

The interface provides flow updates every five minutes in an easily parseable format.

3.4 Archive Server

All PlanetFlow data is permanently archived on a central server so that complaints about traffic can be resolved even if the node that generated it is unreachable. The archive server runs the same Web interface that each node does, and simply extends each query over all of its archived databases, another feature of MySQL that makes it highly suitable for PlanetFlow. The archive server contacts every node periodically to download new data.

The archive server currently stores five weeks' worth of data from about 500 active nodes. To keep the consolidated database size manageable, the archive server periodically compresses and of loads old databases onto DVD. Archive DVDs, and thus all PlanetFlow records, are kept forever and may be queried using pfgrep or other applications that can parse MySQL records. Currently, the archive server is outfitted with 500 GB of storage and keeps about 300 GB of data available for querying at any one time.

3.5 Limitations

PlanetFlow has proven very useful for resolving complaints caused by certain kinds of traffic, but it of course has its limitations. For instance, PlanetFlow cannot be used to completely address problems that arise from objectionable content. If someone determines that a PlanetLab node served copyrighted, illegal, or otherwise objectionable content to a specific set of hosts, PlanetFlow can be used to identify the service which did so. However, it cannot be used to inspect or recover the contents of the traffic, since it logs only flow data. PlanetLab considers itself a common carrier and so does not inspect or censor content. When asked to, the responsible researchers must satisfy site administrators or other authorities that their service is not being used for illegal purposes.

Complaints about misuse of a service—for example, using a CDN to propagate spam or commit fraud—present a similar problem. PlanetFlow can be used to track such complaints back to the service, but additional information is usually required to identify the party that abused it. Many services keep their own application specific logs for this purpose. We could make it easier for services to correlate their logs with PlanetFlow by enabling them to annotate their flows. For instance, a CDN could tag each connection with the IP address of the requester by making a special setsockopt() call. PlanetFlow could retrieve the tag from the meta-data of each packet and store it in the flow logs as additional, application-specific information. If privacy of this information is of importance, it could be made accessible only to PlanetLab administrators.

The primary goal of PlanetFlow's web interface was to minimize the responsibility and involvement of site administrators in complaint resolution. The web interface has met with mixed success. On one hand, given the rise in the number of complaints filed through it, it is evident that people do use it. Furthermore, the interface is so easy to use, we have begun to receive almost frivolous complaints. On the other hand, many complaints regarding PlanetLab are still sent to abuse aliases at hosting sites, which consumes both their support resources and, often, patience. Given enough complaints, site administrators may consider hosting PlanetLab nodes to be too much trouble, unless their researchers continuously emphasize the value of PlanetLab to their institutions.

4 Monitoring

CoMon is an extensible monitoring system designed specifically to monitor activity on PlanetLab [7]. Reporting is done via a Web interface that provides the ability to sort data, run moderately complex selection queries, and show graphs of recent history. In terms of the data collected, CoMon fills a monitoring niche between the general-purpose, passive reporting tools provided by the operating system and application-specific reporting systems. CoMon serves multiple roles, including:

- **"Sufficient" monitoring** -- For many PlanetLab users, CoMon provides enough monitoring to eliminate or reduce the need for experiment-specific monitoring. Especially for projects where development resources are tight, and the most important concerns are processes running out of control, CoMon's reporting is often sufficient to track how the experiment is consuming resources globally.

- **Community-aided problem identification** -- For more attentive PlanetLab researchers, CoMon often provides enough information to determine why some PlanetLab nodes may be acting strangely, or which other experiments may be impacting their experiments. This kind of information has been extremely valuable for preserving the privacy of experiments while still enabling the overall community to help the PlanetLab operations staff in identifying problems.

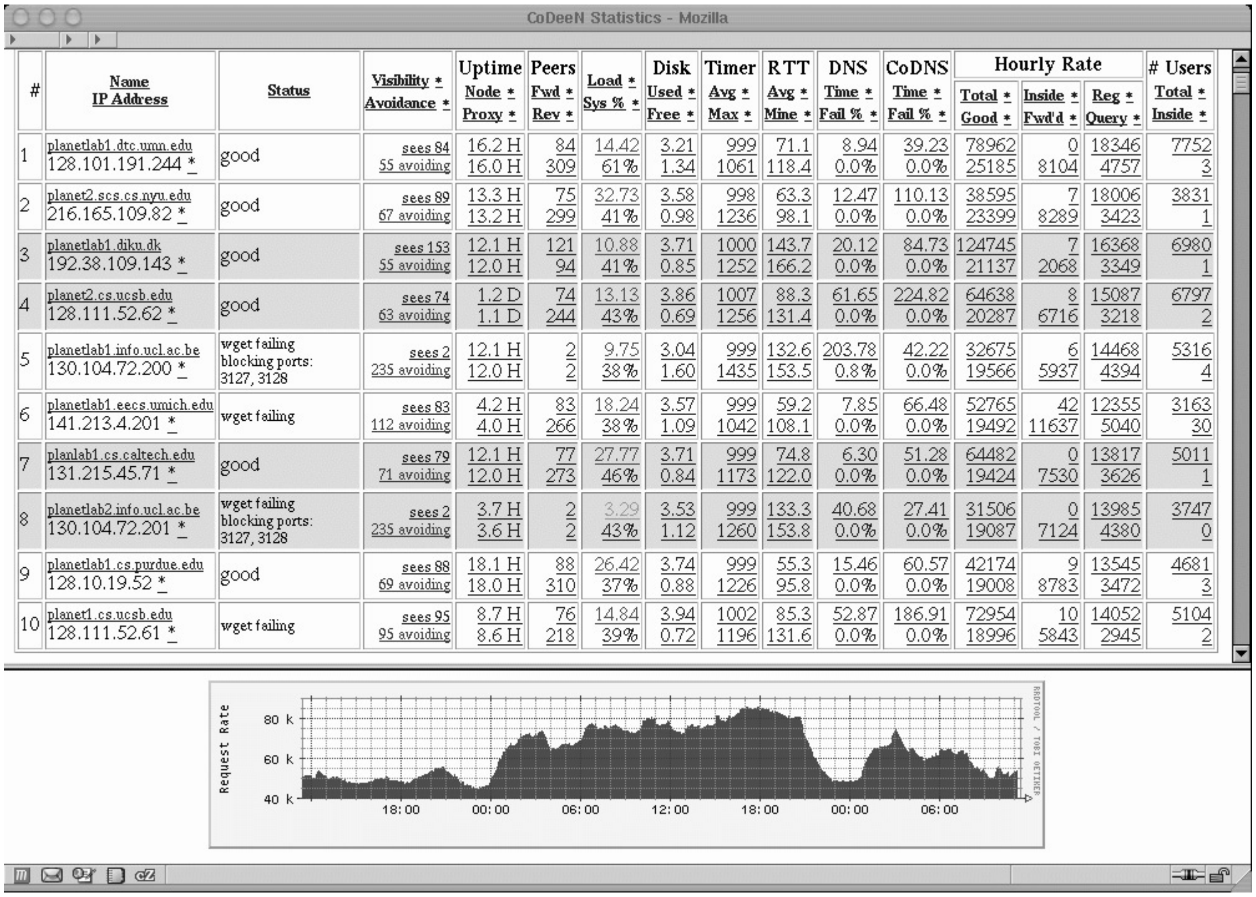
- **Login troubleshooting** -- The CoTest tool uses CoMon-provided data to help users determine why their logins may be failing. It tests a variety of common conditions that often cause login failures, allowing users to provide more information when reporting failures to PlanetLab support.

- **Node selection** -- A relatively recent feature of CoMon allows users to specify arbitrary queries to select a set of nodes. These queries can also output results in various text formats, enabling the use of CoMon in scripting systems. This approach is useful both for identifying problems as well as for selecting nodes when deploying new (short-lived) experiments.

Over time, CoMon has grown across many dimensions, from the number of tests it runs, to the variety of output data it can generate, to the types of problems it can help find. Its mission has also grown in that time, from a purely passive Web-oriented system that only monitored node-oriented data, to one that now maintains a history of resource usage information of every experiment on every PlanetLab node.

4.1 Web Interface

A screen shot of CoMon's monitoring system is given in Figure 4, and shows a number of metrics as columns, with PlanetLab nodes as rows. The sort order can be changed by clicking on a column name, and clicking on any cell value displays a historical graph for that value in the bottom window. Cell values are color-coded to indicate unusually high or low values.



#	Name	Status	Visibility *	Uptime	Peers	Load *	Disk	Timer	RTT	DNS	CoDNS	Hourly Rate			# Users
	IP Address		Avoidance *	Node *	Fwd *	Sys % *	Used *	Avg *	Avg *	Time *	Time *	Total *	Inside *	Reg *	Total *
				Proxy *	Rev *		Free *	Max *	Mine *	Fail % *	Fail % *	Good *	Fwd'd *	Query *	Inside *
1	planetlab1.dtc.umn.edu	good	sees 84	16.2 H	84	14.42	3.21	999	71.1	8.94	39.23	78962	0	18346	7752
	128.101.191.244 *		55 avoiding	16.0 H	309	61%	1.34	1061	118.4	0.0%	0.0%	25185	8104	4757	3
2	planet2.scs.cs.nyu.edu	good	sees 89	13.3 H	75	32.73	3.58	998	63.3	12.47	110.13	38595	7	18006	3831
	216.165.109.82 *		67 avoiding	13.2 H	299	41%	0.98	1236	98.1	0.0%	0.0%	23399	8289	3423	1
3	planetlab1.diku.dk	good	sees 153	12.1 H	121	10.88	3.71	1000	143.7	20.12	84.73	124745	7	16368	6980
	192.38.109.143 *		55 avoiding	12.0 H	94	41%	0.85	1252	166.2	0.0%	0.0%	21137	2068	3349	1
4	planet2.cs.ucsb.edu	good	sees 74	1.2 D	74	13.13	3.86	1007	88.3	61.65	224.82	64638	8	15087	6797
	128.111.52.62 *		63 avoiding	1.1 D	244	43%	0.69	1256	131.4	0.0%	0.0%	20287	6716	3218	2
5	planetlab1.info.ucl.ac.be	wget failing blocking ports: 3127, 3128	sees 2	12.1 H	2	9.75	3.04	999	132.6	203.78	42.22	32675	6	14468	5316
	130.104.72.200 *		235 avoiding	12.0 H	2	38%	1.60	1435	153.5	0.8%	0.0%	19566	5937	4394	4
6	planetlab1.eecs.umich.edu	wget failing	sees 83	4.2 H	83	18.24	3.57	999	59.2	7.85	66.48	52765	42	12355	3163
	141.213.4.201 *		112 avoiding	4.0 H	266	38%	1.09	1042	108.1	0.0%	0.0%	19492	11637	5040	30
7	planlab1.cs.caltech.edu	good	sees 79	12.1 H	77	27.77	3.71	999	74.8	6.30	51.28	64482	0	13817	5011
	131.215.45.71 *		71 avoiding	12.0 H	273	46%	0.84	1173	122.0	0.0%	0.0%	19424	7530	3626	1
8	planetlab2.info.ucl.ac.be	wget failing blocking ports: 3127, 3128	sees 2	3.7 H	2	3.29	3.53	999	133.3	40.68	27.41	31506	0	13985	3747
	130.104.72.201 *		235 avoiding	3.6 H	2	43%	1.12	1260	153.8	0.0%	0.0%	19087	7124	4380	0
9	planetlab1.cs.purdue.edu	good	sees 88	18.1 H	88	26.42	3.74	999	55.3	15.46	60.57	42174	9	13545	4681
	128.10.19.52 *		69 avoiding	18.0 H	310	37%	0.88	1226	95.8	0.0%	0.0%	19008	8783	3472	3
10	planet1.cs.ucsb.edu	wget failing	sees 95	8.7 H	76	14.84	3.94	1002	85.3	52.87	186.91	72954	10	14052	5104
	128.111.52.61 *		95 avoiding	8.6 H	218	39%	0.72	1196	131.6	0.0%	0.0%	18996	5843	2945	2

Figure 4. Screenshot of CoMon: Rows are PlanetLab nodes, and most columns contain two data values, with the column headings indicating the metrics being displayed. The bottom of the window shows a 2-day history of any cell value. Other windows can show histories on all nodes for a given metric, or histories of all metrics for a given node.

CoMon also displays information about slice behavior. This can be viewed as a privacy compromise, where anyone can see resource consumption details about other slices, without having access to the information on a per-process basis. This approach also allows the entire PlanetLab community to get involved in policing resource usage, and to help spot problems on nodes. By making resource usage public information, it was hoped that communal pressure could be exerted on those slices that were behaving poorly. PlanetLab's support staff can obviously get involved at any point, but the goal was to reduce the need for them to have to identify the problem from scratch every time.

Additionally, by expanding the amount of information available about node health, CoMon and other services can be developed in their own slices. This approach is preferable to running everything inside the root context, because not only does it provide the kind of isolation that running unprivileged processes provides, but it also motivates a more decentralized design. As more infrastructure gets built by the community, the fewer feature requests need to be handled by PlanetLab's central developers.

4.2 Per-Node Daemons

CoMon relies on two daemons running on each node, which provide node-centric details as well as slice-centric details. Both daemons accept HTTP requests and respond with HTTP responses, to allow them to be accessed from Web browsers in addition to being used with automated systems. Both daemons provide responses in human-readable text rather than in binary, mostly to allow easy extensibility.

Slice-centric Daemon

Of these two daemons, the information presented by the slice-centric daemon is much simpler -- it reports the aggregate resources used by all processes within each slice. It reports 11 metrics -- the transmit and receive bandwidths for the past 1, 5, and 15 minutes, the physical/virtual memory consumption, the CPU and memory usage, and the number of ports in use. This daemon is an extension of the slicestat sensor server originally developed by Brent Chun.

This daemon gets most of its data from the slicestat sensor operating on each node, and formats it in a manner similar to the top monitoring tool. The daemon has a main event-driven process that responds to client requests, and a second helper process that communicates with the slicestat sensor in order to get the data it needs. All outstanding requests can be satisfied by a single response from the slicestat sensor, since the same data is presented to all clients. A sample of this sensor output is shown in Figure 5.

CTX	TX1	TX15	RX1	RX15	#PR	PMEMMB	VMEMMB	%CPU	%MEM	NAME
516	430	933	4715	1066	5	22.0	34.2	39.3	2.2	nyu_d
610	0	0	0	0	14	22.8	41.1	25.0	2.2	arizona_stork
613	391	558	388	559	58	95.3	160.2	12.5	9.4	princeton_codeen
713	0	0	0	0	10	50.7	83.5	3.6	5.0	irb_snort
594	10	12	10	12	60	49.9	142.6	1.8	4.9	princeton_coblitz
503	1	0	0	0	23	80.0	381.1	1.8	7.9	pl_netflow
518	0	0	0	0	5	3.1	7.4	1.8	0.3	nyu_oasis
598	0	0	0	0	3	6.5	11.9	0.0	0.6	uw_ah
738	0	0	0	0	3	6.2	12.0	0.0	0.6	purdue_4
596	0	0	0	0	5	3.5	9.2	0.0	0.3	ucb_pier_2
726	10	10	51	38	133	68.7	112.1	0.0	6.6	mit_app
577	0	0	0	0	11	9.5	21.6	0.0	0.9	upenn_maoy
636	0	0	0	0	6	8.6	27.1	0.0	0.8	mit_dht
640	4	4	5	4	6	16.8	232.2	0.0	1.7	ucb_srhea
760	0	0	0	0	4	42.0	608.9	0.0	4.1	harvard_sbon_test
0	0	0	0	0	39	25.1	228.1	0.0	2.4	root

[...]

Figure 5. Excerpt from sample output from the slice-centric daemon, showing resource consumption per slice. The columns, from left to right, are context number (numeric userid for the slice), transmit and receive rates for the past 1 and 15 minutes, number of processes, physical and virtual memory consumption, overall CPU and memory utilization, and slice name. Most nodes have on the order of 50 slices running at a time.

Node-centric Daemon

The node-centric reporting currently covers 57 values, which consist of OS-provided metrics, values that are passively measured or synthesized from other sources on the node, and values that are actively measured by means of test programs running on the node. A brief summary of these metrics is provided below:

- **OS-provided** -- uptime, CPU utilization (overall and system), memory size, active memory consumption, disk size, disk space available, swap size, swap space available, date, 1 minute load, 5 minute load, swap in / out rate, and disk read / write rate.

- **Passively-measured/synthesized** -- last time ssh succeeded, last time slice-centric daemon succeeded, clock drift, number of raw ports and ICMP ports in use, number of slices in memory, number of slices using CPU, number of ports in use, number of ports in use for more than an hour, and resource hogs (which experiments are using the most CPU, memory, processes, bandwidth, and ports)

- **Actively measured** -- max / avg value reported by a 1-second timer, max / avg value for time needed to make loopback connection, whether the global file table has free entries, amount of CPU available to a spin-loop, amount of memory pressure seen by a test program, UDP and TCP failure rates for local DNS servers, last HTTP failure time, detecting presence of transparent Web proxies

```
VMStat: 2 1 148384 20916 44904 440904 1 0 [...]
CPUUse: 68 100
DNSFail: 0.0 0.0 0.0 0.0
DfDot: 14% 155.504 179.39
Date: 1133129457.852773000
Uptime: 96899.85
Loads: 5.09 3.75 3.88
Timer: 236.646000 11.412840
FdTestHist: 0x0
ServTest: 12.505000 1.617917
MemInfo: 0.987202 60.7348 14.1503
KernVer: 2.6.12
Burp: 28.5%
MemPress: 98
LastSsh: 1133128201
Test206: 206   0   0   0
SamePorts: 457 list_sameports
SnapPorts: 1469 list_portsnap
SameHog:     178 princeton_codeen
SnapHog:     632 princeton_codeen
RawPorts: 6
```

```
ICMPPorts: 35
CPUHog: 71.7 nyu_d
MemHog: 14.8 princeton_comon
TxHog: 939 nyu_d
RxHog: 567 princeton_codeen
ProcHog: 60 princeton_coblitz
NumSlices: 43 0.000000
LiveSlices: 9
```

Figure 6. Sample output from the node-centric daemon showing all measurements performed. The interpretation and display of the measured values is controlled via a configuration file on the machine that gathers the data from all of the per-node daemons.

The guiding principle for choosing metrics for inclusion is whether they can provide insight into why a researcher's experiments may be behaving strangely on a given node. This approach allows us to prune the dozens of OS-provided metrics to a more manageable set, and it also guides the development of some of the more unusual active measurements. For example, the CPU consumption of a spin loop and the observed behavior of a timer both give some insight into how much CPU PlanetLab's custom scheduler gives each experiment. Likewise, the transparent proxy test, which checks if a remote server sees the same IP address as a local client sees, is useful for experiments that contact HTTP servers, but may not be useful in non-testbed environments.

The structure of the node-centric daemon is intentionally simple – it consists of one main event-driven process that spawns a pool of helpers, as a Web server might handle different CGI programs. Each helper process is persistent, and periodically generates updated values for the quantities it measures. When a client requests data from the node-centric daemon, it receives a response that combines the most recent values from each helper. This approach leads to a large number of processes (over 100 with the current design), but most processes are waiting at any given time and have small footprints. On average, CoMon uses less than 0.5% of each node's CPU for its processing. A sample node-centric response is shown in Figure 6.

4.3 Data Gathering

While the CoMon daemons operate on each node, the bulk of CoMon data gathering and processing operates in a centralized fashion, mostly to reduce complexity and to ensure that we have a properly-provisioned machine available.

Data is collected from the per-node daemons every five minutes, and is stored in a variety of files. Most nodes (typically over 90%) respond within one second, and generally fewer than five nodes take more than 10 seconds to respond. All fetches are performed in parallel to reduce latency. The raw data, in text format, is stored in several places: (1) a file containing the most recent snapshot for all nodes, (2) appended to the end of a file containing all of the data gathered for the current day, and (3) also written into one file per live node so that the last live data is available for post-mortem node analysis.

The most basic processing of the raw data generates metafiles that can then be used by CGI programs to generate sorted HTML tables on demand. The number of these metafiles is O(slices + nodes) – the slice-centric information is shown for all the slices on each node, and for all of the nodes for a particular slice. Additionally, summaries are generated with the slice-centric information, showing the maximum, average, and total values for each of the slice-centric data items across all nodes in a slice. Only two metafiles are generated with the node-centric data -- one that contains all of the node-centric data for each node, and another that omits some of the less-important fields, such as identifying all of the resource hogs. The second metafile is intended to be used on narrower screens, to avoid horizontal scrolling.

From a scalability perspective, the processing steps discussed above are quite tolerable, since the number of individual files generated is roughly # slices + 2 * # nodes. At PlanetLab's current size and usage (approximately 600 nodes and 250 experiments), these steps generate 1500 files every 5 minutes, or 5 files per second. If we assume these files are randomly-accessed, and if we have 2 other seek-causing metadata accesses per file, this rate still yields only 15 seeks/second, which is at least a factor of 5 lower than current disks can handle.

In terms of network and disk bandwidth, the current numbers also leave plenty of headroom. CoMon currently generates 600 MB of raw data per day taking into account both the node-centric and slice-centric fetches. On average, this is less than 7.5 KB/sec. Compression can reduce this number further -- CoMon's daily logs, when compressed with *bzip2*, generally drop to 10% of their raw size.

4.4 Scaling Processing

The scalability issues for CoMon arise from the desire to present graphical two-day histories for any value shown in the tables. We use the popular RRDTool [8], which is designed to efficiently maintain time-series databases for monitoring applications, from which monitoring graphs can be easily generated. We have one separate database file for each row of any table, which provides us the best practical balance between efficiency and flexibility. This approach allows us to add nodes/slices to CoMon without having to reformat the database files. Adding more metrics per row, however, does require updating the database files. While RRDTool has recently added support for transferring values from one database to another, we currently just erase old database files when the format changes.

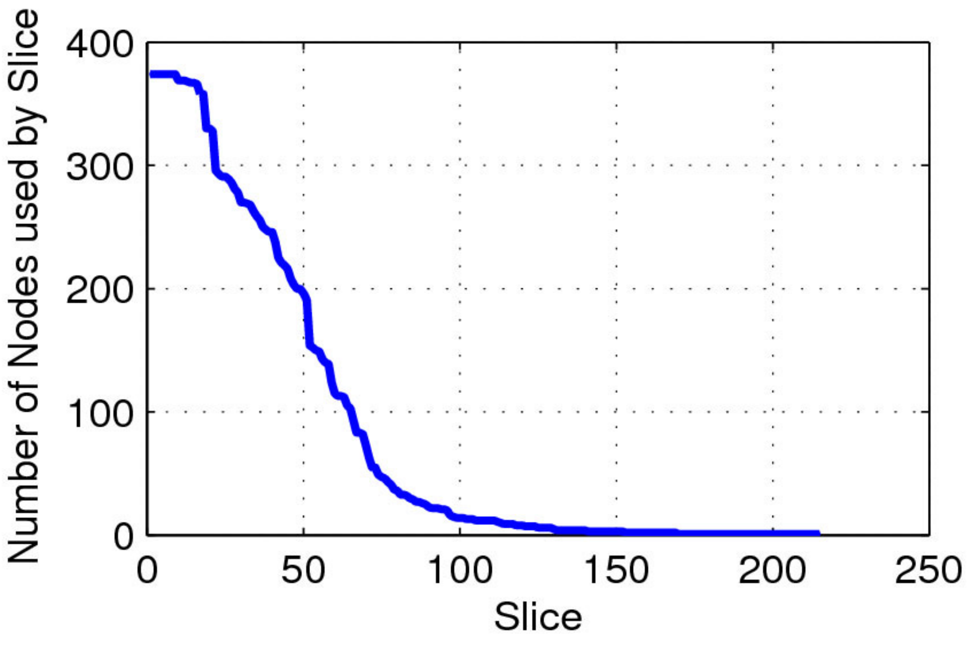


Figure 7. Snapshot of # of nodes used by each slice on 11/28/2005

The total number of database files is then O(# nodes + 3 * # slices + # slivers), where a sliver is a slice instantiated on a node. In theory, the maximum number of slivers is # slices * # nodes, but not all slices are deployed on every node. In practice, about 25 slices (out of 200-250 slices) are deployed across more than three-fourths of all nodes, while another 25 are deployed on more than half. This kind of distribution can be seen in Figure 7 which shows a fairly typical snapshot of the number of nodes used by each slice. In total, of the 100,000 possible slivers that could typically exist, about 20,000-22,000 exist at any given time in the recent past. A longer history of the number of total slivers is shown in Figure 8. The dip at 120 days corresponds to a major PlanetLab upgrade that required significant downtime. The drops near 170 days were due to problems encountered in the process of testing a kernel upgrade. Other smaller bumps are typically the activity near conference submission dates.

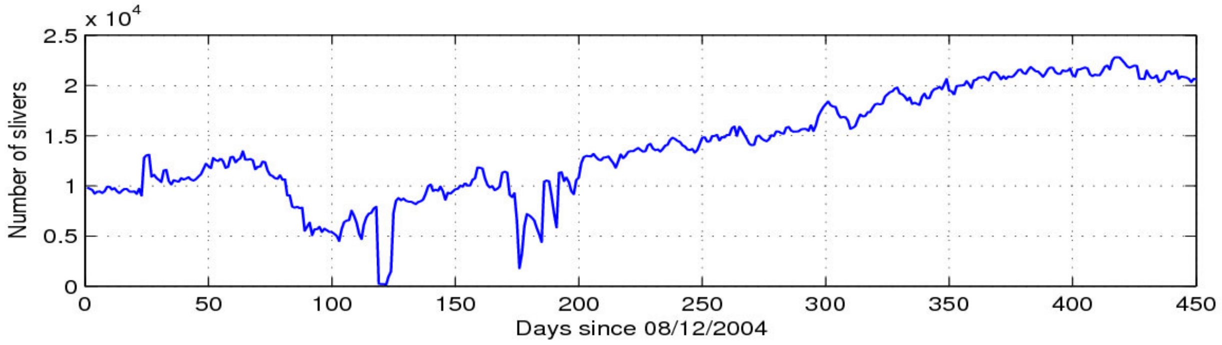


Figure 8. Daily count of total number of slivers

If these databases are to be updated every 5 minutes, the number of seeks involved exceeds the rate that can be achieved by current disks. Assuming an optimistic number of 3 seeks per file, this workload would require 3 high-performance disks running continuously at maximum speed just to maintain the current update rate. In practice, we seem to require more than 3 seeks per file (understandable, given the number of directories involved), and other activities also use the disk. Even if this seek rate was achievable, it would be dangerous from a design standpoint to be so close to the hardware limits at all times. Not only would it limit scalability (any new slices might cause overload), but even a slight miscalculation could cause each update to run slightly above the time allotted, cumulatively increasing the overall load on the system and eventually causing collapse.

The approach we take to this problem stems from the observation that higher disk bandwidth is easier to achieve than higher seek rates, and that preserving disk locality via batching can exploit the higher bandwidth. Batching also naturally implies delaying some operations, possibly causing longer delays, so a naive approach can result in lower perceived performance. To reduce the perceived impact of batching while still using it to address the disk seek problem, we split the data processing into several steps, and batch independently at each step. In this manner, we can control how much delay is introduced in the various tables/graphs, and we can control the resource usage of the various components. The four processing steps are described below.

- **Gather Data** -- Every 5 minutes, gather data from the slice-centric daemons. Write all data sequentially into a snapshot file of the current state, write another copy into an "update" file, and also append it to the history file for the current day. If no other instance of the program is currently writing tables, write the per-node tables, the per-slice tables, and the tables for the maximum/average/total resources consumed by each slice.

- **Split** -- This program checks for the " update" files generated by the "gather data" process. It takes the monolithic update file and splits it into a separate file per slice. It repeats the process for every monolithic update file it finds.

- **Update Stats** -- Using the per-slice update files, this process updates the RRD databases containing the per-slice statistics for maximum, average, and total resource consumption. Only one instance of this process runs at a time, and processing is comparatively CPU intensive. On a fast dual-processor node, this step takes only 10-20 seconds, but can take minutes on a more loaded uniprocessor. With these times and the dependent processing steps, the *graphs* for the per-slice statistics tend to lag no more than 10-15 minutes behind the current time. The tables, in contrast, are updated in real time with the latest gathered data.

- **Update Slivers** -- Takes the per-slice update file and splits it by node to generate per-sliver files. It then takes all of the per-sliver files and updates the corresponding RRD databases. On a fast dual-processor machine with two hard drives in a RAID0 configuration, this step takes 150-250 seconds. On a slower uniprocessor, this step can take over 2000 seconds.

The benefit of this partitioned approach is that we can optimize (to the extent possible) the amount of lag seen by users.[1] The most commonly used portions, the node-centric and slice-centric tables are always up to date with respect to the most recent data gathered. The graphs for the node metrics are also recent, and only on slow systems will the slice summary graphs be delayed. Finally, the graphs for the sliver metrics, which consume the most resources and are the least-frequently used portion of the system, can experience the most lag.

An interesting additional benefit of this approach has been the ability to tune the lag to reduce heat-induced stress on the disks. We have found that 1U rack-mounted systems often cannot efficiently dissipate the heat from drives running at full seek capacity, and that the overheating drives cause regular system lock-ups ranging from several seconds to over a minute at a time. By adding delay loops into the processes that update the RRD databases for the slice summaries and slivers, we can reduce the heat buildup in these systems for the cost of some extra lag in the graphs.

4.5 Node Selection

As CoMon's utility increased, so did many users' desires to do something with CoMon's data, which led to the development of node selection support. Originally, CoMon users had several less-than-appealing options when trying to incorporate CoMon into other systems: they could screen-scrape the HTML tables, they could directly query the sensors, or they could copy-and-paste data from their browsers. Some mechanisms was needed to automate this process and leverage the support CoMon already had developed. This part of the design had several guiding principles: keep the process as simple as possible, make it easily accessible to people familiar with C, allow it to be easily scriptable, and reduce the "buy-in" necessary for users to adopt the solution.

We ultimately chose to extend the table metafile that CoMon generates to be used with the CGI application that generates all CoMon tables. By doing so, we add virtually no overhead to the process of generating regular CoMon pages, making the infrastructure change relatively transparent. We pass the selection criteria in the URL itself, making it easy to be used both with browsers as well as standalone downloading tools such as *curl* and *wget*, popular in scripts. We opted for a simple, C-like syntax (with matching operator precedence) for specifying nodes, since it was compact and would be easily recognizable to most of our target audience. While these choices are not ultimately as expressive as other options, we felt that they presented a good balance between simplicity and utility.

When the node selection support is used, each row is tested against the selection criteria, and if the statement evaluates positively, the node is included in the display. Column names are treated like variables, and the standard set of comparison operations are provided. To get a sense of how this selection works, the following text can be added to the node-centric table's URL to select lightly-loaded nodes only:

[1] To clarify, we use the term lag to define the time difference between the most recent value in the system versus the current wall-clock time.

```
select='resptime > 0 && 1minload < 5 && liveslices <= 5'
```

Specifying a nonzero response time selects only live nodes, while specifying a 1 minute load of less than 5 and fewer than 5 slices actively running further restricts it to only those nodes that have relatively little work competing for the CPU [2]. The selection criteria can also be used to identify problematic nodes, such as this example:

```
select='drift > 1m || (dns1udp > 80 && dns2udp > 80) || (resptime > 0 && gbfree < 5) ||
sshstatus > 2h'
```

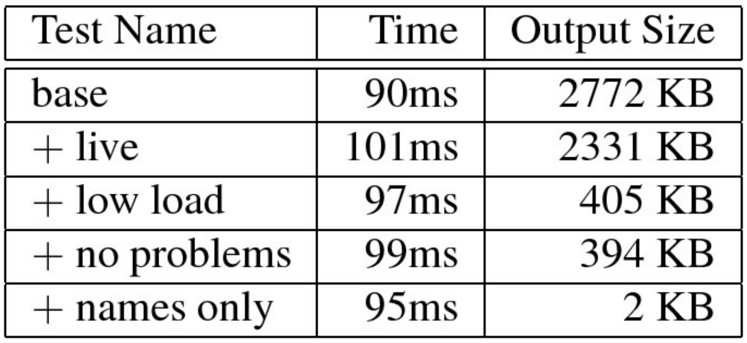
This selects for four types of problems -- clock drifts greater than 1 minute, primary and secondary DNS failure rates above 80%, live nodes that are running short on free disk space, and nodes where the SSH daemon has been refusing connections for over 2 hours. This kind of selection statement would be useful to groups like PlanetLab's operations staff in order to identify problematic nodes, and perhaps show them on a Web page. However, when we combine the two selection categories together, we can find nodes with low load and without problems:

```
select='(resptime > 0 && 1minload < 5 && liveslices <=$ 5) && ((drift > 1m || (dns1udp >
80 && dns2udp >$ 80) || gbfree < 5 || sshstatus > 2h) == 0)'
```

Users can further specify the directive *format=nameonly* in the URL to specify that instead of generating HTML tables as output, CoMon should produce only a list of node names in text format. This kind of URL is then easily amenable to use in scripts. The additional code to support these features consists of a simple parser and evaluator, and some additional support when generating the table metafiles. In total, this node selection code amounts to less than 300 additional semicolon-lines added to the system.

The computational cost of this support is relatively modest, and is actually a net benefit for the interactive user, since the extra latency in performing the selection is almost always recovered by the lower time required by the browser to render fewer rows. Some sample times measured at the server are provided in the table in Figure 9 for the types of selection statements discussed above. Note that while using the node selection support increases runtime, more complicated statements that reduce the amount of data produced can take less time than simpler statements.

[2] Since PlanetLab is a shared infrastructure, higher load levels are not uncommon



Test Name	Time	Output Size
base	90ms	2772 KB
+ live	101ms	2331 KB
+ low load	97ms	405 KB
+ no problems	99ms	394 KB
+ names only	95ms	2 KB

Figure 9. Times and output sizes resulting from various selection options.

References

[1] H. Krawczyk and M. Bellare and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (February 1997).

[2] M. Huang, A. Bavier, L. Peterson. PlanetFlow: maintaining accountability for network services. Operating Systems Review, Jan 2006.

[3] S. Muir, M. Fiuczynski, L. Peterson, J. Cappos, and J. Hartman. Proper: Privileged Operations in a Virtualised System Environment. In Proc. USENIX ’05, Anaheim, CA, Apr. 2005.

[4] N. Horman. Understanding and programming with netlink sockets. *http://people.redhat.com/nhorman/papers/netlink.pdf*

[5] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, 1997.

[6] T. Roscoe, L. Peterson, S. Karlin, and M.Wawrzoniak. A Simple Common Sensor Interface for PlanetLab. Planet-Lab Design Note PDN 03-010, May 2003.

[7] K. Park, V. Pai. CoMon: a mostly-scalable monitoring system for PlanetLab. Operating Systems Review, Jan 2006.

[8] RRDTool. *http://people.ee.ethz.ch/”oetiker/webtools/rrdtool*