Tracking Dynamic Host and Application Metrics at Scale

Monitoring with

Ganglia

Matt Massie,
Bernard Li,
Brad Nicholes,
& Vladimir Vuksan

O'REILLY®

Monitoring with Ganglia

Matt Massie, Bernard Li, Brad Nicholes, and Vladimir Vuksan

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Monitoring with Ganglia
by Matt Massie, Bernard Li, Brad Nicholes, and Vladimir Vuksan

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Kara Ebrahim
Copyeditor: Nancy Wolfe Kotary
Proofreader: Kara Ebrahim
Indexer: Ellen Troutman-Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Kara Ebrahim

November 2012: First Edition.

Revision History for the First Edition:
2012-11-7 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449329709* for release details.

ISBN: 978-1-449-32970-9

[LSI]

1352302880

Table of Contents

Preface

In 1999, I packed everything I owned into my car for a cross-country trip to begin my new job as Staff Researcher at the University of California, Berkeley Computer Science Department. It was an optimistic time in my life and the country in general. The economy was well into the dot-com boom and still a few years away from the dot-com bust. Private investors were still happily throwing money at any company whose name started with an "e-" and ended with ".com".

The National Science Foundation (NSF) was also funding ambitious digital projects like the National Partnership for Advanced Computing Infrastructure (NPACI). The goal of NPACI was to advance science by creating a pervasive national computational infrastructure called, at the time, "the Grid." Berkeley was one of dozens of universities and affiliated government labs committed to connecting and sharing their computational and storage resources.

When I arrived at Berkeley, the Network of Workstations (NOW) project was just coming to a close. The NOW team had clustered together Sun workstations using Myrinet switches and specialized software to win RSA key-cracking challenges and break a number of sort benchmark records. The success of NOW led to a following project, the Millennium Project, that aimed to support even larger clusters built on x86 hardware and distributed across the Berkeley campus.

Ganglia exists today because of the generous support by the NSF for the NPACI project and the Millennium Project. Long-term investments in science and education benefit us all; in that spirit, all proceeds from the sales of this book will be donated to Scholarship America, a charity that to date has helped 1.7 million students follow their dreams of going to college.

Of course, the real story lies in the people behind the projects—people such as Berkeley Professor David Culler, who had the vision of building powerful clusters out of commodity hardware long before it was common industry practice. David Culler's cluster research attracted talented graduated students, including Brent Chun and Matt Welsh, as well as world-class technical staff such as Eric Fraser and Albert Goto. Ganglia's use of a lightweight multicast listen/announce protocol was influenced by Brent Chun's early work building a scalable execution environment for clusters. Brent also helped

me write an academic paper on Ganglia[1] and asked for only a case of Red Bull in return. I delivered. Matt Welsh is well known for his contributions to the Linux community and his expertise was invaluable to the broader teams and to me personally. Eric Fraser was the ideal Millennium team lead who was able to attend meetings, balance competing priorities, and keep the team focused while still somehow finding time to make significant technical contributions. It was during a "brainstorming" (pun intended) session that Eric came up with the name "Ganglia." Albert Goto developed an automated installation system that made it easy to spin up large clusters with specific software profiles in minutes. His software allowed me to easily deploy and test Ganglia on large clusters and definitely contributed to the speed and quality of Ganglia development.

I consider myself very lucky to have worked with so many talented professors, students, and staff at Berkeley.

I spent five years at Berkeley, and my early work was split between NPACI and Millennium. Looking back, I see how that split contributed to the way I designed and implemented Ganglia. NPACI was Grid-oriented and focused on monitoring clusters scattered around the United States; Millennium was focused on scaling software to handle larger and larger clusters. The Ganglia Meta Daemon (gmetad)—with its hierarchical delegation model and TCP/XML data exchange—is ideal for Grids. I should mention here that Federico Sacerdoti was heavily involved in the implementation of gmetad and wrote a nice academic paper[2] highlighting the strength of its design. On the other hand, the Ganglia Monitoring Daemon (gmond)—with its lightweight messaging and UDP/XDR data exchange—is ideal for large clusters. The components of Ganglia complement each other to deliver a scalable monitoring system that can handle a variety of deployment scenarios.

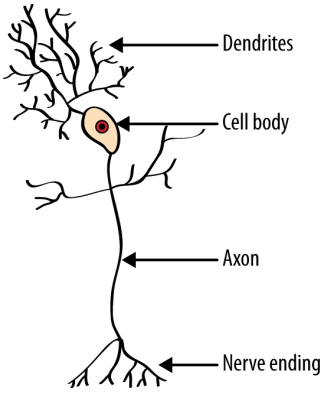
In 2000, I open-sourced Ganglia and hosted the project from a Berkeley website. You can still see the original website today using the Internet Archive's Wayback Machine. The first version of Ganglia, written completely in C, was released on January 9, 2001, as version 1.0-2. For fun, I just downloaded 1.0-2 and, with a little tweaking, was able to get it running inside a CentOS 5.8 VM on my laptop.

I'd like to take you on a quick tour of Ganglia as it existed *over 11 years ago*!

Ganglia 1.0-2 required you to deploy a daemon process, called a *dendrite*, on every machine in your cluster. The dendrite would send periodic heartbeats as well as publish any significant `/proc` metric changes on a common multicast channel. To collect the `dendrite` updates, you deployed a single instance of a daemon process, called an `axon`,

1. Massie, Matthew, Brent Chun, and David Culler. *The Ganglia Distributed Monitoring System: Design, Implementation, and Experience*. Parallel Computing, 2004. 0167-8191.
2. Sacerdoti, Federico, Mason Katz, Matthew Massie, and David Culler. *Wide Area Cluster Monitoring with Ganglia*. Cluster Computing, December 2003.

that indexed the metrics in memory and answered queries from a command-line utility named `ganglia`.



If you ran `ganglia` without any options, it would output the following help:

```
 $ ganglia

GANGLIA SYNTAX

   ganglia [+,-]token [[+,-]token]...[[+,-]token] [number of nodes]

   modifiers
      + sort ascending   (default)
      - sort descending

   tokens
      cpu_num cpu_speed cpu_user cpu_nice cpu_system
      cpu_idle cpu_aidle load_one load_five load_fifteen
      proc_run proc_total rexec_up ganglia_up mem_total
      mem_free mem_shared mem_buffers mem_cached swap_total
      swap_free

   number of nodes
      the default is all the nodes in the cluster or GANGLIA_MAX

   environment variables
      GANGLIA_MAX   maximum number of hosts to return
                    (can be overidden by command line)

EXAMPLES

prompt> ganglia  -cpu_num
   would list all (or GANGLIA_MAX) nodes in ascending order by number of cpus

prompt> ganglia -cpu_num 10
   would list 10 nodes in descending order by number of cpus

prompt> ganglia -cpu_user -mem_free 25
```

```
would list 25 nodes sorted by cpu user descending then by memory free ascending
(i.e., 25 machines with the least cpu user load and most memory available)
```

As you can see from the help page, the first version of `ganglia` allowed you to query and sort by 21 different system metrics right out of the box. Now you know why Ganglia metric names look so much like command-line arguments (e.g., `cpu_num`, `mem_total`). At one time, they were!

The output of the `ganglia` command made it very easy to embed it inside of scripts. For example, the output from Example P-1 could be used to autogenerate an MPI machine file that contained the least-loaded machines in the cluster for load-balancing MPI jobs. Ganglia also automatically removed hosts from the list that had stopped sending heartbeats to keep from scheduling jobs on dead machines.

Example P-1. Retrieve the 10 machines with the least load

```
$ ganglia -load_one 10
hpc0991  0.10
hpc0192  0.10
hpc0381  0.07
hpc0221  0.06
hpc0339  0.06
hpc0812  0.02
hpc0042  0.01
hpc0762  0.01
hpc0941  0.00
hpc0552  0.00
```

Ganglia 1.0-2 had a simple UI written in PHP 3 that would query an `axon` and present the response as a dynamic graph of aggregate cluster CPU and memory utilization as well as the requested metrics in tabular format. The UI allowed for filtering by hostname and could limit the total number of hosts displayed.

Ganglia has come a very long way in the last 11 years! As you read this book, you'll see just how far the project has come.

- Ganglia 1.0 ran only on Linux, whereas Ganglia today runs on dozens of platforms.
- Ganglia 1.0 had no time-series support, whereas Ganglia today leverages the power of Tobi Oetiker's RRDtool or Graphite to provide historical views of data at granularities from minutes to years.
- Ganglia 1.0 had only a basic web interface, whereas Ganglia today has a rich web UI (see Figure P-1) with customizable views, mobile support, live dashboards, and much more.
- Ganglia 1.0 was not extensible, whereas Ganglia today can publish custom metrics via Python and C modules or a simple command-line tool.
- Ganglia 1.0 could only be used for monitoring a single cluster, whereas Ganglia today can been used to monitor hundreds of clusters distributed around the globe.

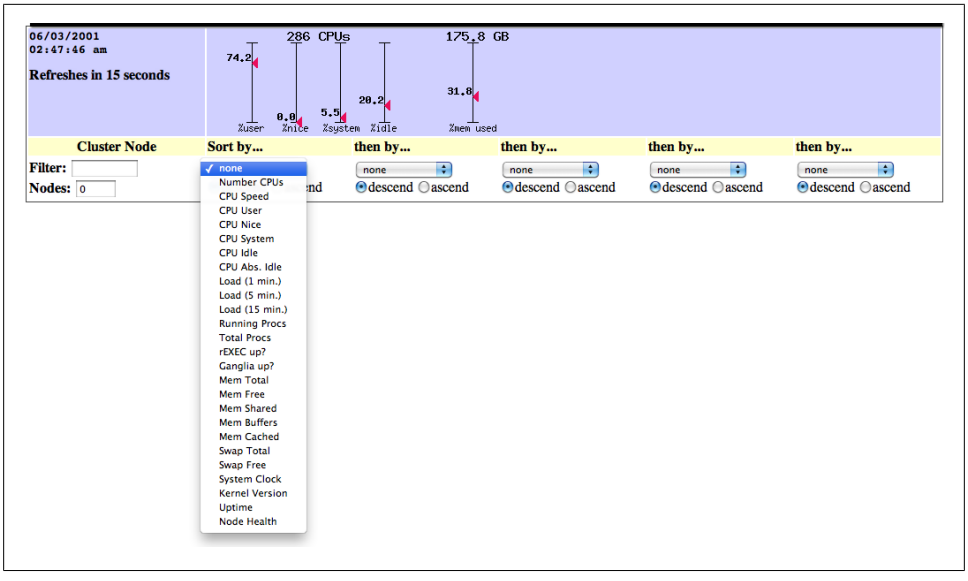


Figure P-1. The first Ganglia web UI

I just checked our download stats and Ganglia has been downloaded more than 880,000 times from our core website. When you consider all the third-party sites that distribute Ganglia packages, I'm sure the overall downloads are well north of a million!

Although the NSF and Berkeley deserve credit for getting Ganglia started, it's the generous support of the open source community that has made Ganglia what it is today. Over Ganglia's history, we've had nearly 40 active committers and hundreds of people who have submitted patches and bug reports. The authors and contributors on this book are all core contributors and power users who'll provide you with the in-depth information on the features they've either written themselves or use every day.

Reflecting on the history and success of Ganglia, I'm filled with a lot of pride and only a tiny bit of regret. I regret that it took us 11 years before we published a book about Ganglia! I'm confident that you will find this book is worth the wait. I'd like to thank Michael Loukides, Meghan Blanchette, and the awesome team at O'Reilly for making this book a reality.

—Matt Massie

Conventions Used in This Book

The following typographical conventions are used in this book:

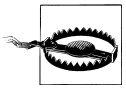
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Monitoring with Ganglia* by Matt Massie, Bernard Li, Brad Nicholes, and Vladimir Vuksan (O'Reilly). Copyright 2013 Matthew Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, 978-1-449-32970-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/ganglia*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

Introducing Ganglia

Dave Josephsen

If you're reading this, odds are you have a problem to solve. I won't presume to guess the particulars, but I'm willing to bet that the authors of this book have shared your pain at one time or another, so if you're in need of a monitoring and metrics collection engine, you've come to the right place. We created Ganglia for the same reason you've picked up this book: we had a problem to solve.

If you've looked at other monitoring tools, or have already implemented a few, you'll find that Ganglia is as powerful as it is conceptually and operationally different from any monitoring system you're likely to have previously encountered. It runs on every popular OS out there, scales easily to very large networks, and is resilient by design to node failures. In the real world, Ganglia routinely provides near real-time monitoring and performance metrics data for computer networks that are simply too large for more traditional monitoring systems to handle, and it integrates seamlessly with any traditional monitoring systems you may happen to be using.

In this chapter, we'd like to introduce you to Ganglia and help you evaluate whether it's a good fit for your environment. Because Ganglia is a product of the labor of systems guys—like you—who were trying to solve a problem, our introduction begins with a description of the environment in which Ganglia was born and the problem it was intended to solve.

It's a Problem of Scale

Say you have a lot of machines. I'm not talking a few hundred, I mean metric *oodles* of servers, stacked floor to ceiling as far as the eye can see. Servers so numerous that they put to shame swarms of locusts, outnumber the snowflakes in Siberia, and must be expressed in scientific notation, or as some multiple of Avogadro's number.

Okay, maybe not quite that numerous, but the point is, if you had lots of machines, how would you go about gathering a metric—the CPU utilization, say—from every host every 10 seconds? Assuming 20,000 hosts, for example, your monitoring system

would need to poll 2,000 hosts per second to achieve a 10-second resolution for that singular metric. It would also need to store, graph, and present that data quickly and efficiently. This is the problem domain for which Ganglia was designed; to monitor and collect massive quantities of system metrics in near real time for Large installations. *Large*. With a capital L.

Large installations are interesting because they force us to reinvent or at least reevaluate every problem we thought we'd already solved as systems administrators. The prospect of firing up rsync or kludging together some Perl is altogether different when 20,000 hosts are involved. As the machines become more numerous, we're more likely to care about the efficiency of the polling protocol, we're more likely to encounter exceptions, and we're less likely to interact directly with every machine. That's not even mentioning the quadratic curve towards infinity that describes the odds of some subset of our hosts going offline as the total number grows.

I don't mean to imply that Ganglia can't be used in smaller networks—swarms of locusts would laugh at my own puny corporate network and I couldn't live without Ganglia—but it's important to understand the design characteristics from which Ganglia was derived, because as I mentioned, Ganglia operates quite differently from other monitoring systems because of them. The most influential consideration shaping Ganglia's design is certainly the problem of scale.

Hosts ARE the Monitoring System

The problem of scale also changes how we think about systems management, sometimes in surprising or counterintuitive ways. For example, an admin over 20,000 systems is far more likely to be running a configuration management engine such as Puppet/Chef or CFEngine and will therefore have fewer qualms about host-centric configuration. The large installation administrator knows that he can make configuration changes to all of the hosts centrally. It's no big deal. Smaller installations instead tend to favor tools that minimize the necessity to configure individual hosts.

Large installation admin are rarely concerned about individual node failures. Designs that incorporate single points of failure are generally to be avoided in large application frameworks where it can be safely assumed, given the sheer amount of hardware involved, that some percentage of nodes are always going to be on the fritz. Smaller installations tend to favor monitoring tools that strictly define individual hosts centrally and alert on individual host failures. This sort of behavior quickly becomes unwieldy and annoying in larger networks.

If you think about it, the monitoring systems we're used to dealing with all work the way they do because of this "little network" mind set. This tendency to centralize and strictly define the configuration begets a central daemon that sits somewhere on the network and polls every host every so often for status. These systems are easy to use in small environments: just install the (usually bloated) agent on every system and

configure everything centrally, on the monitoring server. No per-host configuration required.

This approach, of course, won't scale. A single daemon will always be capable of polling only so many hosts, and every host that gets added to the network increases the load on the monitoring server. Large installations sometimes resort to installing several of these monitoring systems, often inventing novel ways to roll up and further centralize the data they collect. The problem is that even using roll-up schemes, a central poller can poll an individual agent only so fast, and there's only so much polling you can do before the network traffic becomes burdensome. In the real world, central pollers usually operate on the order of minutes.

Ganglia, by comparison, was born at Berkeley, in an academic, Grid-computing culture. The HPC-centric admin and engineers who designed it were used to thinking about massive, parallel applications, so even though the designers of other monitoring systems looked at tens of thousands of hosts and saw a problem, it was natural for the Berkeley engineers to see those same hosts as the solution.

Ganglia's metric collection design mimics that of any well-designed parallel application. Every individual host in the grid is an active participant, and together they cooperate, organically distributing the workload while avoiding serialization and single points of failure. The data itself is replicated and dispersed throughout the Grid without incurring a measurable load on any of the nodes. Ganglia's protocols were carefully designed, optimizing at every opportunity to reduce overhead and achieve high performance.

This cooperative design means that every node added to the network only increases Ganglia's polling capacity and that the monitoring system stops scaling only when your network stops growing. Polling is separated from data storage and presentation, both of which may also be redundant. All of this functionality is bought at the cost of a bit more per-host configuration than is employed by other, more traditional monitoring systems.

Redundancy Breeds Organization

Large installations usually include quite a bit of machine redundancy. Whether we're talking about HPC compute nodes or web, application, or database servers, the thing that makes large installations large is usually the preponderance of hosts that are working on the same problem or performing the same function. So even though there may be tens of thousands of hosts, they can be categorized into a few basic types, and a single configuration can be used on almost all hosts that have a type in common. There are also likely to be groups of hosts set aside for a specific subset of a problem or perhaps an individual customer.

Ganglia assumes that your hosts are somewhat redundant, or at least that they can be organized meaningfully into groups. Ganglia refers to a group of hosts as a "cluster,"

and it requires that at least one cluster of hosts exists. The term originally referred to HPC compute clusters, but Ganglia has no particular rules about what constitutes a cluster: hosts may be grouped by business purpose, subnet, or proximity to the Coke machine.

In the normal mode of operation, Ganglia clusters share a multicast address. This shared multicast address defines the cluster members and enables them to share information about each other. Clusters may use a unicast address instead, which is more compatible with various types of network hardware, and has performance benefits, at the cost of additional per-host configuration. If you stick with multicast, though, the entire cluster may share the same configuration file, which means that in practice Ganglia admins have to manage only as many configuration files as there are clusters.

Is Ganglia Right for You?

You now have enough of the story to evaluate Ganglia for your own needs. Ganglia should work great for you, provided that:

- You have a number of computers with general-purpose operating systems (e.g., not routers, switches, and the like) and you want near real-time performance information from them. In fact, in cooperation with the sFlow agent, Ganglia may be used to monitor network gear such as routers and switches (see Chapter 8 for more information).
- You aren't averse to the idea of maintaining a config file on all of your hosts.
- Your hosts can be (at least loosely) organized into groups.
- Your operating system and network aren't hostile to multicast and/or User Datagram Protocol (UDP).

If that sounds like your setup, then let's take a closer look at Ganglia. As depicted in Figure 1-1, Ganglia is architecturally composed of three daemons: gmond, gmetad, and gweb. Operationally, each daemon is self-contained, needing only its own configuration file to operate; each will start and run happily in the absence of the other two. Architecturally, however, the three daemons are cooperative. You need all three to make a useful installation. (Certain advanced features such as sFlow, zeromq, and Graphite support may belie the use of gmetad and/or gweb; see Chapter 3 for details.)

gmond: Big Bang in a Few Bytes

I hesitate to liken gmond to the "agent" software usually found in more traditional monitoring systems. Like the agents you may be used to, it is installed on every host you want monitored and is responsible for interacting with the host operating system to acquire interesting measurements—metrics such as CPU load and disk capacity. If

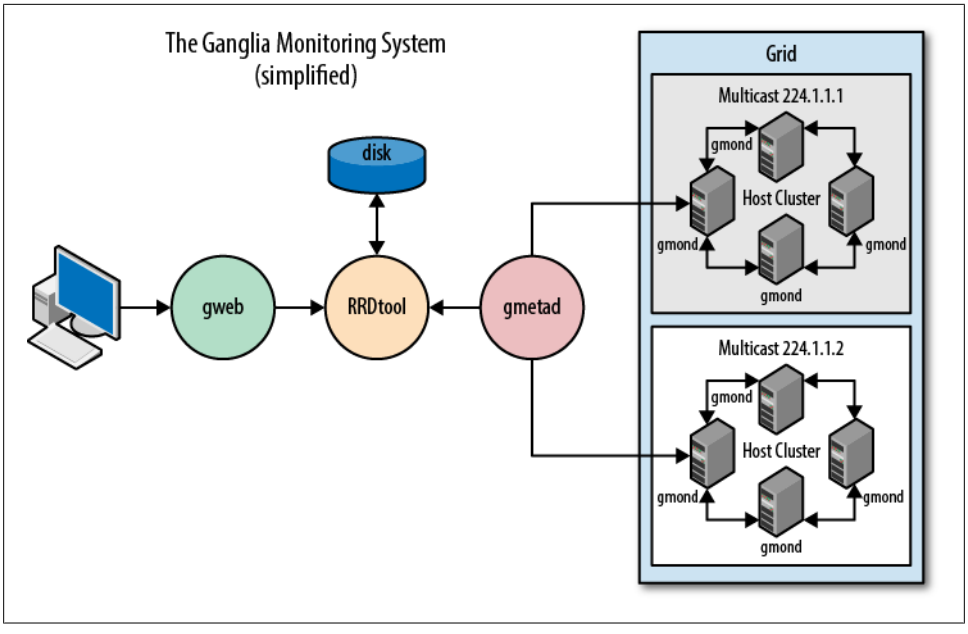


Figure 1-1. Ganglia architecture

you examine more closely its architecture, depicted in Figure 1-2, you'll probably find that the resemblance stops there.

Internally, gmond is modular in design, relying on small, operating system–specific plug-ins written in C to take measurements. On Linux, for example, the CPU plug-in queries the "proc" filesystem, whereas the same measurements are gleaned by way of the OS Management Information Base (MIB) on OpenBSD. Only the necessary plug-ins are installed at compile time, and gmond has, as a result, a modest footprint and negligible overhead compared to traditional monitoring agents. gmond comes with plug-ins for most of the metrics you'll be interested in and can be extended with plug-ins written in various languages, including C, C++, and Python to include new metrics. Further, the included gmetric tool makes it trivial to report custom metrics from your own scripts in any language. Chapter 5 contains in-depth information for those wishing to extend the metric collection capabilities of gmond.

Unlike the client-side agent software employed by other monitoring systems, gmond doesn't wait for a request from an external polling engine to take a measurement, nor does it pass the results of its measurements directly upstream to a centralized poller. Instead, gmond polls according to its own schedule, as defined by its own local configuration file. Measurements are shared with cluster peers using a simple listen/announce protocol via XDR (External Data Representation). As mentioned earlier, these announcements are multicast by default; the cluster itself is composed of hosts that share the same multicast address.

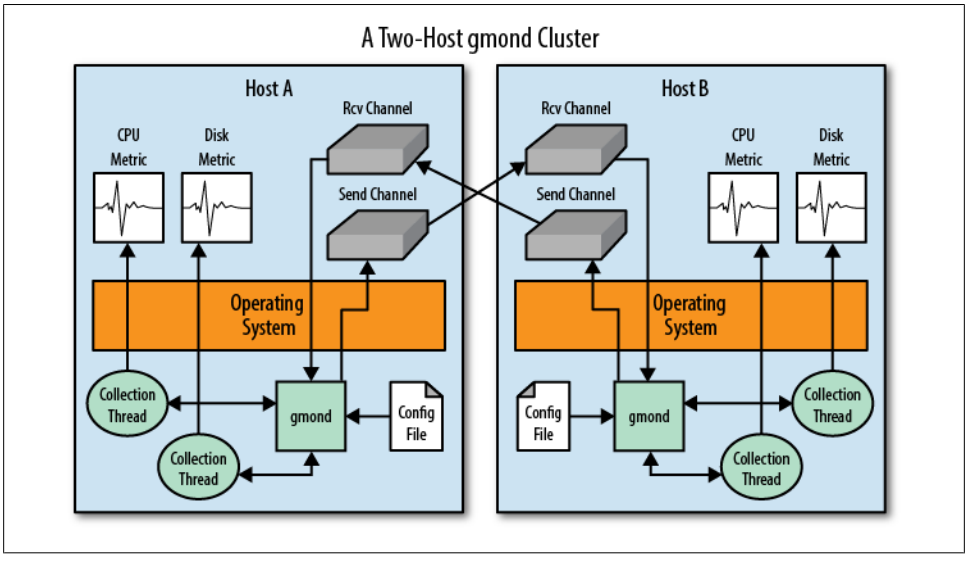


Figure 1-2. gmond architecture

Given that every gmond host multicasts metrics to its cluster peers, it follows that every gmond host must also record the metrics it receives from its peers. In fact, every node in a Ganglia cluster knows the current value of every metric recorded by every other node in the same cluster. An XML-format dump of the entire cluster state can be requested by a remote poller from any single node in the cluster on port 8649. This design has positive consequences for the overall scalability and resiliency of the system. Only one node per cluster needs to be polled to glean the entire cluster status, and no amount of individual node failure adversely affects the overall system.

Reconsidering our earlier example of gathering a CPU metric from 20,000 hosts, and assuming that the hosts are now organized into 200 Ganglia clusters of 100 hosts each, gmond reduces the polling burden by two orders of magnitude. Further, for the 200 necessary network connections the poller must make, every metric (CPU, disk, memory, network, etc.) on every individual cluster node is recorded instead of just the single CPU metric. The recent addition of sFlow support to gmond (as described in Chapter 8) lightens the metric collection and polling load even further, enabling Ganglia to scale to cloud-sized networks.

What performs the actual work of polling gmond clusters and storing the metric data to disk for later use? The short answer is also the title of the next section: gmetad, but there is a longer and more involved answer that, like everything else we've talked about so far, is made possible by Ganglia's unique design. Given that gmond operates on its own, absent of any dependency on and ignorant of the policies or requirements of a centralized poller, consider that there could in fact be more than one poller. Any number of external polling engines could conceivably interrogate any combination of

gmond clusters within the grid without any risk of conflict or indeed any need to know anything about each other.

Multiple polling engines could be used to further distribute and lighten the load associated with metrics collection in large networks, but the idea also introduces the intriguing possibility of special-purpose pollers that could translate and/or export the data for use in other systems. As I write this, a couple of efforts along these lines are under way. The first is actually a modification to gmetad that allows gmetad to act as a bridge between gmond and Graphite, a highly scalable data visualization tool. The next is a project called gmond-zeromq, which listens to gmond broadcasts and exports data to a zeromq message bus.

gmetad: Bringing It All Together

In the previous section, we expressed a certain reluctance to compare gmond to the agent software found in more traditional monitoring systems. It's not because we think gmond is more efficient, scalable, and better designed than most agent software. All of that is, of course, true, but the real reason the comparison pains us is that Ganglia's architecture fundamentally alters the roles between traditional pollers and agents.

Instead of sitting around passively, waiting to be awakened by a monitoring server, gmond is always active, measuring, transmitting, and sharing. gmond imbues your network with a sort of intracluster self-awareness, making each host aware of its own characteristics as well as those of the hosts to which it's related. This architecture allows for a much simpler poller design, entirely removing the need for the poller to know what services to poll from which hosts. Such a poller needs only a list of hostnames that specifies at least one host per cluster. The clusters will then inform the poller as to what metrics are available and will also provide their values.

Of course, the poller will probably want to store the data it gleans from the cluster nodes, and RRDtool is a popular solution for this sort of data storage. Metrics are stored in "round robin" databases, which consist of static allocations of values for various chunks of time. If we polled our data every 10 seconds, for example, a single day's worth of these measurements would require the storage of 8,640 data points. This is fine for a few days of data, but it's not optimal to store 8,640 data points per day for a year for every metric on every machine in the network.

If, however, we were to average thirty 10-second data points together into a single value every 5 minutes, we could store two weeks worth of data using only 4,032 data points. Given your data retention requirements, RRDtool manages these data "rollups" internally, overwriting old values as new ones are added (hence the "round robin" moniker). This sort of data storage scheme lets us analyze recent data with great specificity while at the same time providing years of historical data in a few megabytes of disk space. It has the added benefit of allocating all of the required disk space up front, giving us a very predictable capacity planning model. We'll talk more about RRDtool in Chapter 3.

gmetad, as depicted in Figure 1-1, is foreshadowed pretty well by the previous few paragraphs. It is a simple poller that, given a list of cluster nodes, will poll each cluster, writing whatever data values are returned for every metric on every host to individual round robin databases.

You'll recall that "polling" each cluster requires only that the poller open a read socket to the target gmond node's port 8649, a feat readily accomplished by telnet. Indeed, gmetad could easily be replaced by a shell script that used netcat to glean the XML dump from various gmond nodes and then parse and write the data to RRDtool databases via command-line tools. As of this writing, there is, in fact, already a Python-based replacement for gmetad, which adds a plug-in architecture, making it easier to write custom data-handling logic.

gmetad has a few other interesting features, including the ability to poll data from other gmetad instances, allowing for the creation of federated hierarchal architectures. It includes interactive query functionality and may be polled by external monitoring systems via a simple text protocol on TCP port 8652. Finally, as mentioned in the previous section, gmetad is also capable of sending data to Graphite, a highly scalable data visualization engine.

gweb: Next-Generation Data Analysis

But enough about data collection and storage. I know why you're really here: visualization. You want graphs that make your data dance, brimming with timely, accurate data and contrasted, meaningful colors. And not just pretty graphs, but a snazzy, well-designed UI to go with them—a UI that is generous with the data, summarizing the status of the entire data center in just a few graphs while still providing quick, easy access to every combination of any individual metrics. It should do this without demanding that you preconfigure anything, and it should encourage you to create your own graphs to explore and analyze your data in any way you can imagine.

If it seems like I'm reading your mind, it's because the Ganglia authors are engineers like you, who designed Ganglia's visualization UI, gweb, from their own notion of the ideal data visualization frontend. Quite a bit of thought and real-world experience has gone into its creation, and we think you'll find it a joy to work with. gweb gives you easy, instant access to any metric from any host in the network without making you define anything. It knows what hosts exist, and what metrics are available for those hosts, but it doesn't make you click through hierarchal lists of metrics to see graphs; rather, it graphically summarizes the entire grid using graphs that combine metrics by cluster and provides sane click-throughs for increased specificity.

If you're interested in something specific, you can specify a system name, or a regex or type-glob to combine various metrics from various hosts to create a custom graph of exactly what you want to see. gweb supports click-dragging in the graphs to change the time period, includes a means to easily (and programatically) extract data in various

textual formats (CSV, JSON, and more), and sports a fully functional URL interface so that you can embed interesting graphs into other programs via predictable URLs. There are many other features I could mention—so many, in fact, that we've dedicated an entire chapter (Chapter 4) to gweb alone, so for now we'll have to content ourselves with this short description.

Before I move on, however, I should mention that gweb is a PHP program, which most people run under the Apache web server (although any web server with PHP or FastCGI support should do the trick). It is usually installed on the same physical hardware as gmetad, because it needs access to the RRD databases created by the poller. Installation details and specific software requirements are provided in Chapter 2.

But Wait! That's Not All!

Chapter 2 deals with the installation and configuration of gmond, gmetad, and gweb, and as previously mentioned, Chapter 4 covers gweb's functionality in more detail, but there's a lot more to talk about.

We've documented everything you might ever want to know about extending Ganglia's metric-collection functionality in Chapter 5, from easily adding new metrics through shell scripts using gmetric to writing full-fledged plug-ins for gmond in C, C++, or Python. If you're adept at any of those languages, you should appreciate the thorough documentation of gmond's internals included herein, written by the implementor of gmond's modular interface. I personally wish that documentation of this quality had existed when I undertook to write my first gmond module.

Anyone who has spent any time on the Ganglia mailing lists will recognize the names of the authors of Chapter 6. Bernard and Daniel both made the mistake of answering one too many questions on the Ganglia-General list and have hence been tasked with writing a chapter on troubleshooting. If you have a problem that isn't covered in Chapter 6, odds are you'll eventually get the answer you're looking for from either Bernard or Daniel on the Ganglia lists.

Chapter 7 and Chapter 8 cover interoperation with other monitoring systems. Integration with Nagios, arguably the most ubiquitous open source monitoring system today, is the subject of Chapter 7; Chapter 8 covers sFlow, an industry standard technology for monitoring high-speed switched networks. Ganglia includes built-in functionality that enables it to integrate with both of these tools, each of which extend Ganglia's functionality beyond what would otherwise be a limitation.

Finally, the chapter we're all most excited to bring you is Chapter 9, wherein we've collected detailed descriptions of real-world Ganglia installs from several fascinating organizations. Each case study highlights the varied and challenging monitoring requirements of the organization in question and goes on to describe the Ganglia configuration employed to satisfy them. Any customizations, integration with external tools, and other interesting hurdles are also discussed.

The authors, all of whom are members of and contributors to the Ganglia community, undertook to write this book ourselves to make sure it was the book we would have wanted to read, and we sincerely hope it meets your needs. Please don't hesitate to visit us online. Until then, we bid you adieu by borrowing the famous blessing from O'Reilly's *sed & awk* book: "May you solve interesting problems."

CHAPTER 2

Installing and Configuring Ganglia

Dave Josephsen, Frederiko Costa, Daniel Pocock, and Bernard Li

If you've made it this far, it is assumed that you've decided to join the ranks of the Ganglia user base. Congratulations! We'll have your Ganglia-user conspiracy to conquer the world kit shipped immediately. Until it arrives, feel free to read through this chapter, in which we show you how to install and configure the various Ganglia components. In this chapter, we cover the installation and configuration of Ganglia 3.1.x for some of the most popular operating systems, but these instructions should apply to later versions as well.

Installing Ganglia

As mentioned earlier, Ganglia is composed of three components: gmond, gmetad, and gweb. In this first section, we'll cover the installation and basic setup of each component.

gmond

gmond stands for Ganglia Monitoring Daemon. It's a lightweight service that must be installed on each node from which you want to have metrics collected. This daemon performs the actual metrics collection on each host using a simple listen/announce protocol to share the data it gleans with its peer nodes in the cluster. Using gmond, you can collect a lot of system metrics right out of the box, such as CPU, memory, disk, network, and data about active processes.

Requirements

gmond installation is straightforward, and the libraries it depends upon are installed by default on most modern Linux distributions (as of this writing, those libraries are libconfuse, pkgconfig, PCRE, and APR). Ganglia packages are available for most Linux distributions, so if you are using the package manager shipped with your distribution

(which is the suggested approach), resolving the dependencies should not be problematic.

Linux

The Ganglia components are available in a prepackaged binary format for most Linux distributions. We'll cover the two most popular types here: *.deb*- and *.rpm*-based systems.

Debian-based distributions. To install gmond on a Debian-based Linux distribution, execute:

```
user@host:# sudo apt-get install ganglia-monitor
```

RPM-based distributions. You'll find that some RPM-based distributions ship with Ganglia packages in the base repositories, and others require you to use special-purpose package repositories, such as the Red Hat project's EPEL (Extra Packages for Enterprise Linux) repository. If you're using a RPM-based distro, you should search in your current repositories for the gmond package:

```
user@host:$ yum search ganglia-gmond
```

If the search fails, chances are that Ganglia is not shipped with your RPM distribution. Red Hat users need to install Ganglia from the EPEL repository. The following examples demonstrate how to add the EPEL repository to Red Hat 5 and Red Hat 6.

If you need to add the EPEL repository, be sure to take careful note of the distro version and architecture you are running and match it to that of the EPEL you're adding.

For Red Hat 5.x:

```
user@host:# sudo rpm -Uvh \
http://mirror.ancl.hawaii.edu/linux/epel/5/i386/epel-release-5-4.noarch.rpm
```

For Red Hat 6.x:

```
user@host:# sudo rpm -Uvh \
http://mirror.chpc.utah.edu/pub/epel/6/i386/epel-release-6-7.noarch.rpm
```

Finally, to install gmond, type:

```
user@host:# sudo yum install ganglia-gmond
```

OS X

gmond compiles and runs fine on Mac OS X; however, at the time of this writing, there are no prepackaged binaries available. OS X users must therefore build Ganglia from source. Refer to the following instructions, which work for the latest Mac OS X Lion.

For other versions of Mac OS X, the dependencies might vary. Please refer to Ganglia's website for further information.

Several dependencies must be satisfied before building and installing Ganglia on OS X. These are, in the order they should be installed:

- Xcode >= 4.3
- MacPorts (requires Xcode)
- libconfuse (requires MacPorts)
- pkgconfig (requires MacPorts)
- PCRE (requires MacPorts)
- APR (requires MacPorts)

Xcode is a collection of development tools, and an Integrated Development Environment (IDE) for OS X. You will find Xcode at Apple's developer tools website for download or on the MAC OS X installation disc.

MacPorts is a collection of build instructions for popular open source software for OS X. It is architecturally identical to the venerable FreeBSD Ports system. To install MacPorts, download the installation disk image from the MacPorts website. MacPorts for MAC OS X Lion is here. If you're using Snow Leopard, the download is located here. For older versions, please refer here for documentation and download links.

Once MacPorts is installed and working properly, use it to install both libconfuse and pkconfig:

```
$ sudo port install libconfuse pkgconfig pcre apr
```

After satisfying the previously listed requirements, you are ready to proceed with the installation. Please download the latest Ganglia source release.

Change to the directory where the source file has been downloaded. Uncompress the *tar-gzip* file you have just downloaded:

```
$ tar -xvzf ganglia-major.minor.release.tar.gz
```

On Mac OS X 10.5+, you need to apply a patch so that gmond builds successfully. For further details on the patch, please visit the website. Download the patch file, copy it to the root of the build directory, and run the patch:

```
$ cd ganglia-major.minor.release
$ patch -p0 < patch-file
```

Assuming that you installed MacPorts under the default installation directory (*/opt/local*), export MacPorts' bin directory to your `PATH` and run the configure script, specifying the location of *lib/* and *include/* as options:

```
$ export PATH=$PATH:/opt/local/bin
$ ./configure LDFLAGS="-L/opt/local/lib" CPPFLAGS="-I/opt/local/include"
```

Compile and install Ganglia:

```
$ make
$ sudo make install
```

Solaris

Convenient binary packages for Solaris are distributed in the OpenCSW collection. Follow the standard procedure to install the OpenCSW. Run the `pkgutil` tool on Solaris, and then use the tool to install the package:

```
$ pkgutil
$ CSWgangliaagent
```

The default location for the configuration files on Solaris (OpenCSW) is */etc/opt/csw/ganglia*. You can now start and stop all the Ganglia processes using the normal SMF utility on Solaris, such as:

```
$ svcadm enable cswgmond
```

Other platforms

Because Ganglia is an open source project, it is possible to compile a runnable binary executable of the gmond agent on virtually any platform with a C compiler.

The Ganglia projects uses the autotools build system to detect the tools available on most Linux and UNIX-like environments and build the binaries.

The autotools build system is likely to have support for many other platforms that are not explicitly documented in this book. Please start by reading the INSTALL file in the source tree, and also look online for tips about Ganglia or generic tips about using autotools projects in your environment.

gmetad

gmetad (the Ganglia Meta Daemon) is the service that collects metric data from other gmetad and gmond sources and stores their state to disk in RRD format. It also provides a simple query mechanism for collecting specific information about groups of machines and supports hierarchical delegation, making possible the creation of federated monitoring domains.

Requirements

The requirements for installing gmetad on Linux are nearly the same as gmond, except for the addition of RRDtool, which is required to store and display time-series data collected from other gmetad or gmond sources.

Linux

Once again, you are encouraged to take advantage of the prepackaged binaries available in the repository of your Linux distribution; we provide instructions for the two most popular formats next.

Debian-based distributions. To install gmetad on a Debian-based Linux distribution, execute:

```
user@host:# sudo apt-get install gmetad
```

Compared to gmond, gmetad has additional software dependencies.

RPM-based distributions. As mentioned in the earlier gmond installation section, an EPEL repository must be installed if the base repositories don't provide gmetad. Refer to "gmond" on page 11 to add the EPEL repository. Once you're ready, type:

```
user@host:# sudo yum install ganglia-gmetad
```

OS X

There are only two functional differences between building gmond and gmetad on OS X. First, gmetad has one additional software dependency (RRDtool), and second, you must include the `--with-gmetad` option to the configure script, because only gmond is built by the default Makefile.

Following is the list of requirements that must be satisfied before you can build gmetad on Mac OS X:

- Xcode >= 4.3
- MacPorts (requires Xcode)
- libconfuse (requires MacPorts)
- pkgconfig (requires MacPorts)
- PCRE (requires MacPorts)
- APR (requires MacPorts)
- RRDtool (requires MacPorts)

Refer to "OS X" on page 12 for instructions on installing Xcode and MacPorts. Once you have those sorted out, install the following packages to satisfy the requirements:

```
$ sudo port install libconfuse pkgconfig pcre apr rrdtool
```

Once those packages have been installed, proceed with the Ganglia installation by downloading the latest Ganglia version.

Uncompress and extract the tarball you have just downloaded:

```
$ tar -xvzf ganglia-major.minor.release.tar.gz
```

Successfully building Ganglia 3.1.2 on OS X 10.5 requires that you apply the patch detailed here. Download the patch file and copy it to the root of the extracted Ganglia source tree, then apply it:

```
$ cd ganglia-major.minor.release
$ patch -p0 < patch-file
```

Assuming that you installed MacPorts under the default installation directory (*/opt/local*). Export MacPorts' */bin* directory to your `PATH`, and run the configure script, specifying the location of *lib/* and *include/* as options

```
$ export PATH=$PATH:/opt/local/bin
$ ./configure --with-gmetad LDFLAGS="-L/opt/local/lib" CPPFLAGS="-I/opt/local/include"
```

Compile and install Ganglia:

```
$ make
$ sudo make install
```

Solaris

Convenient binary packages for Solaris are distributed in the OpenCSW collection. Follow the standard procedure to install the OpenCSW. Run the `pkgutil` tool on Solaris, and then use the tool to install the package:

```
$ pkgutil
$ CSWgangliagmetad
```

The default location for the configuration files on Solaris (OpenCSW) is */etc/opt/csw/ganglia*. You can now start and stop all the Ganglia processes using the normal SMF utility on Solaris, as in:

```
$ svcadm enable cswgmetad
```

gweb

Ganglia wouldn't be complete without its web interface: gweb (Ganglia Web). After collecting several different metrics in order to evaluate how our cluster is performing, we certainly need a visual representation, preferably using graphics in the Web. gweb fills this gap. gweb is a PHP frontend in which you display all data stored by gmetad using your browser. Please see the "Demos" section here for live demos of the web frontend.

Requirements

As of Ganglia 3.4.0, the web interface is a separate distribution tarball maintained in a separate source code repository. The release cycle and version numbers of gweb are no

longer in lockstep with the release cycle and version numbers of the Ganglia gmond and the gmetad daemon.

Ganglia developers support gweb 3.4.0 with all versions of gmond/gmetad version 3.1.x and higher. Future versions of gweb may require a later version of gmond/gmetad. It's recommended to check the installation documentation for exact details whenever installing or upgrading gweb.

The frontend, as already mentioned, is a web application. This book covers gweb versions 3.4.x and later, which may not be available to all distributions, requiring more work to get it installed. Before proceeding, please review the requirements to install gweb:

- Apache Web Server
- PHP 5.2 or later
- PHP JSON extension installed and enabled

Linux

If you are installing from the repositories, the installation is pretty straightforward. Requirements will be automatically satisfied, and within a few commands you should be able to play with the web interface.

Debian-based distributions. To install gweb on a Debian-based Linux distribution, execute the following command as either root or user with high privilege:

```
root@host:# apt-get install apache2 php5 php5-json
```

This command installs Apache and PHP 5 to satisfy its dependencies, in case you don't have it already installed. You might have to enable the PHP JSON module as well. Then execute this command:

```
root@host:# grep ^extension=json.so /etc/php5/conf.d/json.ini
```

and if the module is not enabled, enable it with the following command:

```
root@host:# echo 'extension=json.so' >> /etc/php5/conf.d/json.ini
```

You are ready to download the latest gweb. Once it's downloaded, explode and edit Makefile to install gweb:

```
root@host:# tar -xvzf ganglia-web-major.minor.release.tar.gz
root@host:# cd ganglia-web-major.minor.release
```

Edit Makefile and set `DESTDIR` and `APACHE_USER` variables. On Debian-based distros, the default settings are the following:

```
# Location where gweb should be installed to
DESTDIR = /var/www/html/ganglia2
APACHE_USER = www-data
...
```

This means that gweb will be available to the user here. You can change to whichever name you want. Finally, run the following command:

```
root@host:# make install
```

If no errors are shown, gweb is successfully installed. Skip to "Configuring Ganglia" on page 20 for further information on gweb settings.

RPM-based distributions. The way to install gweb on a RPM-based distribution is very similar to installing gweb on a Debian-based distribution. Start by installing Apache and PHP 5:

```
root@host:# yum install httpd php
```

You also need to enable the JSON extension for PHP. It's already included in PHP 5.2 or later. Make sure it's enabled by checking the content of */etc/php.d/json.ini* file. You should have something similar to the following listing:

```
extension=json.ini
```

Download the latest gweb. Once downloaded, explode and edit Makefile to install gweb 2:

```
root@host:# tar -xvzf ganglia-web-major.minor.release.tar.gz
root@host:# cd ganglia-web-major.minor.release
```

Edit Makefile and set the `DESTDIR` and `APACHE_USER` variables. On RPM-based distros, the default settings are:

```
# Location where gweb should be installed to
DESTDIR = /var/www/html/ganglia2
APACHE_USER = apache
...
```

This means that gweb will be available here. You can change to whichever name you want. Finally, run:

```
root@host:# make install
```

If no errors are shown, gweb is successfully installed. Skip to "Configuring Ganglia" on page 20 for further information on gweb settings.

OS X

If you need to install gweb on Mac OS X, you have to follow a slightly different approach than if you were installing in Linux. Again, there isn't any binary package for Mac OS X, leaving you with the option of downloading the source from the website. Before downloading, you have to make sure that your Mac OS X has shipped with a few of the requirements. That's what this section is about.

First off, an HTTP server is required, and chances are good that your Mac OS X installation was shipped with Apache Web Server. You can also install it via MacPorts, but this approach is not covered here. It is your choice. In order to verify your Apache

installation, go to System Preferences → Sharing. Turn *Web Services* on if it is off. Make sure it's running by typing **`http://localhost`** on your browser. You should see a test page. You can also load Apache via Terminal by typing:

```
$ sudo launchctl load -w /System/Library/LaunchDaemons/org.apache.httpd.plist
```

PHP is also required to run gweb. PHP is shipped with Mac OS X, but it's not enabled by default. To enable, edit the *httpd.conf* file and uncomment the line that loads the `php5_module`.

```
$ cd /etc/apache2
$ sudo vim httpd.conf
```

Search for the following line, uncomment (strip the #) it, and save the file:

```
# LoadModule php5_module libexec/apache2/libphp5.so
```

Restart Apache:

```
$ sudo launchctl unload -w /System/Library/LaunchDaemons/org.apache.httpd.plist
$ sudo launchctl load -w /System/Library/LaunchDaemons/org.apache.httpd.plist
```

Now that you have satisfied the requirements, it's time to download and install gweb 2. Please download the latest release. Once you have finished, change to the directory where the file is located and extract its content. Next, `cd` to the extraction directory:

```
$ tar -xvzf ganglia-web-major.minor.release.tar.gz
$ cd ganglia-web-major.minor.release
```

This next step really depends on how Apache Web Server is set up on your system. You need to find out where Apache serves its pages from or, more specifically, its DocumentRoot. Of course, the following location isn't the only possibility, but for clarity's sake, we will work with the default settings. So here, we're using */Library/WebServer/Documents*:

```
$ grep -i documentroot /etc/apache2/httpd.conf
```

Edit the Makefile found in the tarball. Insert the location of your Apache's DocumentRoot and the name of the user that Apache runs. On Mac OS X Lion, the settings are:

```
# Location where gweb should be installed
DESTDIR = /Library/WebServer/Documents/ganglia2
APACHE_USER = _www
...
```

This means that gweb will be available to the user here. You can change this to whichever name you want. Finally, run:

```
$ sudo make install
```

If no errors are shown, Ganglia Web is successfully installed. Read the next sections to configure Ganglia prior to running it for the first time.

Solaris

Convenient binary packages for Solaris are distributed in the OpenCSW collection. Follow the standard procedure to install the OpenCSW. Run the `pkgutil` tool on Solaris, and then use the tool to install the package:

```
$ pkgutil
$ CSWgangliaweb
```

The default location for the configuration files on Solaris (OpenCSW) is */etc/opt/csw/ganglia*. You can now start and stop all the Ganglia processes using the normal SMF utility on Solaris, as in:

```
$ svcadm enable cswapache
```

Configuring Ganglia

The following subsections document the configuration specifics of each Ganglia component. The default configuration shipped with Ganglia "just works" in most environments with very little additional configuration, but we want to let you know what other options are available in addition to the default. We would also like you to understand how the choice of a particular option may affect Ganglia deployment in your environment.

gmond

gmond, summarized in Chapter 1, is installed on each host that you want to monitor. It interacts with the host operating system to obtain metrics and shares the metrics it collects with other hosts in the same cluster. Every gmond instance in the cluster knows the value of every metric collected by every host in the same cluster and by default provides an XML-formatted dump of the entire cluster state to any client that connects to gmond's port.

Topology considerations

gmond's default topology is a multicast mode, meaning that all nodes in the cluster both send and receive metrics, and every node maintains an in-memory database—stored as a hash table—containing the metrics of all nodes in the cluster. This topology is illustrated in Figure 2-1.

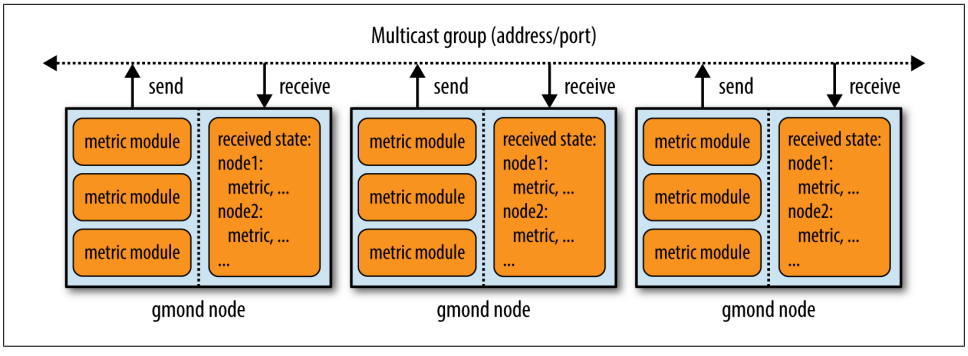


Figure 2-1. Default multicast topology

Of particular importance in this diagram is the disparate nature of the gmond daemon. Internally, gmond's sending and receiving halves are not linked (a fact that is emphasized in Figure 2-1 by the dashed vertical line). gmond does not talk to itself—it only talks to the network. Any local data captured by the metric modules are transmitted directly to the network by the sender, and the receiver's internal database contains only metric data gleaned from the network.

This topology is adequate for most environments, but in some cases it is desirable to specify a few specific listeners rather than allowing every node to receive (and thereby waste CPU cycles to process) metrics from every other node. More detail about this architecture is provided in Chapter 3.

The use of "deaf" nodes, as illustrated in Figure 2-2, eliminates the processing overhead associated with large clusters. The deaf and mute parameters exist to allow some gmond nodes to act as special-purpose aggregators and relays for other gmond nodes. Mute means that the node does not transmit; it will not even collect information about itself but will aggregate the metric data from other gmond daemons in the cluster. Deaf means that the node does not receive any metrics from the network; it will not listen to state information from multicast peers, but if it is not muted, it will continue sending out its own metrics for any other node that does listen.

The use of multicast is not required in any topology. The deaf/mute topology can be implemented using UDP unicast, which may be desirable when multicast is not practical or preferred (see Figure 2-3).

Further, it is possible to mix and match the deaf/mute, and default topologies to create a system architecture that better suits your environment. The only topological requirements are:

1. At least one gmond instance must receive all the metrics from all nodes in the cluster.
2. Periodically, gmetad must poll the gmond instance that holds the entire cluster state.

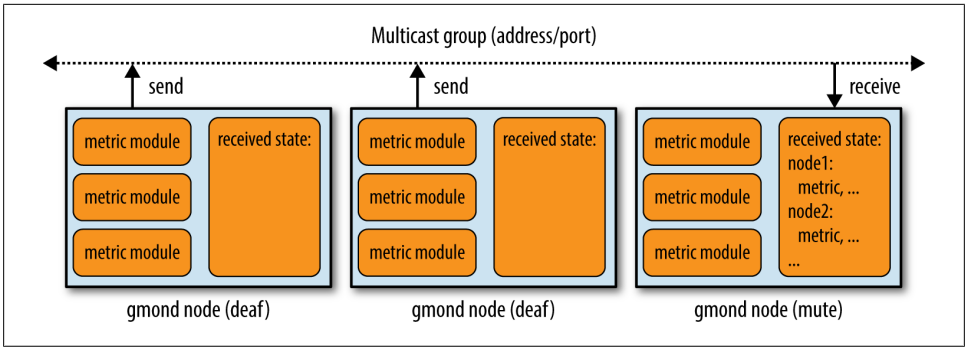


Figure 2-2. Deaf/mute multicast topology

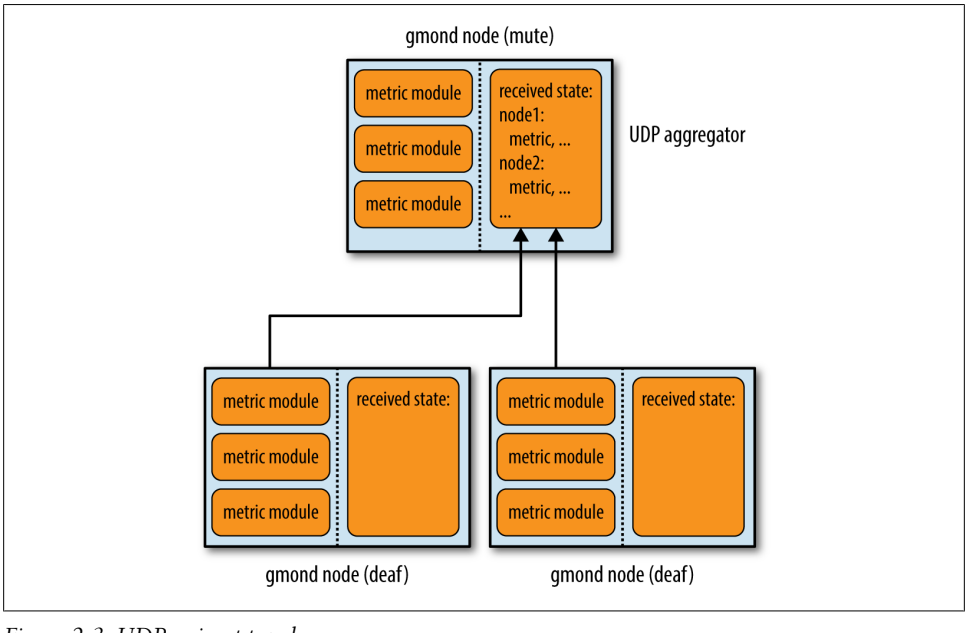


Figure 2-3. UDP unicast topology

In practice, however, nodes not configured with any multicast connectivity do not need to be deaf; it can be useful to configure such nodes to send metrics to themselves using the address 127.0.0.1 so that they will keep a record of their own metrics locally. This makes it possible to make a TCP probe of any gmond for an XML about its own agent state while troubleshooting.

For a more thorough discussion of topology and scalability considerations, see Chapter 3.

Configuration file

You can generate a default configuration file for gmond by running the following command:

```
user@host:$ gmond -t
```

The configuration file is composed of sections, enclosed in curly braces, that fall roughly into two logical categories. The sections in the first category deal with host and cluster configuration; those in the second category deal with the specifics of metrics collection and scheduling.

All section names and attributes are case insensitive. The following attributes, for example, are all equivalent:

```
name NAME Name NaMe
```

Some configuration sections are optional; others are required. Some may be defined in the configuration file multiple times; others must appear only once. Some sections may contain subsections.

The `include` directive can be used to break up the *gmond.conf* file into multiple files for environments with large complex configurations. The `include` directive supports the use of typeglobs. For example, the line:

```
include ('/etc/ganglia/conf.d/*.conf')
```

would instruct gmond to load all files in */etc/ganglia/conf.d/* that ended in ".conf".

gmond.conf: Quick Start

To get gmond up and running quickly just to poke around, all you should need to set is the `name` attribute in the "cluster" section of the default configuration file.

The configuration file is parsed using libconfuse, a third-party API for configuration files. The normal rules of libconfuse file format apply. In particular, boolean values can be set using `yes`, `true`, and `on` for a positive value and their opposites, `no`, `false`, and `off` for a negative value. Boolean values are not handled in a case-sensitive manner.

There are eight sections that deal with the configuration of the host itself.

Section: globals. The globals section configures the general characteristics of the daemon itself. It should appear only once in the configuration file. The following is the default globals section from Ganglia 3.3.1:

```
globals {
  daemonize = yes
  setuid = yes
  user = nobody
  debug_level = 0
  max_udp_msg_len = 1472
```

```
  mute = no
  deaf = no
  allow_extra_data = yes
  host_dmax = 86400 /*secs. Expires (removes from web interface) hosts in 1 day */
  host_tmax = 20 /*secs */
  cleanup_threshold = 300 /*secs */
  gexec = no
  send_metadata_interval = 0 /*secs */
}
```

`daemonize` *(boolean)*
: When true, gmond will fork and run in the background. Set this value to false if you're running gmond under a daemon manager such as daemontools.

`setuid` *(boolean)*
: When true, gmond will set its effective UID to the UID of the user specified by the `user` attribute. When false, gmond will not change its effective user.

`debug_level` *(integer value)*
: When set to zero (0), gmond will run normally. A `debug_level` greater than zero will result in gmond running in the foreground and outputting debugging information. The higher the `debug_level`, the more verbose the output.

`max_udp_msg_len` *(integer value)*
: This value is the maximum size that one packet sent by gmond will contain. It is not a good idea to change this value.

`mute` *(boolean)*
: When true, gmond will not send data, regardless of any other configuration directive. "Mute" gmond nodes are only mute when it comes to other gmond daemons. They still respond to queries from external pollers such as gmetad.

`deaf` *(boolean)*
: When true, gmond will not receive data, regardless of any other configuration directives. In large grids with thousands of nodes per cluster, or carefully optimized HPC grids, in which every CPU cycle spent on something other than the problem is a wasted cycle, "normal" compute nodes are often configured as deaf in order to minimize the overhead associated with aggregating cluster state. In these instances, dedicated nodes are set aside to be mute. In such a setup, the performance metrics of the mute nodes aren't measured because those nodes aren't a computationally relevant portion of the grid. Their job is to aggregate, so their performance data would pollute that of the functional portion of the cluster.

`allow_extra_data` *(boolean)*
: When false, gmond will not send the `EXTRA_ELEMENT` and `EXTRA_DATA` parts of the XML. This value might be useful if you are using your own frontend and would like to save some bandwidth.

`host_dmax` *(integer_value in seconds)*
: Stands for "delete max." When set to 0, gmond will never delete a host from its list, even when a remote host has stopped reporting. If host_dmax is set to a positive

number, gmond will flush a host after it has not heard from it for host_dmax seconds.

`host_tmax` *(integer_value in seconds)*
: Stands for "timeout max." Represents the maximum amount of time that gmond should wait between updates from a host. Because messages may get lost in the network, gmond will consider the host as being down if it has not received any messages from it after four times this value.

`cleanup_threshold` *(integer_value in seconds)*
: Minimum amount of time before gmond will clean up expired data.

`gexec` *(boolean)*
: When true, gmond will announce the host's availability to run gexec jobs. This approach requires that gexecd be running on the host and the proper keys have been installed.

`send_metadata_interval` *(integer_value in seconds)*
: Establishes the interval at which gmond will send or resend the metadata packets that describe each enabled metric. This directive by default is set to 0, which means that gmond will send the metadata packets only at startup and upon request from other gmond nodes running remotely. If a new machine running gmond is added to a cluster, it needs to announce itself and inform all other nodes of the metrics that it currently supports. In multicast mode, this isn't a problem, because any node can request the metadata of all other nodes in the cluster. However, in unicast mode, a resend interval must be established. The interval value is the minimum number of seconds between resends.

`module_dir` *(path; optional)*
: Indicates the directory where the metric collection modules are found. If omitted, defaults to the value of the compile-time option: `--with-moduledir`. This option, in turn, defaults to a subdirectory named *Ganglia* in the directory where libganglia will be installed. To discover the default value in a particular gmond binary, generate a sample configuration file by running:

```
# gmond -t
```

  For example, in a 32-bit Intel-compatible Linux host, the default is usually at */usr/lib/ganglia*.

Section: cluster. Each gmond daemon will report information about the cluster in which it resides using the attributes defined in the cluster section. The default values are the string `"unspecified"`; the system is usable with the default values. This section may appear only once in the configuration file. Following is the default cluster section:

```
cluster {
  name = "unspecified"
  owner = "unspecified"
  latlong = "unspecified"
  url = "unspecified"
}
```

The attributes in the cluster section directly correspond to the attributes in the `CLUSTER` tag in the XML output from gmond.

`name` *(text)*
: Specifies the name of the cluster. When the node is polled for an XML summary of cluster state, this name is inserted in the `CLUSTER` element. The gmetad polling the node uses this value to name the directory where the cluster data RRD files are stored. It supersedes a cluster name specified in the *gmetad.conf* configuration file.

`owner` *(text)*
: Specifies the administrators of the cluster.

`latlong` *(text)*
: Specifies the latitude and longitude GPS coordinates of this cluster on earth.

`url` *(text)*
: Intended to refer to a URL with information specific to the cluster, such as the cluster's purpose or usage details.

The `name` attribute specified in the cluster section does place this host into a cluster. The multicast address and the UDP port specify whether a host is on the cluster. The name attribute acts justs as an identifier when polling.

Section: host. The host section provides information about the host running this instance of gmond. Currently, only the location string attribute is supported. The default host section is:

```
host {
  location = "unspecified"
}
```

`location` *(text)*
: The location of the host in a format relative to the site, although `rack,U[,blade]` is often used.

Section: UDP channels. UDP send and receive channels establish how gmond nodes talk to each other. Clusters are defined by UDP communication channels, which is to say, that a cluster is nothing more than some number of gmond nodes that share the same send and/or receive channels.

By default, every node in a gmond cluster multicasts its own metric data to its peers via UDP and listens for similar UDP multicasts from its peers. This is easy to set up and maintain: every node in the cluster shares the same multicast address, and new nodes are automatically discovered. However, as we mentioned in the previous section on

deaf and mute nodes, it is sometimes desirable to specify individual nodes by their unicast address.

For this reason, any number of gmond send and receive channels may be configured to meet the needs of your particular environment. Each configured send channel defines a new way that gmond will advertise its metrics, and each receive channel defines a way that gmond will receive metrics from other nodes. Channels may be either unicast or multicast and either IPv4 or IPv6.

Note that a gmond node should not be configured to contribute metrics to more than one Ganglia cluster, nor should you attempt to receive metrics for more than one cluster.

UDP channels are created using the `udp_(send|receive)_channel` sections. Following is the default UDP send channel:

```
udp_send_channel {
  #bind_hostname = yes
  mcast_join = 239.2.11.71
  port = 8649
  ttl = 1
}
```

`bind_hostname` *(boolean; optional, for multicast or unicast)*
: Tells gmond to use a source address that resolves to the machine's hostname.

`mcast_join` *(IP; optional, for multicast only)*
: When specified, gmond will create a UDP socket and join the multicast group specified by the IP. This option creates a multicast channel and is mutually exclusive with `host`.

`mcast_if` *(text; optional, for multicast only)*
: When specified, gmond will send data from the specified interface (`eth0`, for example).

`host` *(text or IP; optional, for unicast only)*
: When specified, gmond will send data to the named host. This option creates a unicast channel and is mutually exclusive with `mcast_join`.

`port` *(number; optional, for multicast and unicast)*
: The port number to which gmond will send data. If it's not set, port 8649 is used by default.

`ttl` *(number; optional, for multicast or unicast)*
: The time-to-live, this setting is particularly important for multicast environments, as it limits the number of hops over which the metric transmissions are permitted to propagate. Setting this value to any value higher than necessary could result in metrics being transmitted across WAN connections to multiple sites or even out into the global Internet.

Following is the default UDP receive channel:

```
udp_recv_channel {
  mcast_join = 239.2.11.71
  port = 8649
  bind = 239.2.11.71
}
```

`mcast_join` *(IP; optional, for multicast only)*
: When specified, gmond will listen for multicast packets from the multicast group specified by the IP. If you do not specify multicast attributes, gmond will create a unicast UDP server on the specified port.

`mcast_if` *(text; optional, for multicast only)*
: When specified, gmond will listen for data on the specified interface (`eth0`, for example).

`bind` *(IP; optional, for multicast or unicast)*
: When specified, gmond will bind to the local address specified.

`port` *(number; optional, for multicast or unicast)*
: The port number from which gmond will receive data. If not set, port 8649 is used by default.

`family` *(inet4|inet6; optional, for multicast or unicast)*
: The IP version, which defaults to `inet4`. If you want to bind the port to an inet6 port, specify `inet6` in the family attribute. Ganglia will not allow IPv6=>IPv4 mapping (for portability and security reasons). If you want to listen on both `inet4` and `inet6` for a particular port, define two separate receive channels for that port.

`acl` *(ACL definition; optional, for multicast or unicast)*
: An access control list may be specified for fine-grained access control to a receive channel. See "Access control" on page 29 for details on ACL syntax.

Section: TCP Accept Channels. TCP Accept Channels establish the means by which gmond nodes report the cluster state to gmetad or other external pollers. Configure as many of them as you like. The default TCP Accept Channel is:

```
tcp_accept_channel {
   port = 8649
}
```

`bind` *(IP; optional)*
: When specified, gmond will bind to the local address specified.

`port` *(number)*
: The port number on which gmond will accept connections.

`family` *(inet4|inet6; optional)*
: The IP version, which defaults to `inet4`. If you want to bind the port to an inet6 port, you need to specify `inet6` in the family attribute. Ganglia will not allow IPv6=>IPv4 mapping (for portability and security reasons). If you want to listen

on both `inet4` and `inet6` for a particular port, define two separate receive channels for that port.

`interface` *(text; optional)*
: When specified, gmond will listen for data on the specified interface (`eth0`, for example).

`acl` *(ACL definition; optional)*
: An access control list (discussed in the following section) may be specified for fine-grained access control to an accept channel.

Access control. The `udp_recv_channel` and `tcp_accept_channel` directives can contain an Access Control List (ACL). This list allows you to specify addresses and address ranges from which gmond will accept or deny connections. Following is an example of an ACL:

```
acl {
    default = "deny"
    access {
      ip = 192.168.0.0
      mask = 24
      action = "allow"
    }
    access {
      ip = ::ff:1.2.3.0
      mask = 120
      action = "deny"
    }
}
```

The syntax should be fairly self-explanatory to anyone with a passing familiarity with access control concepts. The `default` attribute defines the default policy for the entire ACL. Any number of `access` blocks may be specified that list hostnames or IP addresses and associate `allow` or `deny` actions to those addresses. The `mask` attribute defines a subnet mask in CIDR notation, allowing you to specify address ranges instead of individual addresses. Notice that in case of conflicting ACLs, the first match wins.

Optional section: sFlow. sFlow is an industry standard technology for monitoring high-speed switched networks. Originally targeted at embedded network hardware, sFlow collectors now exist for general-purpose operating systems as well as popular applications such as Tomcat, memcached, and the Apache Web Server. gmond can be configured to act as a collector for sFlow agents on the network, packaging the sFlow agent data so that it may be transparently reported to gmetad. Further information about sFlow interoperability is provided in Chapter 8. The entire sFlow section is optional. Following is the default sFlow configuration:

```
#sflow {
# udp_port = 6343
# accept_vm_metrics = yes
# accept_jvm_metrics = yes
# multiple_jvm_instances = no
# accept_http_metrics = yes
# multiple_http_instances = no
```

```
    # accept_memcache_metrics = yes
    # multiple_memcache_instances = no
    #}
```

`udp_port` *(number; optional)*
: The port on which gmond will accept sFlow data.

The remaining configuration parameters deal with application-specific sFlow data types. See Chapter 8 for details.

Section: modules. The modules section contains the parameters that are necessary to load a metric module. Metric modules are dynamically loadable shared object files that extend the available metrics gmond is able to collect. Much more information about extending gmond with modules can be found in Chapter 5.

Each `modules` section must contain at least one `module` subsection. The `module` subsection is made up of five attributes. The default configuration contains every module available in the default installation, so you should not have to change this section unless you're adding new modules. The configuration for an imaginary `example_module` is provided here:

```
modules {
     module {
       name = "example_module"
       language = "C/C++"
       enabled = yes
       path = "modexample.so"
       params = "An extra raw parameter"
       param RandomMax {
         value = 75
       }
       param ConstantValue {
         value = 25
       }
     }
}
```

`name` *(text)*
: The name of the module as determined by the module structure if the module was developed in C/C++. Alternatively, the name can be the name of the source file if the module has been implemented in an interpreted language such as Python.

`language` *(text; optional)*
: The source code language in which the module was implemented. Defaults to `"C/C++"` if unspecified. Currently, only C, C++, and Python are supported.

`enabled` *(boolean; optional)*
: Allows a metric module to be easily enabled or disabled through the configuration file. If the `enabled` directive is not included in the module configuration, the enabled state will default to `yes`.

If a module that has been disabled contains a metric that is still listed as part of a collection group, gmond will produce a warning message but will continue to function normally by ignoring the metric.

`path` *(text)*
: The path from which gmond is expected to load the module (C/C++ compiled dynamically loadable module only). If the value of `path` does not begin with a forward slash, the value will be appended to that of the `module_path` attribute from the `globals` section.

`params` *(text; optional)*
: Used to pass a string parameter to the module initialization function (C/C++ module only). Multiple parameters can be passed to the module's initialization function by including one or more param sections. Each param section must be named and contain a value directive.

Section: collection_group. The `collection_group` entries specify the metrics that gmond will collect, as well as how often gmond will collect and broadcast them. You may define as many collection groups as you wish. Each collection group must contain at least one `metric` section.

These are logical groupings of metrics based on common collection intervals. The groupings defined in *gmond.conf* do not affect the groupings used in the web interface, nor is it possible to use this mechanism to specify the names of the groups for the web interface. An excerpt from the default configuration follows:

```
collection_group {
  collect_once = yes
  time_threshold = 1200
  metric {
    name = "cpu_num"
    title = "CPU Count"
  }
}
collection_group {
  collect_every = 20
  time_threshold = 90
  /* CPU status */
  metric {
    name = "cpu_user"
    value_threshold = "1.0"
    title = "CPU User"
  }
  metric {
    name = "cpu_system"
    value_threshold = "1.0"
    title = "CPU System"
  }
}
```

`collect_once` *(boolean)*
: Some metrics are said to be "nonvolatile" in that they will not change between reboots. This includes metrics such as the OS type or the number of CPUs installed in the system. These metrics need to be collected only once at startup and are configured by setting the `collect_once` attribute to `yes`. This attribute is mutually exclusive with `collect_every`.

`collect_every` *(seconds)*
: This value specifies the polling interval for the collection group. In the previous example, the `cpu_user` and `cpu_system` metrics will be collected every 20 seconds.

`time_threshold` *(seconds)*
: The maximum amount of time that can pass before gmond sends all metrics specified in the `collection_group` to all configured `udp_send_channels`.

`name` *(text)*
: The name of an individual metric as defined within the metric collection module. Typically, each loaded module defines several individual metrics. An alternative to `name` is `name_match`. By using the `name_match` parameter instead of `name`, it is possible to use a single definition to configure multiple metrics that match a regular expression. The Perl-compatible regular expression (pcre) syntax is used (e.g., `name_match = "multicpu_([a-z]+)([0-9]+)"`).

You can get a list of the available metric names by running gmond with an `-m` switch.

`value_threshold` *(number)*
: Each time a metric value is collected, the new value is compared with the last measured value. If the difference between the last value and the current value is greater than the `value_threshold`, the entire collection group is sent to the `udp_send_channels` defined. The units denoted by the value vary according to the metric module. For CPU stats, for example, the value represents a percentage, and network stats interpret the value as a raw number of bytes.

Any time a `value_threshold` is surpassed by any single metric in a collection group, all metrics in that collection group are sent to every UDP receive channel.

`title` *(text)*
: A user-friendly title for the metric for use on the web frontend.

gmetad

gmetad, the Ganglia Meta Daemon, is installed on the host that will collect and aggregate the metrics collected by the hosts running gmond. By default, gmetad will collect and aggregate these metrics in RRD files, but it is possible to configure gmetad to forward metrics to external systems such as Graphite instead.

gmetad listens on tcp port 8651 for connections from remote gmetad instances and will provide an XML dump of the grid state to authorized hosts. It also responds to interactive requests on tcp port 8652. The interactive facility allows simple subtree and summation views of the grid state XML tree. gweb uses this interactive query facility to present information that doesn't fit naturally in RRD files, such as OS version.

gmetad topology

The simplest topology is a single gmetad process polling one or more gmond instances, as illustrated in Figure 2-4.

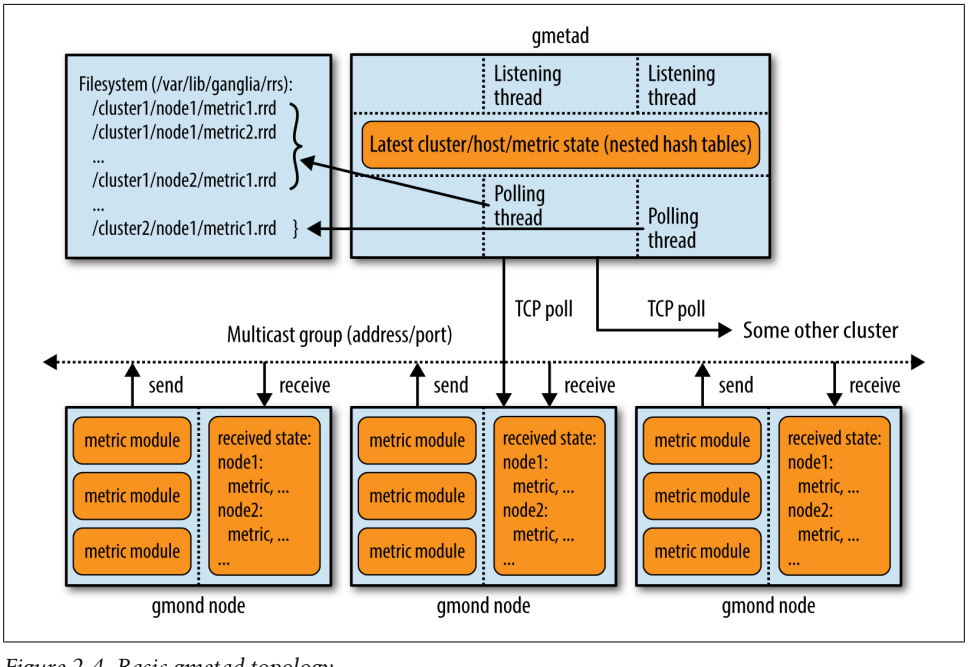


Figure 2-4. Basic gmetad topology

Redundancy/high availability is a common requirement and is easily implemented. Figure 2-5 shows an example in which two (redundant) gmetads poll multiple gmonds in the same cluster. The gmetads will poll node2 only if they are unable to poll node1 successfully. Both gmetads are always polling (active-active clustering).

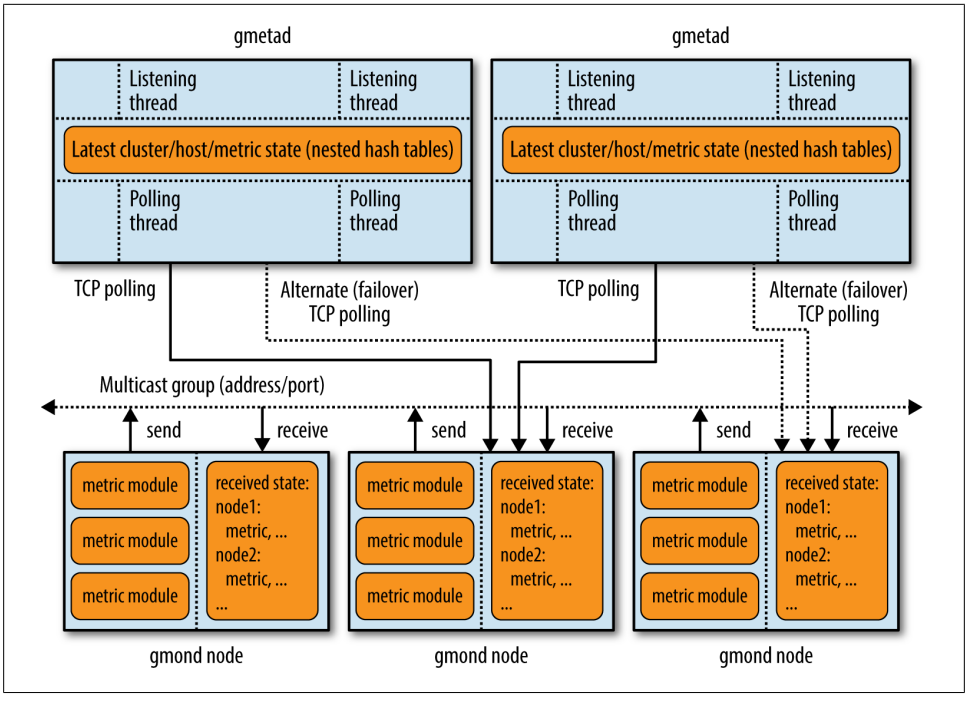


Figure 2-5. gmetad topology for high availability (active-active)

gmetad is not limited to polling gmond: a gmetad can poll another gmetad to create a hierarchy of gmetads. This concept is illustrated in Figure 2-6.

gmetad, by default, writes all metric data directly to RRD files on the filesystem, as illustrated in Figure 2-4.

In large installations in which there is an IO constraint, rrdcached acts as a buffer between gmetad and the RRD files, as illustrated in Figure 2-7.

gmetad.conf: gmetad configuration file

The *gmetad.conf* configuration file is composed of single-line attributes and their corresponding values. Attribute names are case insensitive, but their values are not. The following attributes, for example, are all equivalent:

```
name NAME Name NaMe
```

Most attributes are optional; others are required. Some may be defined in the configuration file multiple times; others must appear only once.

The data_source attribute. The `data_source` attribute is the heart of gmetad configuration. Each `data_source` line describes either a gmond cluster or a gmetad grid from which this gmetad instance will collect information. gmetad is smart enough to automatically make the distinction between a cluster and a grid, so the `data_source` syntax is the same

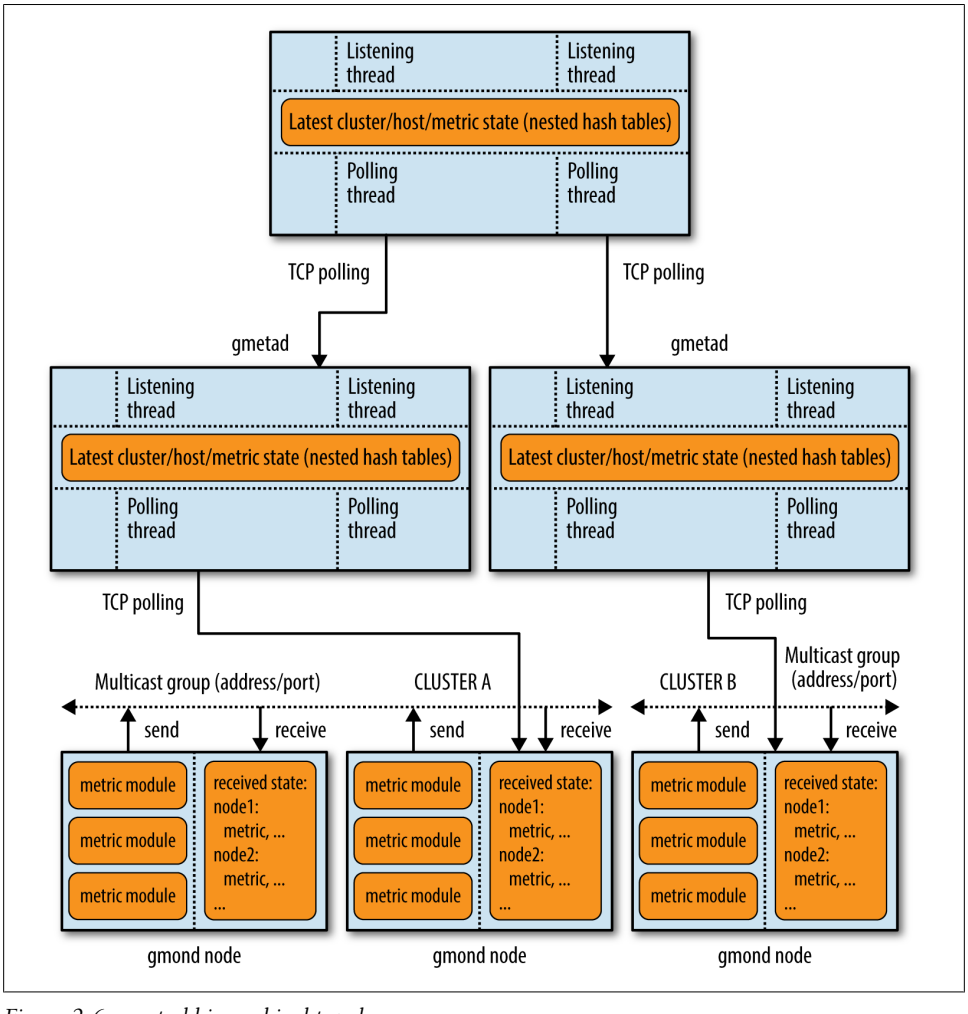


Figure 2-6. gmetad hierarchical topology

for either. If gmetad detects that the `data_source` refers to a cluster, it will maintain a complete set of round robin databases for the data source. If, however, gmetad detects that the `data_source` refers to a grid, it will maintain only summary RRDs.

Setting the `scalable` attribute to `off` overrides this behavior and forces gmetad to maintain a full set of RRD files for grid data sources.

The following examples, excerpted from the default configuration file, are valid data sources:

```
data_source "my cluster" 10 localhost my.machine.edu:8649 1.2.3.5:8655
data_source "my grid" 50 1.3.4.7:8655 grid.org:8651 grid-backup.org:8651
data_source "another source" 1.3.4.8:8655  1.3.4.8
```

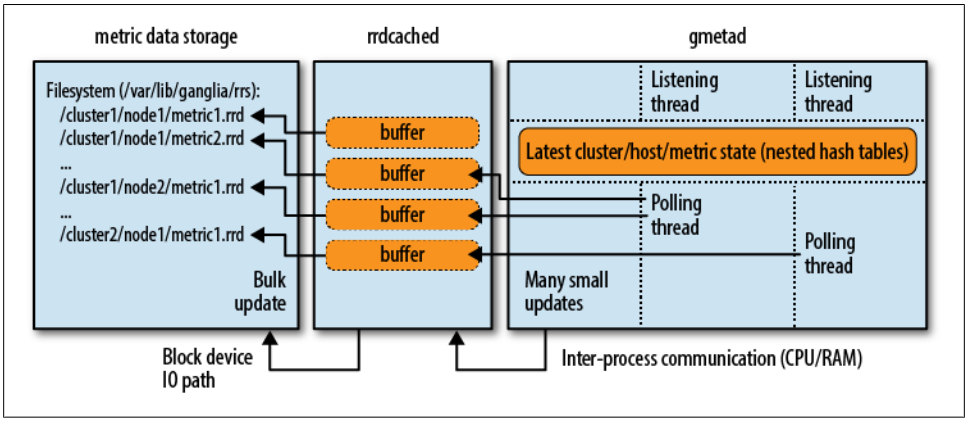


Figure 2-7. gmetad with rrdcached

Each `data_source` is composed of three fields. The first is a string that uniquely identifies the source. The second field is a number that specifies the polling interval for the `data_source` in seconds. The third is a space-separated list of hosts from which gmetad may poll the data. The addresses may be specified as IP addresses or DNS hostnames and may optionally be suffixed by a colon followed by the port number where the gmond `tcp_accept_channel` is to be found. If no port number is specified, gmetad will attempt to connect to tcp/8649.

gmetad will check each specified host in order, taking the status data from the first host to respond, so it's not necessary to specify every host in a cluster in the `data_source` definition. Two or three are usually sufficient to ensure that data is collected in the event of a node failure.

gmetad daemon behavior. The following attributes affect the functioning of the gmetad daemon itself:

`gridname` *(text)*
: A string that uniquely identifies the grid for which this gmetad instance is responsible. This string should be different from the one set in gmond. The one set in *gmond.conf* (at cluster { name = "XXX" }) is used in the `CLUSTER` tag that wraps all the hosts that particular gmond instance has collected. The `gridname` attribute will wrap all data sources specified in a `GRID` tag, which could be thought of as a collection of clusters defined in the `data_source`.

`authority` *(URL)*
: The authority URL for this grid. Used by other gmetad instances to locate graphs for this instance's data sources. By default, this value points to "http://hostname/ganglia/".

`trusted_hosts` *(text)*
: A space-separated list of hosts with which this gmetad instance is allowed to share data. `Localhost` is always trusted.

`all_trusted` *(on|off)*
: Set this value to `on` to override the `trusted_hosts` attribute and allow data sharing with any host.

`setuid_username` *(UID)*
: The name of the user gmetad will set the UID to after launch. This defaults to `nobody`.

`setuid` *(on|off)*
: Set this to `off` to disable setuid.

`xml_port` *(number)*
: The gmetad listen port. This value defaults to 8651.

`interactive_port` *(number)*
: The gmetad interactive listen port. This value defaults to 8652.

`server_threads` *(number)*
: The number of simultaneous connections allowed to connect to the listen ports. This value defaults to 4.

`case_sensitive_hostnames` *(1|0)*
: In earlier versions of gmetad, the RRD files were created with case-sensitive hostnames, but this is no longer the case. Legacy users who wish to continue to use RRD files created by Ganglia versions before 3.2 should set this value to 1. Since Ganglia 3.2, this value has defaulted to 0.

RRDtool attributes. Several attributes affect the creation and handling of RRD files.

`RRAs` *(text)*
: These specify custom Round Robin Archive values. The default is (with a "step size" of 15 seconds):

```
"RRA:AVERAGE:0.5:1:5856" "RRA:AVERAGE:0.5:4:20160" "RRA:AVERAGE:0.5:40:52704"
```

: The full details of an RRA specification are contained in the manpage for `rrdcreate`(1).

`umask` *(number)*
: Specifies the umask to apply to created RRD files and the directory structure containing them. It defaults to 022.

`rrd_rootdir` *(path)*
: Specifies the base directory where the RRD files will be stored on the local filesystem.

Graphite support. It is possible to export all the metrics collected by gmetad to Graphite, an external open source metrics storage and visualization tool, by setting the following attributes.

`carbon_server` *(address)*
: The hostname or IP of a remote carbon daemon.

`carbon_port` *(number)*
: The carbon port number, which defaults to 2003.

`graphite_prefix` *(text)*
: Graphite uses dot-separated paths to organize and refer to metrics, so it is probably desirable to prefix the metrics from gmetad with something descriptive like *`data center1.gmetad`*, so Graphite will organize them appropriately.

`carbon_timeout` *(number)*
: The number of milliseconds gmetad will wait for a response from the Graphite server. This setting is important because gmetad's carbon sender is not threaded and will block waiting on a response from a down carbon daemon. Defaults to 500.

gmetad interactive port query syntax. As mentioned previously, gmetad listens on TCP port 8652 (by default) for interactive queries. The interactive query functionality enables client programs to get XML dumps of the state of only the portion of the Grid in which they're interested.

Interactive queries are performed via a text protocol (similar to SMTP or HTTP). Queries are hierarchal, and begin with a forward slash (/). For example, the following query returns an XML dump of the entire grid state:

```
/
```

To narrow the query result, specify the name of a cluster:

```
/cluster1
```

To narrow the query result further, specify the name of a host in the cluster:

```
/cluster1/host1
```

Queries may be suffixed with a filter to modify the type of metric information returned by the query (as of this writing, `summary` is the only filter available). For example, you can request only the summary metric data from `cluster1` like so:

```
/cluster1?filter=summary
```

gweb

Of the three daemons that comprise Ganglia, gweb is both the most configurable, and also least in need of configuration. In fact, there is no need to change anything whatsoever in gweb's default configuration file to get up and running with a fully functional web UI.

Apache virtual host configuration

Although gweb itself requires no configuration to speak of, some web server configuration is necessary to get gweb up and running. Any web server with PHP support will

do the job, and although web server configuration is beyond of the scope of this book, the Apache Web Server is such a common choice that it has been included an example of a virtual host configuration for Apache. Assuming that gweb is installed in */var/www/html/ganglia2* on a host that resolves to `myganglia.example.org`, the following configuration should get you started with Apache:

```
NameVirtualHost *:80

<VirtualHost *:80>
   ServerName myganglia.example.org
   ServerAlias myganglia

   DocumentRoot /var/www/html/ganglia2

   # Other directives here
</VirtualHost>
```

This is, of course, a simplistic example. For further reading on the subject, we recommend further reading here.

gweb options

gweb is configured by way of the *conf.php* file. In fact, this file overrides and extends the default configuration set in *conf_default.php*. *conf.php* is located in the web root directory. This file is well documented, and as of this writing there are more than 80 options, so it won't cover them all, but it will cover some of the more important ones, and make note of some option categories—just so you're aware they're there.

The file, as its name suggests, is itself a PHP script made up of variable assignments. Unlike the other configuration files, assignments might span multiple lines. Attribute names are themselves keys in gweb's `$conf` data structure, so they are case sensitive, and look like PHP array assignments. The following line, for example, informs gweb of the location of the RRDtool binary:

```
$conf['rrdtool'] = "/usr/bin/rrdtool";
```

All attributes in the file are required, and some may be defined multiple times; others must appear only once. Some values are derived from other values. For example, the `rrds` attribute is derived from `gmetad_root`:

```
$conf['rrds'] = "${conf['gmetad_root']}/rrds";
```

Application settings. Attributes in this category affect gweb's functional parameters—its own home directory, for example, or the directories in which it will check for RRDs or templates. These are rarely changed by the user, but a few of them bear mentioning.

`templates` *(path)*
: Specifies the directory in which gweb will search for template files. Templates are like a skin for the site that can alter its look and feel.

`graphdir` *(path)*
: Specifies the directory where the user may drop JSON definitions of custom graphs. As described in the next chapter, users may specify custom report graphs in JSON format and place them in this directory, and they will appear in the UI.

`rrds` *(path)*
: Specifies the directory where the RRD files are to be found.

As described in Chapter 7, various Nagios integration features may be set in gweb's *conf.php*. Collectively, these enable Nagios to query metric information from gweb instead of relying on remote execution systems such as Nagios Service Check Acceptor (NSCA) and Nagios Remote Plugin Executor (NRPE).

Look and feel. gweb may be configured to limit the number of graphs it displays at once (`max_graphs`) and to use a specified number of columns for the grid and host views. There are also a number of boolean options that affect the default behavior of the UI when it is first launched, such as `metric_groups_initially_collapsed` .

The *config.php* file defines numerous settings that modify the functional attributes of the graphs drawn in the UI. For example, you may change the colors used to plot the values in the built-in load report graph and the default colors used in all the graphs and even define custom time ranges.

Security. Attributes in this category include the following:

`auth_system` *(readonly|enabled|disabled)*
: gweb includes a simple authorization system to selectively allow or deny individual users access to specific parts of the application. This system is enabled by setting `auth_system` to `enabled`. For more information on the authorization features in gweb, see Chapter 4.

Advanced features. Here are some advanced features:

`rrdcached_socket` *(path)*
: Specifies the path to the rrdcached socket. rrdcached is a high-performance caching daemon that lightens the load associated with writing data to RRDs by caching and combining the writes. More information may be found in Appendix A.

`graph_engine` *(rrdtool|graphite)*
: gweb can use Graphite instead of RRDtool as the rendering engine used to generate the graphs in the UI. This approach requires you to install patched versions of whisper and the Graphite webapp on your gweb server. More information can be found here.

Postinstallation

Now that Ganglia is installed and configured, it's time to get the various daemons started, verify that they're functional, and ensure that they can talk to each other.

Starting Up the Processes

Starting the processes in a specific order is not necessary; however, if the daemons are started in the order recommended here, there won't be a delay waiting for metadata to be retransmitted to the UDP aggregator and users won't get error pages or incomplete data from the web server:

1. If you're using the UDP unicast topology, start the UDP aggregator nodes first. This ensures that the aggregator nodes will be listening when the other nodes send their first metadata transmission.
2. Start all other gmond instances.
3. If you're using rrdcached, start all rrdcached instances.
4. Start gmetad instances at the lowest level of the hierarchy (in other words, gmetad instances that don't poll any other gmetad instances).
5. Work up the hierarchy starting any other gmetad instances.
6. Start the Apache web servers. Web servers are started after gmetad; otherwise, the PHP scripts can't contact gmetad and the users see errors about port 8652.

Remember rrdcached

If gmetad is configured to use rrdcached, it is essential for rrdcached to be running before gmetad is started.

Testing Your Installation

gmond and gmetad both listen on TCP sockets for inbound connections. To test whether gmond is operational on a given host, `telnet` to gmond's TCP port:

```
user@host:$ telnet localhost 8649
```

In reply, gmond should output an XML dump of metric data. If the gmond is deaf or mute, it may return a rather empty XML document, with just the `CLUSTER` tag. gmetad may be likewise tested with telnet like so:

```
user@host:$ telnet localhost 8651
```

A functioning gmetad will respond with an XML dump of metric data.

See Chapter 6 for a more comprehensive list of techniques for validating the state of the processes.

Firewalls

Firewall problems are common with new Ganglia installations that span network subnets. Here, we've collected the firewall requirements of the various daemons together to help you avoid interdaemon communication problems:

1. gmond uses multicast by default, so clusters that span a network subnet need to be configured with unicast senders and listeners as described in the previous topology sections. If the gmond hosts must traverse a firewall to talk to each other, allow udp/8649 in both directions. For multicast, support for the IGMP protocol must also be enabled in the intermediate firewalls and routers.
2. gmond listens for connections from gmetad on TCP port 8649. If gmetad must traverse a firewall to reach some of the gmond nodes, allow tcp/8649 inbound to a few gmond nodes in each cluster.
3. gmetad listens for connections on TCP port 8651 and 8652. The former port is analogous to gmond's 8649, while the latter is the "interactive query" port to which specific queries may be sent. These ports are used by gweb, which is usually installed on the same host as gmetad, so unless you're using some of the advanced integration features, such as Nagios integration, or have custom scripts querying gmetad, you shouldn't need any firewall ACLs for gmetad.
4. gweb runs in a web server, which usually listens on ports 80 and 443 (if you enable SSL). If the gweb server is separated from the end users by a firewall (a likely scenario), allow inbound tcp/80 and possibly tcp/443 to the gweb server.
5. If your Ganglia installation uses sFlow collectors and the sFlow collectors must traverse a firewall to reach their gmond listener, allow inbound udp/6343 to the gmond listener.

CHAPTER 3

Scalability

Daniel Pocock and Bernard Li

Who Should Be Concerned About Scalability?

Scalability is discussed early in this book because it needs to be factored in at the planning stage rather than later on when stability problems are observed in production.

Scalability is not just about purchasing enough disk capacity to store all the RRD files. Particular effort is needed to calculate the input/output operations per second (IOPS) demands of the running gmetad server. A few hours spent on these calculations early on can avoid many hours of frustration later.

The largest Ganglia installation observed by any of the authors is a tier-1 investment bank with more than 50,000 nodes. This chapter is a must-read for enterprises of that size: without it, the default Ganglia installation will appear to be completely broken and may even flood the network with metric data, interfering with normal business operations. If it's set up correctly (with a custom configuration), the authors can confirm that Ganglia performs exceptionally well in such an environment.

In fact, the number of nodes is not the only factor that affects scalability. In a default installation, Ganglia collects about 30 metrics from each node. However, third-party metric modules can be used to collect more than 1,000 metrics per node, dramatically increasing the workload on the Ganglia architecture. Administrators of moderate-sized networks pursuing such aggressive monitoring strategies need to consider scalability in just the same way as a large enterprise with a huge network does.

For a small installation of a few dozen hosts, this chapter is not required. It is quite possible that the default installation will "just work" and never require any attention.

gmond and Ganglia Cluster Scalability

The default multicast topology is often quite adequate for small environments in which the number of sender/receiver nodes is not too large. For larger environments, an

analysis of scalability is suggested. It is necessary to consider how the receiving nodes are affected as the network is scaled:

Memory impact
: The more nodes and the more metrics per node, the greater the memory consumption on those gmond processes that are configured to receive metrics from other nodes.

CPU impact
: gmond is bound by a single thread. The faster the rate of metrics arriving from the network, the more the CPU core is utilized. The metric arrival rate depends on three things: the number of nodes, the number of metrics per node, and the rate at which the nodes are configured to transmit fresh values.

gmond has to process all the metric data received from the other nodes in the cluster or grid. As the data volume increases—due to more nodes joining the cluster, or a higher rate of metric transmissions—every node must allocate more and more CPU time to process the incoming data, which can be undesirable and unnecessary.

To reduce this overhead, the solution is the use of "deaf" nodes, as illustrated in the deaf/mute node topology of Figure 2-2.

gmetad Storage Planning and Scalability

In a small Ganglia installation, the default gmetad configuration (storing the RRD files on disk, not using rrdcached in any way) will just work. No special effort needs to be made and this section can be skipped.

If you have a large installation, it is important to read this section carefully before proceeding to configure gmetad.

RRD File Structure and Scalability

An RRD file is made up of a header followed by a series of one or more arrays. If the entire RRD file fits into one block on disk (4,096 bytes, typically) then all modifications to the file can, at best, be made with a single IO request.

The actual amount of data stored for a single data point in the RRD file is 8 bytes. If data points are stored at 60-second intervals, a single 4,096-byte disk block will store just over eight hours of data points.

With a single-data source (DS) RRD file, an update that writes to only one RRA will change only 8 bytes of data in the entire 4K page. Furthermore, for the operating system to make this write, it first has to have the entire page/block in RAM. If it is not in RAM, there is a read IO before the write IO, a total of two IOs for one write. Having enough physical RAM for the page cache to retain all the currently active blocks from each RRD can avoid this double IO problem, reducing the IO load by 50 percent.

If the RRD file is keeping data for an extended period of time, or keeping different types of data (both MIN and MAX values), it may span more than one block and multiple IOs are required when a single update occurs. This issue can be compounded by the read-before-write problem just described.

The RRD file structure interleaves the data from different data sources but not from different consolidation functions (such as MIN and MAX values). In RRDtool deployments that store data from multiple sources in a single RRD file, this means that all the values for a particular time interval are likely to be stored in the same disk block accessible with a single disk IO read or write operation. However, this optimization is not of any benefit with Ganglia: because Ganglia supports a variable number of data sources for each host and RRD files are inherently static in structure, Ganglia creates a different RRD file for each metric, rather than creating multiple data sources within a single RRD file.

If more than one consolidation function is used (for example, MIN, MAX, and AVERAGE values are retained for a single data source), these values are not interleaved. The RRD file contains a separate array for each function. Theoretically, if the sample interval and retention periods were equivalent, such values could be interleaved, but this approach is not currently supported by RRDtool.

Although interleaving is significantly more efficient for writing, it has the slight disadvantage that reading/reporting on a single consolidation function for a single source has to read all the interleaved data, potentially multiplying the number of blocks that must be read. In many situations, the write workload is much heavier than the reporting workload, so this read overhead of interleaving is far outweighed by the benefits of the strategy.

When data is recorded for different step values (e.g., 8 hours of samples at a 20-second interval and 7 days of samples at a 15-minute interval), there is no way to interleave the data with different step values. In this example, on every 60th write to the array of 20-second samples, there will also be a write to the array of 15-minute samples. It is likely that the array of 15-minute samples is in the second or third disk block of the file, so it is an extra write IO operation, it requires extra space in the page cache, and it may not be a contiguous block suitable for a sequential read or write. This may not seem like a major overhead for one write out of every 60, until you contemplate the possibility that every RRD file may simultaneously make that extra write IO operation at the same time, stressing the write IO system and page cache in order to handle the amount of IO that is processed for every other sample. In fact, RRDtool's designers anticipated this nightmare scenario, and every RRD file has a randomized time offset to ensure that the IO workload is spread out. Nonetheless, it is still important to bear in mind the extra IO effort of accessing multiple regions in the RRD file.

Understanding the file format at this level is essential to anyone involved in an extreme Ganglia deployment. RRDtool is an open source project with an active community

behind it, and more demanding users are encouraged to join the RRDtool mailing list to discuss such issues and strategies for dealing with scalability.

In conclusion, when deciding how to configure the RRA parameters in *gmetad.conf*, it is essential to consider not only the disk space but also the extra IO demand created by every consolidation function and sample interval.

Acute IO Demand During gmetad Startup

The first thing to be aware of is the startup phase. If gmetad stops for many hours, then on the next startup, it has to fill in the gaps in the RRD files. gmetad and RRDtool fill such gaps by writing the value NaN into each timeslot. This is a common scenario if a server goes offline for a 12-hour weekend maintenance window, or if a server crashes at midnight and is started up again by staff arriving eight hours later. If the RRD parameters are configured to retain data with short sample intervals for many hours, this situation implies a lot of gaps to be filled.

During the startup phase, gmetad and RRDtool are trying to write to RRD files from each cluster in parallel. This means that the disk write activity is not sequential.

Consequently, if there is a truly huge number of RRD files and if the disk storage system does not cope well with a random access workload (e.g., if it is not an SSD or SAN), the startup phase may take many hours or may not even complete at all. During this phase, no new data is collected or stored. Sometimes, the only way to bypass this startup issue is to delete all the existing RRDs (losing all history) and start afresh. Therefore, when planning for scalability of the storage system, this startup phase should be considered the worst-case scenario that the storage must support. Decisions about the storage system, sample intervals, and retention periods must be carefully considered to ensure a smooth startup after any lengthy period of downtime.

gmetad IO Demand During Normal Operation

After the startup phase, it is also necessary to consider the IO demands during normal operation:

Real-time write workload
: If each RRD file spans more than one disk block, then storing an update to the file is likely to involve multiple nonsequential writes: one write to the header (first block), one write to each RRA, and one write to the file access time (directory-level metadata). If the RRD stores MIN, MAX, and AVERAGE values, and there are RRAs for 1-minute, 15-minute, and 1-hour samples, that is 9 distinct RRAs and therefore 11 write operations. Moving to the next RRD file is likely to be nonsequential, as gmetad does not actively order the files or the writes in any particular way. The gmetad processes runs a separate thread for each Ganglia cluster. Thus, there can be concurrent write access. This combination of nonsequential writing

and concurrent access creates a pattern of disk write access that appears almost truly random.

Reading to generate web graphs/reports
: Each time the user accesses a page in the web UI, the web server reads from every RRD that is needed (maybe 30 separate files for a host view) to create the necessary graphs. Due to the multithreaded nature of the web browser and web server, the read workload is likely to be random access.

Reading for reporting
: If any batch reports are executed on an hourly or daily basis, the workload may be sequential or random access, depending on the amount of data extracted by the report and the way the reporting script is written.

Due to the relatively simple architecture of Ganglia and RRDtool, they are inherently very scalable and adaptable. Keep the previous points in mind as you design your Ganglia deployment, but rest assured that the lightweight and efficient architecture has been scrutinized by many talented engineers and there's not much to improve upon. As long as the workload is relatively constant (a constant number of hosts in the network), the IO workload will also remain somewhat constant. There are clear solutions for this type of workload, which are covered extensively in the rest of this chapter.

When you consider all of these IO demands together, it should become obvious that careful planning and real-time monitoring of the gmetad servers is needed.

Forecasting IO Workload

Before committing to a large installation of Ganglia, it is desirable to forecast the IO workload. Here is a brief outline of how to make an accurate forecast:

1. Decide on the exact RRA structure you require (sample intervals, retention periods, and functions such as MIN, MAX, or AVERAGE).
2. Create a logical volume big enough for one RRD file (4 MB should be fine). Format the logical volume and mount it. Create one sample RRD file on the new filesystem.
3. Use `iostat` to observe write IO on the logical volume:

   ```
   $ iostat -k 60 -x /dev/dm-X
   ```

   where `X` is replaced by the actual device node.
4. Use the `rrdupdate` command (`man rrdupdate`) to manually write updates to the RRD file while observing the `iostat` IO levels. The simulation should generate enough writes to span all the sample intervals. For example, if keeping 60-second samples for 8 hours and 15-minute samples for 1 week, the simulation should generate 30 minutes worth of rrdupdates to fill two of the 15-minute samples:

```
# start time
TS=`date +%s`

# interval between simulated polls
STEP=30

# duration of test (not real time)
PERIOD=`expr 60 '*' 30`

LAST_TS=`expr $TS + $PERIOD`
while [ $TS -lt $LAST_TS ];
do
  rrdtool update /mnt/test_lv/testing.rrd ${TS}@0.0
  TS=`expr $TS + $STEP`
done
```

5. Look at the output from the `iostat` command to see how many read and write IOPS occurred. Look for the IOPS count and not the rate.
6. Divide the IOPS count by the value of `$PERIOD` to find the actual IOPS rate for a system running in real time (the simulation runs faster than real time, so the rate reported by `iostat` is meaningless). Multiply that by the anticipated number of hosts and metrics to estimate the real IO workload.
7. Repeat the simulation with rrdcached.

Testing the IO Subsystem

The previous section described how to estimate the required IO throughput (measured in IOPS). It is important to verify that the purchased disk system/SAN can support the maximum throughput demands.

In many corporate environments, there is a standard SAN solution for all users. The SAN may or may not be configured to operate at the IO levels claimed by the manufacturer. Existing applications on the SAN may already be using a significant portion of its IO capacity. Therefore, it is necessary to run a test to verify the actual performance that the SAN can offer.

It is a good idea to run such testing several times during the day and night to observe whether the SAN provides consistent performance. For example, the SAN may be slower on Sundays due to IO-heavy backup/replication tasks. It may run slowly between 7:00 am and 8:00 am due to a daily reporting batch that hammers multiple databases. The only way to discover these patterns is to run a simulation every 15 minutes for a week.

There are many tools for testing disk IO capacity. The example presented here is for the flexible IO tester tool (`man fio`). It is available as the package fio on Debian.

First, create a configuration file for the tool, a sample of which is shown here:

```
# Do some important numbers on SSD drives, to gauge what kind of
# performance you might get out of them.
#
# Sequential read and write speeds are tested, these are expected to be
# high. Random reads should also be fast, random writes are where crap
# drives are usually separated from the good drives.
#
# This uses a queue depth of 4. New SATA SSD's will support up to 32
# in flight commands, so it may also be interesting to increase the queue
# depth and compare. Note that most real-life usage will not see that
# large of a queue depth, so 4 is more representative of normal use.
#
[global]
bs=4k
ioengine=libaio
iodepth=1
size=1g
direct=1
runtime=60
directory=/mnt/san_volume
filename=fio.test.file.delete_me

[seq-read]
rw=read
stonewall

[rand-read]
rw=randread
stonewall

[seq-write]
rw=write
stonewall

[rand-write]
rw=randwrite
stonewall
```

There are two things to consider changing:

`directory=/mnt/san_volume`
: Change to the actual mount point of a SAN filesystem.

`size=1g`
: Increasing this value could make the test span more disks in the SAN, which makes the test output more reliable, but it also makes the test take longer.

Now run the test (where *config.fio* is the config file name):

```
fio --output=san_test.out config.fio
```

The output report (*san_test.out*) is created in the current directory. Here is a sample of how it might look:

```
...

rand-write: (groupid=3, jobs=1): err= 0: pid=23038
  write: io=46512KB, bw=793566B/s, iops=193, runt= 60018msec
    slat (usec): min=13, max=35317, avg=97.09, stdev=541.14
    clat (msec): min=2, max=214, avg=20.53, stdev=18.56
     bw (KB/s) : min=    0, max=  882, per=98.54%, avg=762.72, stdev=114.51
  cpu          : usr=0.85%, sys=2.27%, ctx=11972, majf=0, minf=21
  IO depths    : 1=0.1%, 2=0.1%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued r/w: total=0/11628, short=0/0

     lat (msec): 4=1.81%, 10=32.65%, 20=31.30%, 50=26.82%, 100=6.71%
     lat (msec): 250=0.71%

...

Run status group 3 (all jobs):
  WRITE: io=46512KB, aggrb=774KB/s, minb=793KB/s, maxb=793KB/s, mint=60018msec, ...
Disk stats (read/write):
  dm-23: ios=277015/36964, merge=0/0, ticks=277884/471056, in_queue=749060, ...
    md2: ios=0/0, merge=0/0, ticks=0/0, in_queue=0, util=-nan%, aggrios=0/0, ...
    sda: ios=0/0, merge=0/0, ticks=0/0, in_queue=0, util=-nan%
```

In this sample, much of the output is truncated, so that only the `rand-write` job output is displayed. This job gives the best simulation of gmetad/RRDtool write behavior.

The key point to look at is the IOPS throughput rate, `iops=193` in the sample. Compare this to the IOPS estimate calculated in the section "Forecasting IO Workload" on page 47 to find out whether the SAN will cope.

If you find that the IOPS capacity of the SAN is insufficient, consider options such as reducing the number of RRAs for different functions (e.g., keep only MAX and not MIN or AVERAGE), get a faster SAN, or review the list of suggestions in the next section.

Dealing with High IO Demand from gmetad

Here are some common strategies:

Store RRD files on fast disks (RAID0, SAN, SSD).
: By default, RRD files are stored in */var/lib/ganglia/rrds*. This directory usually resides on the local disk of the server that runs gmetad. If you are experiencing slowdowns on your gmetad servers, consider purchasing a RAID system and storing the RRD files there. Alternatively, good experiences have been reported by users placing RRD files on SAN storage or solid-state drives (SSDs).

Use a disk controller with a large hardware write cache.
: Many RAID controllers now offer a 512 MB or 1 GB write cache. Furthermore, these often have a battery-backup or flash-backup option that is essential for reliability with such a large write cache.

Don't use NFS.
: NFS has a very strict protocol for ensuring that writes are fully committed to disk. Although this approach is very reliable (and necessary) in a client/server model, it is inappropriate for applications that have a heavy IO demand, particularly if the application is accessing multiple files.

Mount the filesystem with the `noatime` *option.*
: When a user invokes a web view that contains many graphs, many different RRD files are accessed and `atime` is updated for every RRD file (metadata write operation). If it is a cluster view and the files are from different hosts, then each metadata access is on a different directory, so a read operation actually involves a massive nonsequential write operation at the same time. On a RAID5 system, this issue is compounded by the R-W-R algorithm (one logical IOP = three physical IOPS). Therefore, enabling the `noatime` option can eliminate a huge amount of unnecessary IO and have a huge benefit.

Store RRD files on RAM disk.
: Similar to the previous solution, one could create a RAM disk using the physical memory available on the gmetad server. For instance, if your server has 8 GB of RAM, you could dedicate 4 GB specifically for storing RRD files, which could be accomplished by adding the following line in */etc/fstab* on a Linux system:

```
tmpfs /var/lib/ganglia/rrds tmpfs defaults 0 0
```

  and mounting the directory accordingly. By default, half of the physical memory will be used by tmpfs.

  Because anything that is stored on tmpfs is volatile, the downside of this solution is that you will need a method of regularly synchronizing the data on tmpfs to persistent disk. The interval of backup should be determined by how much data you are willing to lose if the server crashes unexpectedly. You'll also need to set up some script to automatically preload the data stored on persistent disk back to tmpfs after each reboot prior to gmetad starting up.

Calculate your expected IOPS workload and test the storage.
: Considering the size of your RRD files, the number of RRAs, sample intervals, and retention periods, you can estimate the rate of IOPS.

  Use a tool such as the flexible IO tester (`man fio`) to verify that your block device supports the target IO rate.

Use `rrdcached`.
: From the `rrdcached` manpage, `rrdcached` "is a daemon that receives updates to existing RRD files, accumulates them, and, if enough have been received or a defined time has passed, writes the updates to the RRD file." This daemon comes

with RRDtool 1.4 and later and has been used very successfully in boosting scalability.

For instructions on how to set up `rrdcached` with Ganglia, please refer to "Configuring gmetad for rrdcached" on page 211.

Set up a reverse proxy.
: Use a web proxy such as Squid to cache the graphs so that the same graphs are not constantly regenerated for multiple users. Doing so can dramatically reduce the IO read workload.

Make data available through pregenerated reports.
: Set up cron jobs to prepare static HTML reports at desired intervals. Give users access to the static HTML reports rather than direct access to gweb.

Make sure there is sufficient physical RAM.
: When updating RRDs, `rrdtool` must read and write entire disk blocks rather than just writing out the bytes that change. Consequently, having enough RAM available for the page cache to buffer all active disk blocks (at least one 4,096-byte block per RRD file) can avoid the need for `rrdtool` to read blocks from disk before the write IO is executed.

CHAPTER 4

The Ganglia Web Interface

Vladimir Vuksan and Alex Dean

So far, this book has dealt with the collection of data. Now we will discuss visualizing it. Visualization of these data is the primary responsibility of a web-based application known as gweb. This chapter is an introduction to gweb and its features. Whether the job is understanding how a problem began in your cluster or convincing management that more hardware is required, a picture is worth a thousand data points.

Navigating the Ganglia Web Interface

gweb is organizaed into a number of top-level tabs: Main, Search, Views, Aggregated Graphs, Compare Hosts, Events, Automatic Rotation, Live Dashboard, and Mobile. These tabs allow you to easily jump right to the information you need.

The gweb Main Tab

gweb's navigation scheme is organized around Ganglia's core concepts: grids, clusters, and nodes. As you click deeper into the hierarchy, breadcrumb-style navigation links allow you to return to higher-level views. Figure 4-1 shows how you can easily navigate to exactly the view of the data you want.

Grid View

The grid view (Figure 4-2) provides the highest-level view available. Grid graphs summarize data across all hosts known to a single gmetad process. Grid view is the jumping-off point for navigating into more details displays dealing with individual clusters and the hosts that compose those clusters:

1. Clicking on any grid-level summary graphs brings up the *all time periods* display. Clicking again enlarges the graph you're interested in.
2. Clicking on any cluster-level graph displays the cluster view.

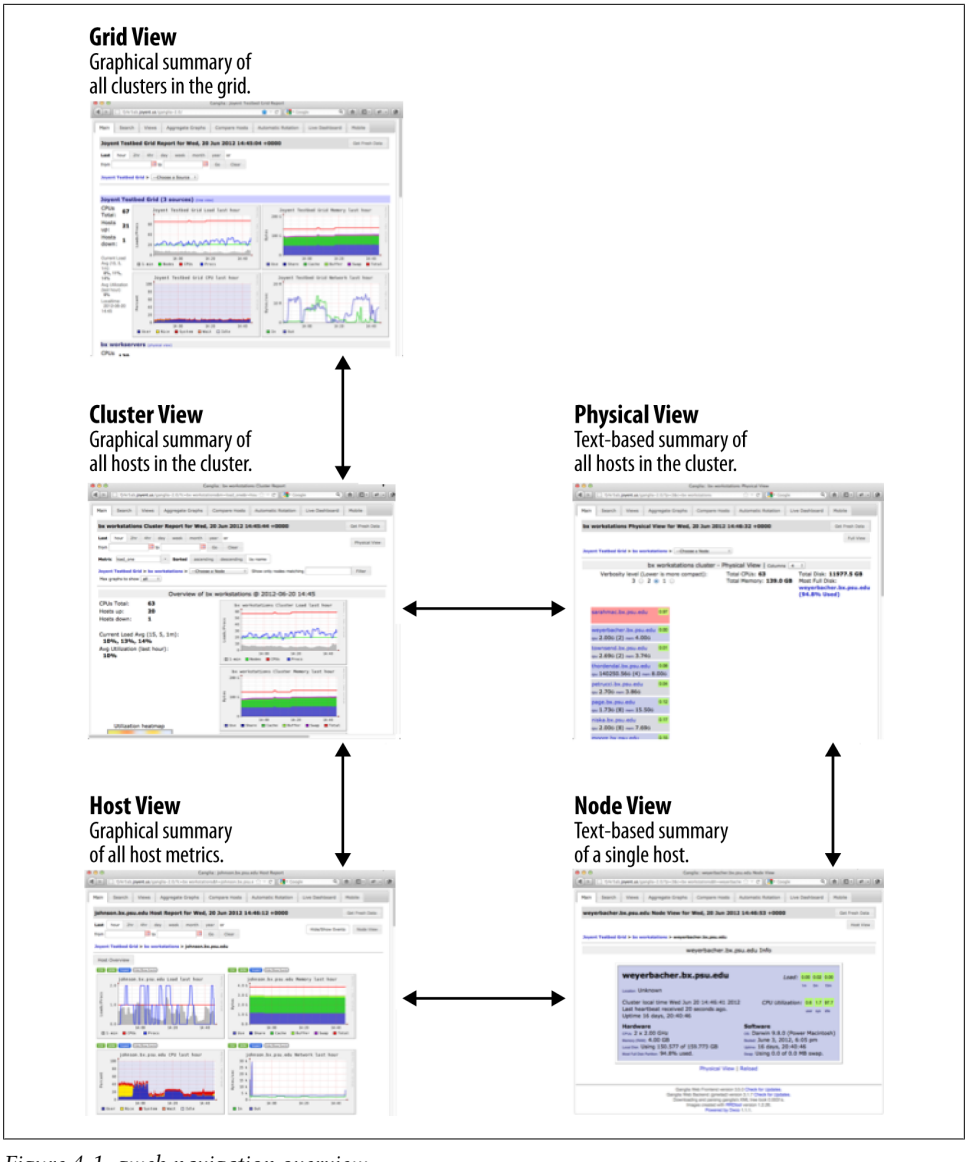


Figure 4-1. gweb navigation overview

Cluster View

A cluster is a collection of gmonds. They may be grouped by physical location, common workload, or any other criteria. The top of the cluster view (Figure 4-3) displays summary graphs for the entire cluster. A quick view of each individual host is further down the page.

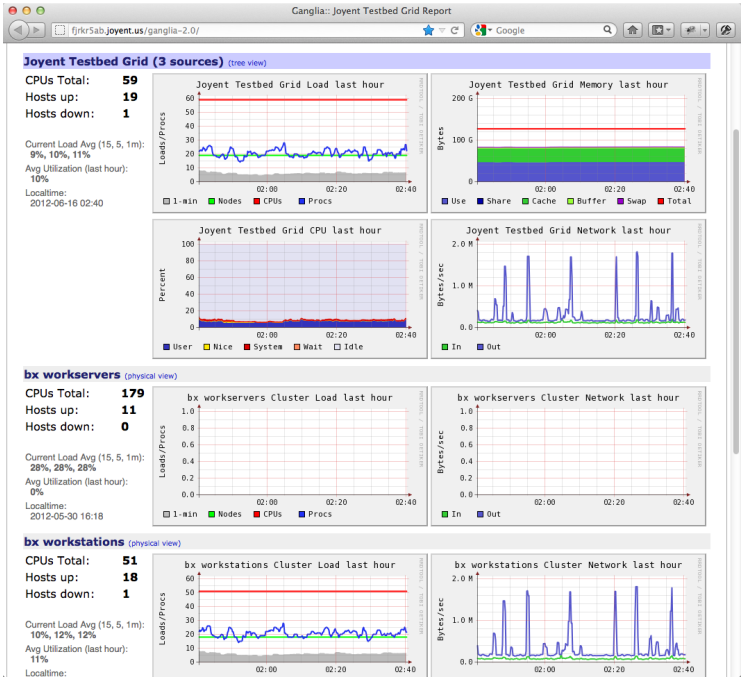


Figure 4-2. Grid view

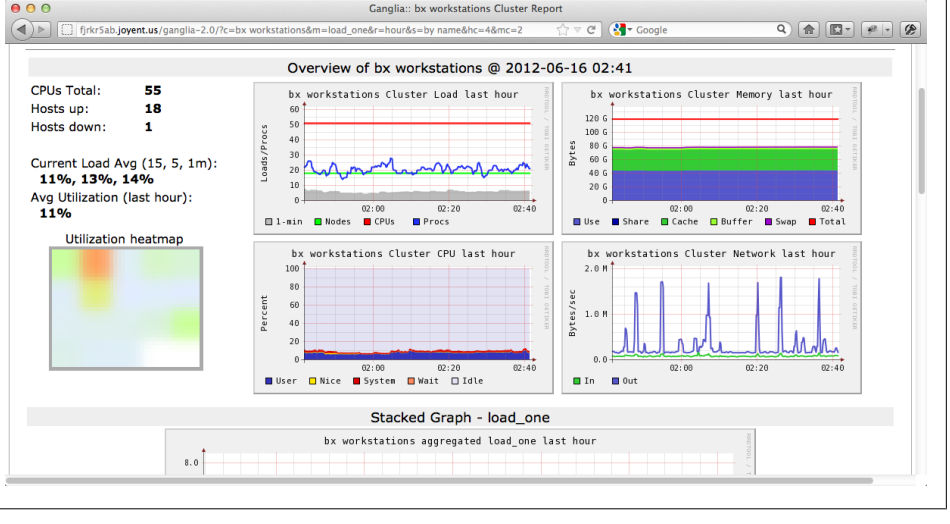


Figure 4-3. Cluster view

1. Clicking on a cluster summary shows you that summary of a range of time periods.
2. Clicking on an individual host takes you to the host display.

The background color of the host graphs is determined by their one-minute load average. The metric displayed for each host can be changed using the Metric select box near the top of the page.

The utilization heatmap provides an alternate display of the one-minute load averages. This is a very quick way to get a feeling for how evenly balanced the workload is in the cluster at the present time. The heatmap can be disabled by setting

```
$conf["heatmaps_enabled"]=0
```

in *conf.php*.

When working with a cluster with thousands of nodes, or when using gweb over a slow network connection, loading a graph for each node in the cluster can take a significant amount of time. `$conf["max_graphs"]` can be defined in *conf.php* to address this problem: to set an upper limit on the number of host graphs that will be displayed in cluster view.

Physical view

Cluster view also provides an alternative display known as *physical view* (Figure 4-4), which is also very useful for large clusters. Physical view is a compressed text-only display of all the nodes in a cluster. By omitting images, this view can render much more quickly than the main cluster view.

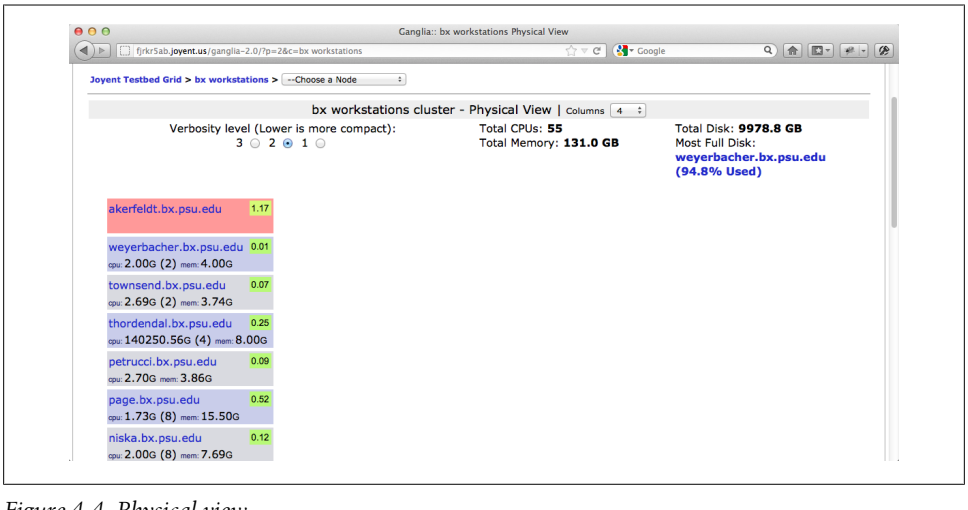


Figure 4-4. Physical view

Clicking on a hostname in physical view takes you to the *node view* for that host. Node view is another text-only view, and is covered in more detail in "Host View" on page 58.

Adjusting the time range

Grid, cluster, and host views allow you to specify the time span (Figure 4-5) you'd like to see. Monitoring an ongoing event usually involves watching the last few minutes of data, but questions like "what is normal?" and "when did this start?" are often best answered over longer time scales.

Figure 4-5. Choosing a time range

You are free to define your own time spans as well via your *conf.php* file. The defaults (defined in *conf_default.php*) look like this:

```
#
# Time ranges
# Each value is the # of seconds in that range.
#
$conf['time_ranges'] = array(
   'hour' => 3600,
   '2hr'  => 7200,
   '4hr'  => 14400,
   'day'  => 86400,
   'week' => 604800,
   'month'=> 2419200,
   'year' => 31449600
);
```

All of the built-in time ranges are relative to the current time, which makes it difficult to see (for example) five minutes of data from two days ago, which can be a very useful view to have when doing postmortem research on load spikes and other problems. The time range interface allows manual entry of begin and end times and also supports zooming via mouse gestures.

In both cluster and host views, it is possible to click and drag on a graph to zoom in on a particular time frame (Figure 4-6). The interaction causes the entire page to reload, using the desired time period. Note that the resolution of the data displayed is limited by what is stored in the RRD database files. After zooming, the time frame in use is reflected in the *custom* time frame display at the top of the page. You can clear this by clicking *clear* and then *go*. Zoom support is enabled by default but may be disabled by setting `$conf["zoom_support"] = 0` in *conf.php*.

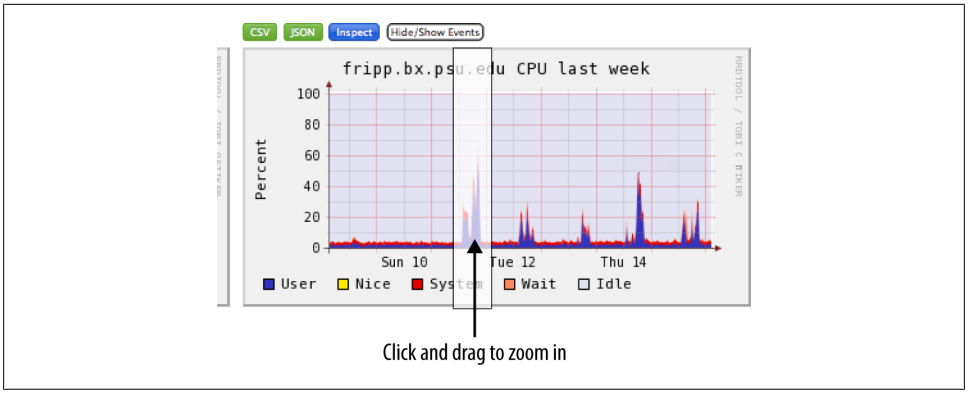


Figure 4-6. Zooming in on an interesting time frame

Host View

Metrics from a single gmond process are displayed and summarized in the host view (Figure 4-7). Summary graphs are displayed at the top, and individual metrics are grouped together lower down.

Host Overview contains textual information about the host, including any string metrics being reported by the host, such as last boot time or operating system and kernel version.

Viewing individual metrics

The "inspect" option for individual metrics, which is also available in the "all time periods" display, allows you to view the graph data interactively:

1. Raw graph data can be exported as CSV or JSON.
2. Events can be turned off and on selectively on all graphs or specific graphs.
3. Trend analysis can make predictions about future metric values based on past data.
4. Graph can be time-shifted to show overlay of previous period's data.

Node view

Node view (Figure 4-8) is an alternative text-only display of some very basic information about a host, similar to the physical view provided at the cluster level.

Graphing All Time Periods

Clicking on a summary graph at the top of the grid, cluster, or host views leads to an "all time periods" view of that graph. This display shows the same graph over a variety of time periods: typically the last hour, day, week, month, and year. This view is very

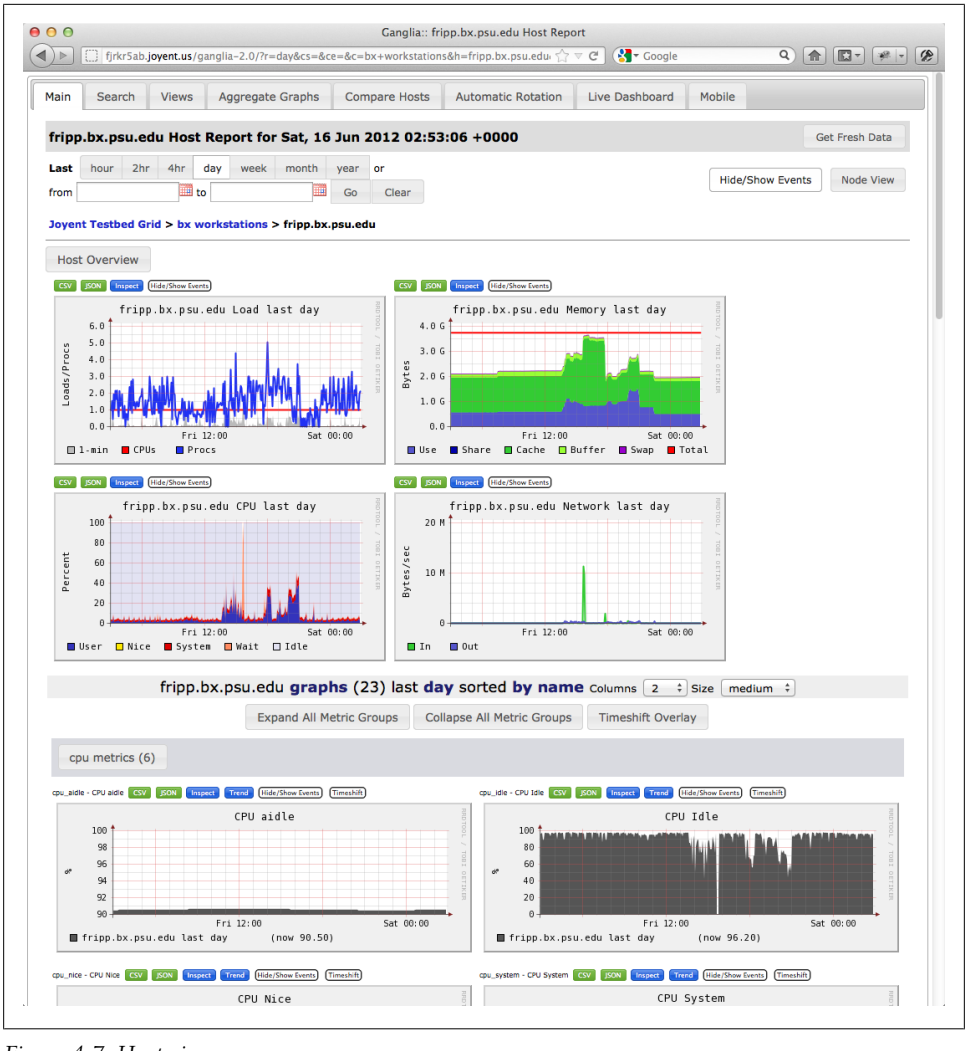


Figure 4-7. Host view

useful when determining when a particular trend may have started or what *normal* is for a given metric.

Many of the options described for viewing individual metrics are also available for all time periods, include CSV and JSON export, interactive inspection, and event display.

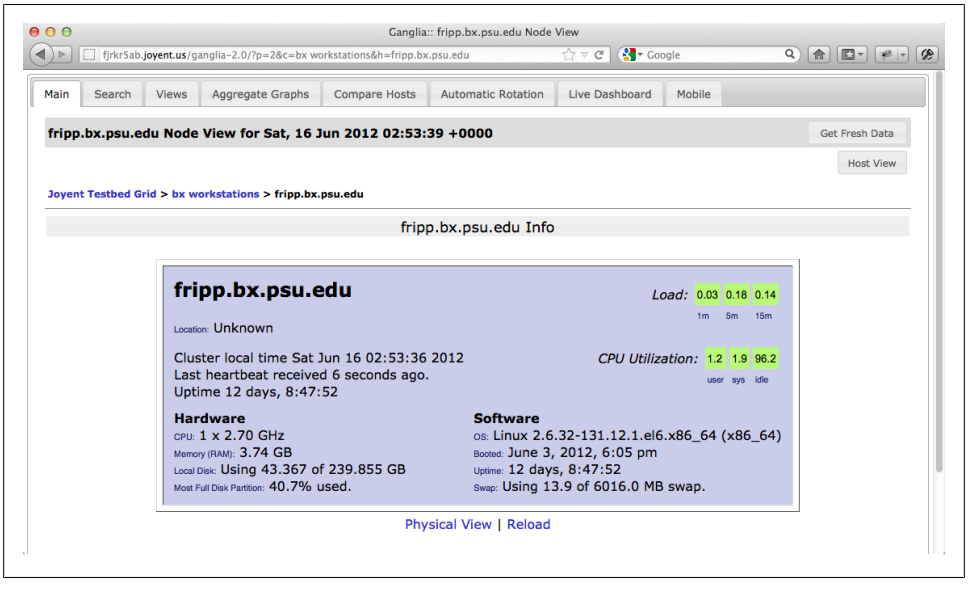


Figure 4-8. Node view

The gweb Search Tab

Search allows you to find hosts and metrics quickly. It has multiple purposes:

- Find a particular metric, which is especially useful if a metric is rare, such as `outgoing_sms_queue`.
- Quickly find a host regardless of a cluster.

Figure 4-9 shows how gweb search autocomplete allows you to find metrics across your entire deployment. To use this feature, click on the Search tab and start typing in the search field. Once you stop typing, a list of results will appear. Results will contain:

- A list of matching hosts.
- A list of matching metrics. If the search term matches metrics on multiple hosts, all hosts will be shown.

Click on any of the links and a new window will open that will take you directly to the result. You can keep clicking on the results; for each result, a new window will open.

The gweb Views Tab

Views are an arbitrary collection of metrics, host report graphs, or aggregate graphs. They are intended to be a way for a user to specify things of which they want to have a single overview. For example, a user might want to see a view that contains aggregate

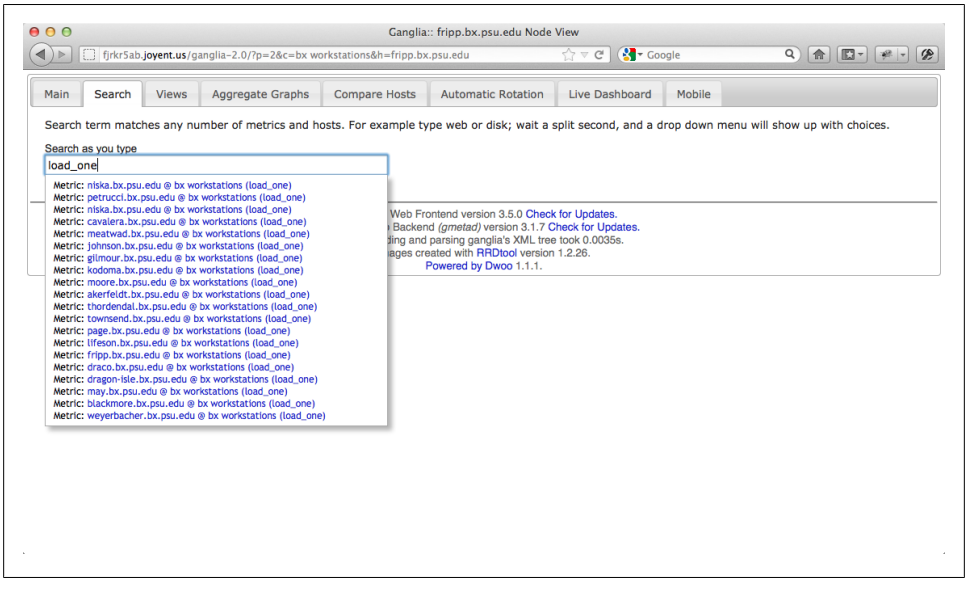


Figure 4-9. Searching for load_one metrics

load on all servers, aggregate throughput, load on the MySQL server, and so on. There are two ways to create/modify views: one is via the web GUI, and the other by programatically defining views using JSON.i

Creating views using the GUI
: To create views click the Views tab, then click Create View. Type your name, then click Create.

Adding metrics to views using the GUI
: Click the plus sign above or below each metric or composite graph; a window will pop up in which you can select the view you want the metric to be added. Optionally, you can specify warning and critical values. Those values will appear as vertical lines on the graph. Repeat the process for consecutive metrics. Figure 4-10 shows the UI for adding a metric to a view.

Defining views using JSON
: Views are stored as JSON files in the *conf_dir* directory. The default for the *conf_dir* is */var/lib/ganglia/conf*. You can change that by specifying an alternate directory in *conf.php*:

```
$conf['conf_dir'] = "/var/www/html/conf";
```

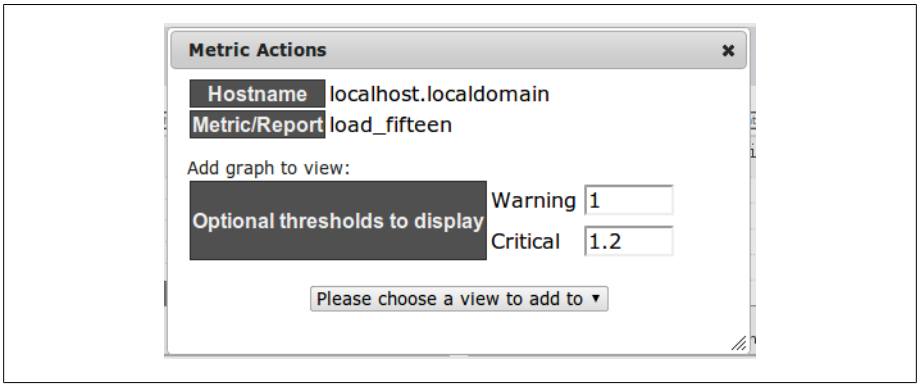


Figure 4-10. Metric actions dialog

You can create or edit existing files. The filename for the view must start with *view_* and end with *.json* (as in, *view_1.json* or *view_jira_servers.json*). It must be unique. Here is an example definition of a view that will result with a view with three different graphs:

```
{
 "view_name":"jira",
 "items":[
  { "hostname":"web01.domain.com","graph":"cpu_report"},
  { "hostname":"web02.domain.com","graph":"load_report"},
  { "aggregate_graph":"true",
      "host_regex":[
        {"regex":"web[2-7]"},
        {"regex":"web50"}
      ],
      "metric_regex":[
        {"regex":"load_one"}
      ],
      "graph_type":"stack",
      "title":"Location Web Servers load"
  }
  ],
  "view_type":"standard"
 }
```

Table 4-1 lists the top-level attributes for the JSON view definition. Each item can have the attributes listed in Table 4-2.

Table 4-1. View items

Key	Value
view_name	Name of the view, which must be unique.
view_type	Standard or Regex. Regex view allows you to specify regex to match hosts.
items	An array of hashes describing which metrics should be part of the view.

Table 4-2. Items configuration

Key	Value
hostname	Hostname of the host that we want metric/graph displayed.
metric	Name of the metric, such as load_one.
graph	Graph name, such as cpu_report or load_report. *You can use metric or graph keys but not both.*
aggregate_graph	If this value exists and is set to true, the item defines an aggregate graph. This item needs a hash of regular expressions and a description.
warning	(Optional) Adds a vertical yellow line to provide visual cue for a warning state.
critical	(Optional) Adds a vertical red line to provide visual cue for a critical state.

Once you compose your graphs, it is often useful to validate JSON—for example, that you don't have extra commas. To validate your JSON configuration, use Python's `json.tool`:

```
$ python -m json.tool my_report.json
```

This command will report any issues.

The gweb Aggregated Graphs Tab

Aggregate graphs (Figure 4-11) allow you to create composite graphs combining different metrics. At a minimum, you must supply a host regular expression and metric regular expression. This is an extremely powerful feature, as it allows you to quickly and easily combine all sorts of metrics. Figure 4-12 includes two aggregate graphs showing all metrics matching host regex of *loc* and metric regex of *load*.

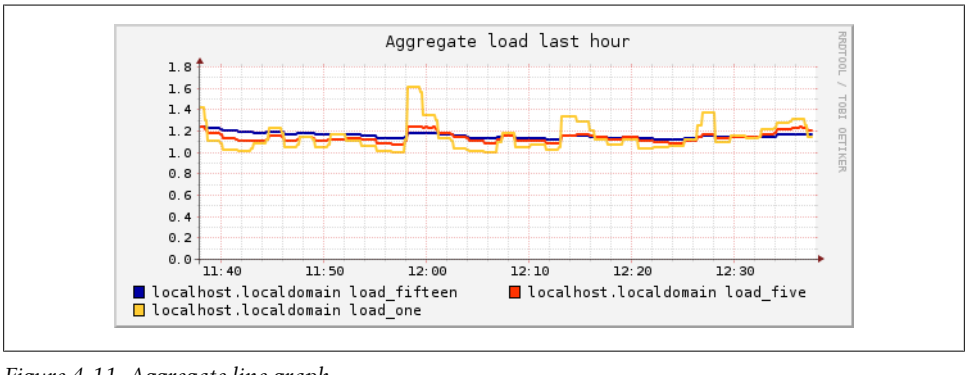


Figure 4-11. Aggregate line graph

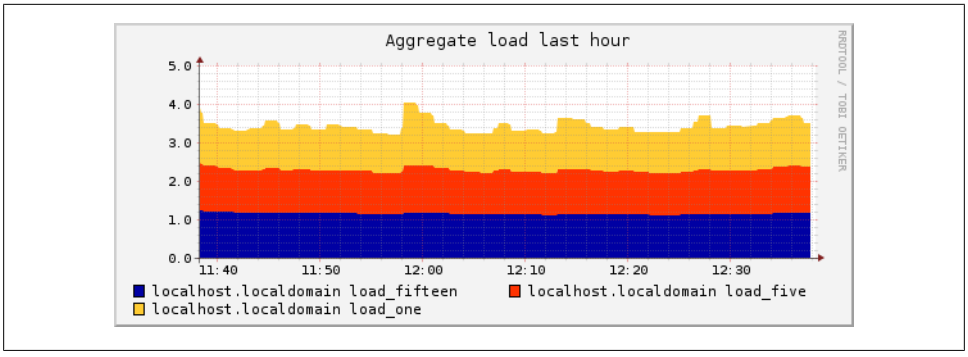


Figure 4-12. Aggregate stacked graph

Decompose Graphs

Related to aggregate graphs are decompose graphs, which decompose aggregate graphs by taking each metric and putting it on a separate graph. This feature is useful when you have many different metrics on an aggregate graph and colors are blending together. You will find the Decompose button above the graph.

The gweb Compare Hosts Tab

The compare hosts feature allows you to compare hosts across all their matching metrics. It will basically create aggregate graphs for each metric. This feature is helpful when you want to observe why a particular host (or hosts) is behaving differently than other hosts.

The gweb Events Tab

Events are user-specified "vertical markers" that are overlaid on top of graphs. They are useful in providing visual cues when certain events happen. For example, you might want to overlay software deploys or backup jobs so that you can quickly associate change in behavior on certain graphs to an external event, as in Figure 4-13. In this example, we wanted to see how increased `rrdcached` write delay would affect our CPU wait IO percentage, so we added an event when we made the change.

Alternatively, you can overlay a timeline to indicate the duration of a particular event. For example, Figure 4-14 shows the full timeline for a backup job.

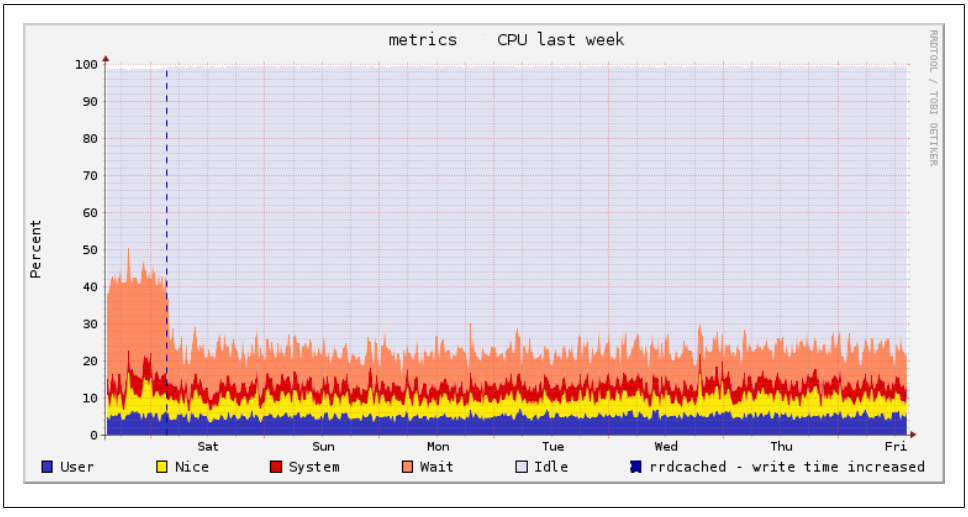


Figure 4-13. Event line overlay

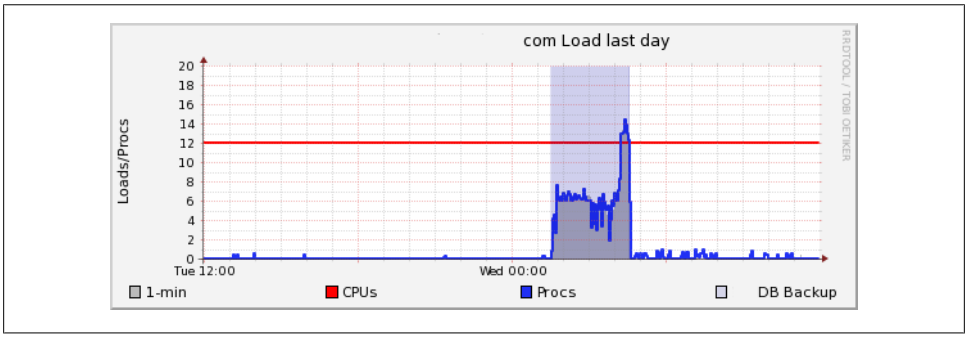


Figure 4-14. Event timeline overlay

By default, Ganglia stores event in a JSON hash that is stored in the *events.json* file. This is an example JSON hash:

```
[
 { "event_id":"1234",
     "start_time":1308496361,
     "end_time":1308496961,
     "summary":"DB Backup",
     "description":"Prod daily db backup",
     "grid":"*",
     "cluster":"*",
     "host_regex":"centos1"
 },
 { "event_id":"2345",
     "start_time":1308497211,
     "summary":"FS cleanup",
```

```
        "grid":"*",
        "cluster":"*",
        "host_regex":"centos1"
    }
]
```

It is also possible to use a different backend for events, which can be useful if you need to scale up to hundreds or thousands of events without incurring the processing penalty associated with JSON parsing. This feature is configured with two configuration options in your *conf_default.php* file. You should have PHP support for MySQL installed on your gweb server before attempting to configure this support. The database schema can be imported from *conf/sql/ganglia.mysql*:

```
# What is the provider used to provide events
# Examples: "json", "mdb2"
$conf['overlay_events_provider'] = "mdb2";
# If using MDB2, connection string:
$conf['overlay_events_dsn'] = "mysql://dbuser:dbpassword@localhost/ganglia";
```

Alternatively, you can add events through the web UI or the API.

Events API

An easy way to manipulate events is through the Ganglia Events API, which is available from your gweb interface at `/ganglia/api/events.php`. To use it, invoke the URL along with key/value pairs that define events. Key/value pairs can be supplied as either GET or POST arguments. The full list of key/value pairs is provided in Table 4-3.

Table 4-3. Events options

Key	Value
action	`add` to add a new event, `edit` to edit, `remove` or `delete` to remove an event.
start_time	Start time of an event. Allowed options are `now` (uses current system time), UNIX timestamp, or any other well-formed date, as supported by PHP's `strtotime` function.
end_time	Optional. Same format as `start_time`.
summary	Summary of an event. It will be shown in the graph legend.
host_regex	Host regular expression, such as `web-|app-`.

Examples

To add an event from your cron job, execute a command such as:

```
curl "http://mygangliahost.com/ganglia/api/events.php?\
  action=add&start_time=now&\
  summary=Prod DB Backup&host_regex=db02"
```

or:

```
curl -X POST --data " action=add&start_time=now\
  &summary=Prod DB Backup&host_regex=db02" \
  http://mygangliahost.com/ganglia/api/events.php
```

API will return a JSON-encoded status message with either `status = ok` or `status = error`.

If you are adding an event, you will also get the `event_id` of the event that was just added in case you want to edit it later, such as to add an `end_time`.

The gweb Automatic Rotation Tab

Automatic rotation is a feature intended for people in data centers who need to continuously rotate metrics to help spot early signs of trouble. It is intended to work in conjunction with views. To activate it, click Automatic Rotation and then select the view you want rotated. Metrics will be rotated until the browser window is closed. You can change the view while the view is rotated; changes will be reflected within one full rotation. Graphs rotate every 30 seconds by default. You can adjust the rotation delay in the GUI.

Another powerful aspect of automatic rotation is that if you have multiple monitors, you can invoke different views to be rotated on different monitors.

The gweb Mobile Tab

gweb mobile represents the Ganglia web interface optimized for mobile devices. This mobile view is found by visiting `/ganglia/mobile.php` on your gweb host. It is intended for any mobile browsers supported by the jQueryMobile toolkit. This display covers most WebKit implementations, including Android, iPhone iOS, HP webOS, Blackberry OS 6+, and Windows Phone 7. The mobile view contains only a subset of features, including views optimized for a small screen, host view, and search.

Custom Composite Graphs

Ganglia comes with a number of built-in composite graphs, such as a load report that shows current load, number of processes running, and number of CPUs; a CPU report that shows system CPU, user CPU, and wait IO CPU all on the same graph; and many others. You can define your own composite graphs in two ways: PHP or JSON.

Defining graphs via PHP is more complex but gives you complete control over every aspect of the graph. See the example PHP report for more details.

For typical use cases, JSON is definitely the easiest way to configure graphs. For example, consider the following JSON snippet, which will create a composite graph that shows all load indexes as lines on one graph:

```
{
   "report_name" : "load_all_report",
   "title" : "Load All Report",
   "vertical_label" : "load",
   "series" : [
      { "metric": "load_one", "color": "3333bb", "label": "Load 1",
        "line_width": "2", "type": "line" },
      { "metric": "load_five", "color": "ffea00", "label": "Load 5",
        "line_width": "2", "type": "line" },
      { "metric": "load_fifteen", "color": "dd0000", "label": "Load 15",
        "type": "line" }
   ]
}
```

To use this snippet, save it as a file and put it in the *graph.d* subdirectory of your gweb installation. The filename must contain *_report.json* in it to be considered by the web UI. So you can save this file in your gweb install as *load_all_report.json*.

There are two main sections to the JSON report. The first is a set of configurations for the overall report, and the second is a list of options for the specific data series that you wish to graph. The configuration options passed to the report are shown in Table 4-4.

Table 4-4. Graph configuration

Key	Value
report_name	Name of the report that web UI uses.
title	Title of the report to show on a graph.
vertical_label	Y-axis description (optional).
series	An array of metrics to use to compose a graph. More about how those are defined in Table 4-5.

Options for series array are listed in Table 4-5. Note that each series has its own instance of the different options.

Table 4-5. Series options

Key	Value
metric	Name of a metric, such as load_one and cpu_system. If the metric doesn't exist it will be skipped.
color	A 6 hex-decimal color code, such as 000000 for black.
label	Metric label, such as Load 1.
type	Item type. It can be either line or stack.
line_width	If type is set to line, this value will be used as a line width. If this value is not specified, it defaults to 2. If type is stack, it's ignored even if set.

Once you compose your graphs, it is often useful to validate JSON. One example would be to verify that there are no extra commas, etc. To validate your JSON configuration, use Python's `json.tool`:

```
$ python -m json.tool my_report.json
```

This command will report any issues.

Other Features

There are a number of features in gweb that are turned off by default or can be adjusted:

Metric groups initially collapsed
: By default, when you click on a host view, all of the metric groups are expanded. You can change this view so that only metric graph titles are shown and you have to click on the metric group to expand the view. To make this collapsed view the default behavior, add the following setting to *conf.php*:

```
$conf['metric_groups_initially_collapsed'] = true;
```

Filter hosts in cluster view
: If you'd like to display only certain hosts in the cluster view, it is possible to filter them out using the text box that is located next to the "Show Node" dropdown. The filter accepts regular expressions, so it is possible to show any host that has "web" in its name by entering `web` in the filter box; to show only webservers web10–web17, type `web1[0-7]`; or, to show web03 and web04 and all MySQL servers, type `(web0[34]|mysql)`. Note that the aggregate graphs will still include data from all hosts, including those not displayed due to filters.

Default refresh period
: The host and cluster view will refresh every 5 minutes (300 seconds). To adjust it, set the following value in *conf.php*:

```
$conf['default_refresh'] = 300;
```

Strip domain name from hostname in graphs
: By default, the gweb interface will display fully qualified domain names (FQDN) in graphs. If all your machines are on the same domain, you can strip the domain name by setting the `strip_domainname` option in *conf.php*:

```
$conf['strip_domainname'] = true;
```

Set default time period
: You can adjust the default time period shown by adjusting the following variable:

```
$conf['default_time_range'] = 'hour';
```

Authentication and Authorization

Ganglia contains a simple authorization system to selectively allow or deny users access to certain parts of the gweb application. We rely on the web server to provide authentication, so any Apache authentication system (htpasswd, LDAP, etc.) is supported. Apache configuration is used for examples in this section, but the system works with any web server that can provide the required environment variables.

Configuration

The authorization system has three modes of operation:

`$conf['auth_system'] = 'readonly';`
: Anyone is allowed to view any resource. No one can edit anything. This is the default setting.

`$conf['auth_system'] = 'disabled';`
: Anyone is allowed to view or edit any resource.

`$conf['auth_system'] = 'enabled';`
: Anyone may view public clusters without login. Authenticated users may gain elevated privileges.

If you wish to enable or disable authorization, add the change to your *conf.php* file.

When a user successfully authenticates, a hash is generated from the username and a secret key and is stored in a cookie and made available to the rest of gweb. *If the secret key value becomes known, it is possible for an attacker to assume the identity of any user.*

You can change this secret value at any time. Users who have already logged in will need to log in again.

Enabling Authentication

Enabling authentication requires two steps:

1. Configure your web server to require authentication when accessing *gweb/login.php*, and to provide the `$_SERVER['REMOTE_USER']` variable to *gweb/login.php*. (This variable is not needed on any other gweb page.)
2. Configure your web server to provide `$_SERVER['ganglia_secret']`. This is a secret value used for hashing authenticated user names.

If *login.php* does not require authentication, the user will see an error message and no authorization will be allowed.

Sample Apache configuration

More information about configuring authentication in Apache can be found here. Note that Apache need only provide authentication; authorization is provided by gweb configuration. A sample Apache configuration is provided here:

```
SetEnv ganglia_secret yourSuperSecretValueGoesHere
<Files "login.php">
  AuthType Basic
  AuthName "Ganglia Access"
  AuthUserFile /var/lib/ganglia/htpasswd
  Require valid-user
</Files>
```

Other web servers

Sample configurations for other web servers such as Nginx and Lighttpd are available on the gweb wiki.

Access Controls

The default access control setup has the following properties:

- Guests may view all public clusters.
- Admins may view all public and private clusters and edit configuration (views) for them.
- Guests may not view private clusters.

Additional rules may be configured as required. This configuration should go in *conf.php*. The `GangliaAcl` configuration property is based on the `Zend_Acl` property. More documentation is available here.

Note that there is no built-in distinction between a *user* and a *group* in `Zend_Acl`. Both are implemented as roles. The system supports the configuration of hierarchical sets of ACL rules. We implement user/group semantics by making all user roles children of the `GangliaAcl::GUEST` role, and all clusters children of `GangliaAcl::ALL`:

Name	Meaning
`GangliaAcl::ALL_CLUSTERS`	Every cluster should descend from this role. Guests have *view* access on Gan `gliaAcl::ALL_CLUSTERS`.
`GangliaAcl::GUEST`	Every user should descend from this role. (Users may also have other roles, but this one grants global view privileges to public clusters.)
`GangliaAcl::ADMIN`	Admins may access all private clusters and edit configuration for any cluster.
`GangliaAcl::VIEW`	This permission is granted to guests on all clusters, and then selectively denied for private clusters.
`GangliaAcl::EDIT`	This permission is used to determine whether a user may update views and perform any other configuration tasks.

Actions

Currently, we only support two actions, *view* and *edit*. These are applied on a per-cluster basis. So one user may have *view* access to all clusters, but *edit* access to only one.

Configuration Examples

These should go in your *conf.php* file. The usernames you use must be the ones provided by whatever authentication system you are using in Apache. If you want to explicitly allow/deny access to certain clusters, you need to spell that out here.

All later examples assume you have this code to start with:

```
$acl = GangliaAcl::getInstance();
```

Making a user an admin

```
$acl->addRole( 'username', GangliaAcl::ADMIN );
```

Defining a private cluster

```
$acl->addPrivateCluster( 'clustername' );
```

Granting certain users access to a private cluster

```
$acl->addPrivateCluster( 'clustername' );
$acl->addRole( 'username', GangliaAcl::GUEST );
$acl->allow( 'username', 'clustername', GangliaAcl::VIEW );
```

Granting users access to edit some clusters

```
$acl->addRole( 'username', GangliaAcl::GUEST );
$acl->add( new Zend_Acl_Resource( 'clustername' ), GangliaAcl::ALL_CLUSTERS );
$acl->allow( 'username', 'clustername', GangliaAcl::EDIT );
```

CHAPTER 5

Managing and Extending Metrics

Brad Nicholes, Daniel Pocock, and Jeff Buchbinder

In this chapter, we describe the various ways in which the Ganglia monitoring environment can be extended. Primarily, we discuss two ways in which to extend Ganglia, including the development and deployment of additional metric modules, and the use of a standalone utility called gmetric. Either is suitable, and you should use whichever approach works best for your environment. If you are someone who likes to get down and dirty in source code, developing a gmond module might be the way to go. On the other hand, if you just need a quick way to introduce an additional metric in your environment, gmetric is the perfect utility.

gmond: Metric Gathering Agent

The Ganglia monitoring daemon (gmond) is a lightweight agent whose primary purpose is to gather and report metric values. The first step in monitoring any system through Ganglia is to install the gmond daemon on each machine. Once installed and running, this daemon uses a simple listen/announce protocol via eXternal Data Representation (XDR) to collect and share monitoring state information with other gmond services within a cluster. In the Ganglia monitoring world, a cluster is defined as a group of gmond services that are all listening and sharing data with each other. A Ganglia cluster might consist of anything from a single gmond node to many nodes, all talking to one another. In the default configuration, each node listens and talks on a single multicast channel. As metric data is placed on the channel, every node in the cluster has an opportunity to retrieve and store the data in its own internal data store. Being configured in this way allows all cluster nodes to act as a backup for one another. Within the cluster, one gmond instance is usually designated as the primary node and is queried from time to time by the gmetad aggregating service. However, because any cluster node could potentially act as the primary node, gmetad has the ability to quickly switch from one node to another in the event that the primary node goes down. This means that within a cluster of gmond nodes, there is no weak link or single point of failure for

data collection. If any node within the cluster goes down, there is always another one ready to step up and take its place.

There are two different modes in which gmond clusters can be configured. The default mode, which was described previously, is the multicast mode in which each gmond node in a cluster is configured to listen for metric data as well as send its own data via a single multicast channel. In multicast mode, each gmond node not only gathers metric data from the host on which it is installed but also stores the last metric values gathered by every other node in the cluster. In this way, every node in a cluster is capable of acting as the primary node or reporting node for the gmetad aggregator in case of a failover situation. Which node within the cluster is designated as the primary node is determined through the configuration of gmetad itself. The gmetad configuration also determines which nodes will act as failover nodes in case the primary node goes down. The ability for any gmond node to report metrics for the entire cluster makes Ganglia a very highly robust monitoring tool.

The second mode in which gmond can be configured is unicast mode. Unlike multicast mode, unicast mode specifically declares one or more gmond instances as being the primary node or reporting node for the cluster. The primary node's job is to listen for metric data from all other leaf nodes in the cluster, store the latest metric values for each leaf node, and report those values to gmetad when queried. The major difference between multicast mode and unicast mode is that most of the nodes in the cluster neither listen for, nor store metric data from, any other nodes in the cluster. In fact, in many configurations, the leaf nodes in the cluster are configured to be "deaf" and the primary node is configured to be "mute." What this means is that a deaf gmond instance is only capable of gathering and sending its own metric data. A gmond instance that is mute is only capable of listening for and storing data from other nodes in the cluster. It is usually the mute nodes that are designated as the gmetad reporting nodes.

Another difference between unicast and multicast modes is that each gmond leaf node is configured to send its data via a UDP socket connection rather than a multicast channel. At first glance, unicast mode would appear to be less robust, due to the fact that not every node in the cluster can act as a reporting node—and if the primary node failed, no metric values would be reported to gmetad. However, because more than one instance of gmond can be designated as the primary node, redundancy can be achieved by configuring backup primary nodes and allowing each leaf node to send its metric data to both the primary node as well as any backup nodes. One thing to keep in mind is that multicast mode and unicast mode are not necessarily mutually exclusive. Both multicast and unicast can be used within the same cluster at the same time. Also, the configuration of gmond can include any number of send and received channels, which allows the configuration of a gmond metric gathering and reporting cluster to be extremely flexible in order to best fit your needs.

Base Metrics

From the very first release of Ganglia, gmond was designed to collect dozens of system metrics that included a series of CPU-, memory-, disk-, network-, and process-related values. Prior to version 3.1 of Ganglia, the set of metrics that gmond was able to gather was fixed. There was no way to extend this set of fixed metrics short of hacking the gmond source code, which limited Ganglia's ability to expand and adapt. However, there was a way to inject new metric values into the Ganglia monitoring system. Using a very simple utility that shipped with the Ganglia monitoring system, called gmetric, additional metric values could be gathered and written to the same unicast and multicast channels on which each gmond agent listened. gmetric's ability to inject data into the system allowed each gmond instance within a cluster to read and store these new metric values as if they had been originally collected by gmond. Even though gmetric provided a simple way of injecting a new metric into the system, the reality was that gmond was still incapable of gathering anything outside of its hard-coded set of metrics. This hard-coded set of metrics became known as the default or base metrics that most monitoring systems are used to gathering. Table 5-1 shows the set of included metrics. Beyond the base metrics, there are many other metrics that are provided through addition modules. These modules, along with a description of the metrics that they provided, are listed in Appendix A.

Table 5-1. Default gmond metrics

Metric name	Reporting units	Description	Type
`load_one`	Average over period	One-minute load average	CPU
`cpu_intr`	Percent	Percentage of time CPU is participating in IO interrupts	CPU
`load_five`	Average over period	Five-minute load average	CPU
`cpu_sintr`	Percent	Percentage of time CPU is participating in soft IO interrupts	CPU
`load_fifteen`	Average over period	Fifteen-minute load average	CPU
`cpu_idle`	Percent	Percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk IO request	CPU
`cpu_aidle`	Percent	Percent of time since boot idle CPU (not available on all OSs)	CPU
`cpu_nice`	Percent	Percentage of CPU utilization that occurred while executing at the user level with nice priority	CPU
`cpu_user`	Percent	Percentage of CPU utilization that occurred while executing at the user level	CPU
`cpu_system`	Percent	Percentage of CPU utilization that occurred while executing at the system level	CPU

Metric name	Reporting units	Description	Type
cpu_wio	Percent	Percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk IO request (not available on all OSs)	CPU
cpu_num	Count	Total number of CPUs (collected once)	CPU
cpu_speed	Mhz	CPU Speed in terms of MHz (collected once)	CPU
part_max_used	Percent	Maximum percent used for all partitions	Disk
disk_total	Gb	Total available disk space, aggregated over all partitions	Disk
disk_free	Gb	Total free disk space, aggregated over all partitions	Disk
mem_total	Kb	Total amount of memory displayed in KBs	Memory
proc_run	Count	Total number of running processes	Memory
mem_cached	Kb	Amount of cached memory	Memory
swap_total	Kb	Total amount of swap space displayed in KBs	Memory
mem_free	Kb	Amount of available memory	Memory
mem_buffers	Kb	Amount of buffered memory	Memory
mem_shared	Kb	Amount of shared memory	Memory
proc_total	Count	Total number of processes	Memory
swap_free	Kb	Amount of available swap memory	Memory
pkts_out	Packets/second	Packets out per second	Network
pkts_in	Packets/second	Packets in per second	Network
bytes_in	Bytes/second	Number of bytes in per second	Network
bytes_out	Bytes/second	Number of bytes out per second	Network
os_release	String	Operating system release date	System
gexec	Boolean	gexec available	System
mtu	Integer	Network maximum transmission unit	System
location	String	Location of the machine	System
os_name	String	Operating system name	System
boottime	Time	The last time that the system was started	System
sys_clock	Time	Time as reported by the system clock	System
heartbeat	Integer	Last heartbeat	System
machine_type	String	System architecture	System

One of the advantages of supporting a fixed set of metrics was that gmond could be built as a very simplistic self-contained metric gathering daemon. It allowed gmond to

fit within a very small and very predictable footprint and thereby avoid skewing the metrics through its own presence on the system. However, the disadvantage was obvious: despite producing a very vital set of metrics in terms of determining system capacity and diagnosing system issues by means of historical trending, gmond was incapable of moving beyond this base set of metrics. Of course, introducing the ability for gmond to expand would certainly increase its footprint and the risk of skewing the metrics. But given the fact that expanding gmond would be done through a modular interface, the user would have the ability to determine gmond's footprint through the configuration itself. Weighing the potential increase in footprint against the need to monitor more than just the basic metrics, the decision was made to enhance gmond by providing it with a modular interface.

Extended Metrics

With the introduction of Ganglia 3.1 came the ability to extend gmond through a newly developed modular interface. Although there are many different ways in which a modular interface could have been implemented, the one chosen for gmond was very closely modeled after one originally developed for the Apache HTTP server. Those familiar with the Apache Web Server may recognize one of its main features: the ability to extend functionality by adding modules to the server itself. In fact, without modules, the Apache Web Server is almost useless. By adding and configuring modules to the web server, its capabilities can be expanded in ways that for the most part, are taken for granted. So rather than reinvent a new type of modular interface, why not just reuse a tried and true interface? Of course, the fact that gmond is built on top of the Apache Portability Runtime (APR) libraries made the Apache way of implementing a modular interface an obvious fit.

With the addition of the modular interface to gmond in version 3.1, gmond was no longer a single self-contained executable program. Even the base metrics that were included as a fixed part of gmond were separated out and reimplemented as modules. This meant that if desired, gmond's footprint could be reduced even beyond the previous version by eliminating some of the base metrics as well. Because the base set of metrics are essential to any system, why would anybody want to reduce or even eliminate them? Back in "Configuring Ganglia" on page 20, the cluster configuration of the various gmond head and leaf nodes was described, including the multicast configuration where head nodes could be configured as mute. By configuring a head node as mute, basically there is no need for the node to gather metrics because it wouldn't have the ability to send them anyway. Therefore, if a node that has been configured to be mute can't send metrics, why include in its footprint the overhead of the metric gathering modules? Why not just make that instance of gmond as lean and mean as possible by eliminating all metric gathering ability? In addition to that scenario, if a specific instance of gmond is configured to gather metrics for only a specific device (such as a video card), why include CPU, network, memory, disk, or system metrics if they aren't needed or wanted? The point here is that the system administrator who is implementing

the Ganglia monitoring throughout his data center now has the ability and flexibility to configure and optimize the monitoring agents in a way that exactly fits his needs. No more, no less. In addition, the system administrator also has the flexibility to gather more than just basic system metrics. Ultimately, with the introduction of the modular interface, if a metric can be acquired programatically, a metric module can be written to track and report it through Ganglia.

When Ganglia 3.1 was initially released, it not only included the modular interface with a set of base metric modules but also included some new modules that extended gmond's metric gathering capabilities, such as TcpConn, which monitors TCP connection, and MultiCpu and MultiDisk for monitoring individual CPUs and disks, respectively. In addition to adding new metrics to gmond, these modules as well as others were included as examples of how to build a C/C++ or Python Ganglia module.

Extending gmond with Modules

Prior to the introduction of the modular interface, the gmetric utility, which will be discussed later in this chapter, was the only way to inject new metrics into the Ganglia monitoring system. gmetric is great for quickly adding simple metrics, but for every metric that you wanted to gather, a separate instance of gmetric with its own means of scheduling had to be configured. The idea behind introducing the modular interface in gmond was to allow metric gathering to take advantage of everything that gmond was already doing. It was a way to configure and gather a series of metrics in exactly the same way as the core metrics were being gathered already. By loading a metric gathering module into gmond, there was no need to set up cron or some other type of scheduling mechanism for each additional metric that you wanted to gather. gmond would handle it all and do so through the same configuration file and in exactly the same way as the core set of metrics.

Of course with every new feature like this, there are trade-offs. As of Ganglia 3.1, gmond would no longer be a single all-inclusive executable that could simply be copied to a system and run. The new modular gmond required modules that are separate dynamically loadable modules. Part of the transition from a single executable also included splitting out other components such as the Apache Portable Runtime (APR) library, which was previously being statically linked with gmond as well. The result of this new architecture was the fact that gmond became a little more complex. Rather than being a single executable, it was now an executable with library dependencies and loadable modules. However, given the fact that gmond is now much more flexible and expandable, the trade-off was worth it.

In the current version of gmond, there are two types of pluggable modules, C/C++ and Python. The advantages and disadvantages of each are, for the most part, the same advantages and disadvantages of the C/C++ languages versus the Python scripting language. Obviously, the C programing language provides the developer with a much lower-level view of the system and the performance that comes with a precompiled

language. At this level, the programmer would also have the ability to take full advantage of the C runtime and APR library functionality. However, C does not have many of the conveniences of a scripting language such as Python. The Python scripting language provides many conveniences that make writing a gmond module trivial. Even a beginning Python programmer could have a gmond Python module up and running in a matter of just a few minutes. The Python scripting language hides the complexity of compiled languages such as C but at the cost of a larger memory and processing footprint. One advantage to gmond modular interface is that there is plenty of room for other types of modules as well. As of the writing of this book, work is being done to allow modules to be written in Perl or PHP as well. You can enable these with `--enable-perl` and `--enable-php`, respectively.

C/C++ Modules

The first modular interface to be introduced into Ganglia 3.1 was the C/C++ interface. As mentioned previously, if you were to open the hood and take a peek at the gmond source code, and if you were familiar at all with the Apache HTTP server modules, you would probably notice a similarity. The implementation of the gmond modular interface looks very similar to the modular interface used by Apache. There were two major reasons for this similarity. First, one of the major components of gmond is the APR library, a cross-platform interface intended to provide a set of APIs to common platform functionality in a common and predictable manner. In other words, APR allows a software developer to take advantage of common platform features (that is, threading, memory management, networking, disk access, and so on) through a common set of APIs. By building software such as gmond on top of APR, the software can run on multiple platforms without having to write a lot of specialized code for each supported platform. Because gmond was built on APR, all of the APR APIs were already in place to allow gmond to load and call dynamically loadable modules. In addition, there was already a tried and proven example of exactly how to do it with APR. The example was the Apache HTTP server itself. If you haven't guessed already, APR plays a very significant role in gmond—everything from loading and calling dynamically loadable modules to network connections and memory management. Although having a deep knowledge of APR is not a requirement when writing a C/C++ module for gmond, it would be a good idea to familiarize yourself with at least the memory management aspects of APR. Interacting with APR memory management concepts and even some APIs may be necessary, as you will see in the following sections.

At this point, you might be wondering what the second reason is for modeling the gmond modular interface after the Apache HTTP server, as the first reason seemed sufficient. Well, the second reason is that the Ganglia developer who implemented the modular interface also happened to be a member of the Apache Software Foundation and already had several years of experience working on APR and the Apache HTTP server. So it just seemed like a good idea and a natural way to go.

Anatomy of a C/C++ module

As mentioned previously, the gmond modular interface was modeled after the same kind of modular interface that is used by the Apache HTTP server. If you are already familiar with Apache server modules, writing a gmond module should feel very familiar as well. If you aren't familiar with this type of module, then read on. Don't worry—the Ganglia project has source code examples that you can reference and can also be used as a template for creating your own module. Many of the code snippets used in the following sections were taken from the `mod_example` gmond metric module source code. If you haven't done so already, check out the source code for `mod_example`. It is a great place to start after having decided to implement your own C/C++ gmond metric module.

A gmond module is composed of five parts: the `mmodule` structure that defines the module interface, the array of `Ganglia_25metric` structures that define the metric that the module supports, the `metric_init` callback function, the `metric_cleanup` callback function, and the `metric_handler` callback function. The following sections go into each one of these module parts in a little more detail.

mmodule structure. The `mmodule` structure defines everything that gmond needs to know about a module in order for gmond to be able to load the module, initialize it, and call each of the callback functions. In addition, this structure also contains information that the metric module needs to know in order for it to function properly within the gmond environment. In other words, the `mmodule` structure is the primary link and the initial point of data exchange between gmond and the corresponding metric module. The `mmodule` structure for a typical metric module implementation might look something like this:

```
mmodule example_module =
 {
   STD_MMODULE_STUFF, /* Standard Initialization Stuff */
   ex_metric_init,    /* Metric Init Callback */
   ex_metric_cleanup, /* Metric Cleanup Callback */
   ex_metric_info,    /* Metric Definitions Array */
   ex_metric_handler, /* Metric Handler Callback */
 };
```

When defining the `mmodule` structure within your metric module, the first thing to notice about the structure is that it contains pointer references to each of the other four required parts of every gmond module. The data that are referenced by these pointers provide gmond with the necessary information and entry points into the module. The rest of the structure is filled in automatically by a C macro called `STD_MMODULE_STUFF`. At this point, there is really no need to understand what this C macro is really doing. But in case you have to know, it initializes to null several other internal elements of the `mmodule` structure and fills in a little bit of static information. All of the elements that are initialized by the C macro will be filled in by gmond at runtime with vital information that the module needs in order to run properly. Some of these elements include the module name, the initialization parameters, the portion of the gmond configuration

file that corresponds to the module, and the module version. Following is the complete definition of the `mmodule` structure. Keep in mind that the data stored in this structure can be referenced and used by your module at any time. The `mmodule` structure is defined in the header file *gm_metric.h*.

```
typedef struct mmodule_struct mmodule;
 struct mmodule_struct {
    int version;
    int minor_version;
    const char *name;               /* Module File Name */
    void *dynamic_load_handle;
    char *module_name;              /* Module Name */
    char *metric_name;
    char *module_params;            /* Single String Parameter */
    apr_array_header_t *module_params_list; /* Array of Parameters */
    cfg_t *config_file;             /* Module Configuration */
    struct mmodule_struct *next;
    unsigned long magic;
    int (*init)(apr_pool_t *p);     /* Init Callback Function */
    void (*cleanup)(void);          /* Cleanup Callback Function */
    Ganglia_25metric *metrics_info; /* Array of Metric Info */
    metric_func handler;            /* Metric Handler Callback Function */
 };
```

Ganglia_25metric structure. The name of the `Ganglia_25metric` structure does not seem to be very intuitive, especially as the purpose of this structure is to track the definitions of each of the metrics that a metric module supports. Nevertheless, every gmond module must define an array of `Ganglia_25metric` structures and assign a reference pointer to this array in the `metric_info` element of the `mmodule` structure. Again, taking a look at an example, an array of `Ganglia_25metric` structures might look like this:

```
static Ganglia_25metric ex_metric_info[] =
 {
   {0, "Random_Numbers", 90, GANGLIA_VALUE_UNSIGNED_INT,
      "Num", "both", "%u", UDP_HEADER_SIZE+8,
      "Example module metric (random numbers)"},
   {0, "Constant_Number", 90, GANGLIA_VALUE_UNSIGNED_INT,
      "Num", "zero", "%u", UDP_HEADER_SIZE+8,
      "Example module metric (constant number)"},
   {0, NULL}
 };
```

In the previous example, there are actually three array entries, but only two of them actually define metrics. The third entry is simply a terminator and must exist in order for gmond to appropriately iterate through the metric definition array. Taking a closer look at the data that each `Ganglia_25metric` entry provides, the elements within the structure include information such as the metric's name, data type, metric units, description, and extra metric metadata. For the most part, the elements of this structure match the parameter list of the gmetric utility that will be discussed in a later section. For a more in-depth explanation of the data itself, see "Extending gmond with gmetric" on page 97. The `Ganglia_25metric` structure is defined in the header file *gm_protocol.h*.

```
typedef struct Ganglia_25metric Ganglia_25metric;
 struct Ganglia_25metric {
    int key;       /* Must be 0 */
    char *name;    /* Metric Name */
    int tmax;      /* Gather Interval Max */
    Ganglia_value_types type; /* Metric Data Type */
    char *units;   /* Metric Units */
    char *slope;   /* Metric Slope */
    char *fmt;     /* printf Style Formatting String */
    int msg_size;  /* UDP message size */
    char *desc;    /* Metric Description */
    int *metadata; /* Extra Metric Metadata */
 };
```

metric_init callback function. The `metric_init` callback function is the first of three functions that must be defined and implemented in every gmond metric module. By the name of this function, you can probably guess that its purpose is to perform any module initialization that may be required. The `metric_init` function takes one parameter: a pointer to an APR memory pool. We mentioned earlier that it would probably be a good idea to understand some of the memory management concepts of APR. This is the point at which that knowledge will come in handy.

The following code snippet is an example of a typical `metric_init` callback function. The implementation in this example reads the module initialization parameters that were specified in the gmond configuration for the module and it defines some extra metric metadata that will be attached to metric information as it passes through gmond and the rest of the Ganglia system.

```
static int ex_metric_init ( apr_pool_t *p )
 {
   const char* str_params = example_module.module_params;
   apr_array_header_t *list_params = example_module.module_params_list;
   mmparam *params;
   int i;

   srand(time(NULL)%99);

   /* Read the parameters from the gmond.conf file. */
   /* Single raw string parameter */
   if (str_params) {
     debug_msg("[mod_example]Received string params: %s", str_params);
   }
   /* Multiple name/value pair parameters. */
   if (list_params) {
     debug_msg("[mod_example]Received following params list: ");
     params = (mmparam*) list_params->elts;
     for(i=0; i < list_params->nelts; i++) {
       debug_msg("\tParam: %s = %s", params[i].name, params[i].value);
       if (!strcasecmp(params[i].name, "RandomMax")) {
         random_max = atoi(params[i].value);
       }
       if (!strcasecmp(params[i].name, "ConstantValue")) {
         constant_value = atoi(params[i].value);
```

```
    }
  }
}

/* Initialize the metadata storage for each of the metrics and then
 *  store one or more key/value pairs.  The define MGROUPS macro defines
 *  the key for the grouping attribute. */
MMETRIC_INIT_METADATA(&(example_module.metrics_info[0]),p);
MMETRIC_ADD_METADATA(&(example_module.metrics_info[0]),MGROUP,"random");
MMETRIC_ADD_METADATA(&(example_module.metrics_info[0]),MGROUP,"example");

/*
 * Usually a metric will be part of one group, but you can add more
 * if needed as shown above where Random_Numbers is both in the random
 * and example groups.
 */
MMETRIC_INIT_METADATA(&(example_module.metrics_info[1]),p);
MMETRIC_ADD_METADATA(&(example_module.metrics_info[1]),MGROUP,"example");

return 0;
}
```

As gmond loads each metric module, one of the first things that it does is allocate an APR memory pool specifically for the module. Any data that needs to flow between the module and gmond must be allocated from this memory pool. One of the first examples of this is the memory that will need to be allocated to hold the extra metric metadata that will be attached to the metrics themselves. Fortunately, there are some helper C macros that will make sure that the memory allocation is done properly.

As mentioned previously, there were several elements of the `mmodule` structure that are initialized by the `STD_MMODULE_STUFF` macro but filled in at runtime by gmond. At the time when gmond loads the metric module and just before it calls the `metric_init` function, gmond fills in the previously initialized elements of the `mmodule` structure. What it means is that when your module sees the `mmodule` structure for the first time, all of its elements have been initialized and populated with vital data. Part of this data includes the module parameters that were specified in the corresponding module block of the gmond configuration file.

There are actually two elements of the `mmodule` structure that can contain module parameter values. The first element is called `module_params`. This element is defined as a string pointer and will contain only a single string value. The value of this element is determined by the configuration `params` (plural) directive within a module block. This value can be any string value and can be formatted in any way required by the module. The value of this parameter will be passed straight though to the module as a single string value. The second element is the `module_params_list`. The difference between the `module_params` and the `module_params_list` elements is the fact that the latter element is defined as an APR array of key/value pairs. The contents of this array are defined by one or more `param` (singular) directive blocks within corresponding module blocks of the gmond configuration file. Each `param` block must include a `name` attribute and a `value` directive. The name and value of each of the parameters will be included in the

`module_params_list` array and can be referenced by your module initialization function. There are two different ways of passing parameters from a configuration file to a metric module merely for convenience. If your module requires a simple string value, referencing the `module_params` string from the `mmodule` structure is much more convenient than iterating through an APR array of name/value pairs. Additionally, as there is no restriction on the format of the string contained in the `module_params` element, you can actually format the string in any way you like and then allow your module to parse the string into multiple parameters. Basically, whichever method of passing parameters to your module works best for you, do it that way. Or use both methods—it doesn't really matter.

There is one other aspect of metric module initialization that should be explained at this point: the definition or addition of extra module metadata. Each metric that is gathered by gmond carries with it a set of metadata or attributes about the metric itself. In previous versions of Ganglia, these metric attributes were fixed and could not be modified in any way. These attributes included the metric name, data type, description, units, and various other data such as the domain name or IP address of the host from which the metric was gathered. Because gmond can be expanded through the module interface, it is only fair that the metadata or metric attributes also be allowed to expand. As part of the module initialization, extra attributes can be added to each metric definition. A few of the standard extended attributes include the group or metric category that the metric belongs to, spoofing host, and spoofing IP address if the gmond module is gathering metrics from a remote machine. However, the extra metric metadata is not restricted to these extra attributes. Any data can be defined and set as extra metadata in a metric definition.

Defining the extra metadata for a metric definition includes adding a key/value pair to an APR array of metadata elements. Because adding an element to an APR array includes allocating memory from an APR memory pool as well as calling the appropriate APR array functions, C macros have been defined to help make this functionality a little easier to deal with. There are two convenience macros for initializing and adding extra metadata to the APR array: `MMETRIC_INIT_METADATA` and `MMETRIC_ADD_METADATA`. The first macro allocates the APR array and requires as the last parameter the reference to the APR memory pool that was passed into the `metric_init` callback function. The second macro adds a new metadata name/value pair to the array by calling the appropriate APR array functions. Because the extra metadata becomes part of the metric definition, this data can be referenced by your module at any time. If extra metadata was set that helps to identify a metric at the time that the module `metric_handler` function is called, this data could be referenced by accessing the `mmodule` structure. But keep in mind that because the extra metadata is attached to the metric itself, this data will also be passed through gmetad to the web frontend allowing the Ganglia web frontend to better identify and display metric information.

metric_cleanup function. The `metric_cleanup` callback function is the second function that must be implemented in every metric module and is also the last function that will be

called just before gmond terminates. The purpose of this function is to allow your module to clean up any memory allocations or threads, close any files, or simply tidy up any loose ends before the module is unloaded. Although many metric modules will have no need for this function, it still must be implemented and it will still be called on shutdown. The `metric_cleanup` function does not take any parameters, but at the time that it is called, all of the data that is stored in the `mmodule` structure is still valid and can still be referenced. Following is an example of a typical `metric_cleanup` callback function:

```
static void ex_metric_cleanup ( void )
 {
   /* Do any necessary cleanup here */
 }
```

metric_handler function. Finally, the `metric_handler` function is the last of the three functions that every metric module must implement. It is the callback function that actually does all of the work. It is called every time that gmond determines that your module needs to provide a new metric value that corresponds to any one of the metric definitions your module supports. The `metric_handler` function takes one parameter, which is an index into the `Ganglia_25metric` structure array that contains the definitions of your module's metrics.

```
static g_val_t ex_metric_handler ( int metric_index )
 {
   g_val_t val;

   /* The metric_index corresponds to the order in which
      the metrics appear in the metric_info array
   */
   switch (metric_index) {
     case 0:
       val.uint32 = rand()%random_max;
       break;
     case 1:
       val.uint32 = constant_value;
       break;
     default:
       val.uint32 = 0; /* default fallback */
   }

   return val;
 }
```

By using this metric definition index, your module can easily identify the metric that should be gathered, do the work necessary to gather the last metric value, and then finally return the value back to gmond. The return type of the `metric_handler` function is the `g_val_t` structure. The actual metric value that is returned from the `metric_handler` should be assigned to the element within the `g_val_t` structure that corresponds to the data type specified in the metric definition. In other words, if the metric definition stated that the data type of the metric was `GANGLIA_VALUE_UNSIGNED_INT`, the metric value that is gathered by the `metric_handler` function for this metric should be assigned to

the `uint32` element within the `g_val_t` structure. Unlike the Python metric module (which will be discussed in "Extending gmond with gmetric" on page 97), there is only one single `metric_handler` callback function in a C/C++ metric module. This single callback function must be implemented in such a way that it is able to gather metric values for any of the metrics supported by your module.

Outside of the five parts that are required of every metric module, your module is free to do anything else that may be required in order to gather metric values. Keep in mind that your module must also be well behaved. In other words, every time that gmond calls a metric module callback function, it expects the function to return a value in as short of a time as possible. gmond is not a multithreaded gathering agent and therefore, every millisecond that your metric handler callback function takes to gather a metric value is a millisecond that every other metric handler will have to wait before its turn to gather a metric value. If the metrics that your module supports take a significant time to gather, you might consider starting your own gathering thread within your module and caching the metric values instead. This way when gmond asks for a metric, our module can simply return the latest metric value from its cache.

Configuring a C/C++ metric module

The configuration of a gmond C/C++ module is a matter of telling gmond the name and location of the dynamically loadable module. In addition, the configuration may also include module parameters in the form of a single string parameter or a series of name/value pairs. Unlike a Python module, which will be discussed later in this chapter, a C/C++ module is loaded directly by gmond itself rather than by a module that acts as a language proxy. There are two required configuration directives that every module configuration file must provide: `name` and `path`. Previously, we discussed how to construct a C/C++ metric module and how the primary link between a metric module and gmond is the `mmodule` structure. During the implementation of the module, one of the first things that is done is to define and name the `mmodule` structure. The name of this structure is very important because the name is how gmond will identify the module throughout its lifetime. Because the `mmodule` structure's name is the module's unique identifier, the `name` directive in the module configuration file must match the `mmodule` structure name exactly. gmond will use this name to dynamically import the `mmodule` structure into its process space. Once gmond has imported this structure, it has all of the information it needs to continue to load and configure the C/C++ module for proper performance. The path to the module binary is also a required directive for every metric module. The `path` directive can either contain a full file path or just the module filename alone. If the `path` directive contains just the module filename, gmond will derive the module location in one of the following ways: the `module_dir` directive specified in the globals section of the configuration file, the `--with-moduledir` build parameter, or the `--prefix` build parameter. If none of these parameters have been specified, gmond will assume the location of the module binary to be the *lib(64)/ganglia* subdirectory relative to the Ganglia installation location.

```
modules {
  module {
    name = "example_module"
    path = "modexample.so"
    params = "A single string parameter"
    param RandomMax {
      value = 75
    }
    param ConstantValue {
      value = 25
    }
  }
}
```

The rest of the module configuration for a C/C++ module is optional. Of course, the `module` section must contain the `name` and `path` directives that have already been discussed. An optional parameter, which is not shown in the example above, is the `language` directive. The `language` directive is required for all other types of modules, but for a C/C++ module, if the `language` directive is not specified, the value "C/C++" is assumed. Specifying module parameters is also optional—unless, of course, your module requires parameters.

There are two ways in which module parameters can be passed to a module. The first way is by defining a single string parameter through the `params` directive. This directive just provides the configuration with a very simple way of passing either a raw string or a module-specific formatted string to the module's initialization function. gmond does not care what this string looks like and will not attempt to parse it in any way. Parsing the parameter string or interpreting the contents of the string is strictly up to the module itself. The second way to pass parameters to a module is through a series of `param` blocks. A module configuration can define as many `param` blocks as required by the module. Each `param` block must include a name and a value. The name of the `param` block represents the key that corresponds to the parameter value. gmond will pass all of the key/value pairs to the module's initialization function as part of an APR array. It is again up to the module to extract the parameter values from the APR array and interpret them appropriately. For an example of how to deal with both types of module parameters, see the `mod_example.c` example in the Ganglia source code.

Deploying a C/C++ metric module

Now that you have implemented, built, and configured your C/C++ module, it is time to deploy the module into your Ganglia environment. Deploying a module is as simple as copying the dynamically loadable module to the location that was specified in your module configuration as well as copying the module configuration to the Ganglia configuration file directory, */etc/conf.d*. There is really nothing more to it than that. In order to verify that the new module is actually being loaded and configured within the Ganglia environment, start an instance of gmond using the `-m` parameter. With the `-m` parameter specified, a new instance of gmond will not be started. However, this parameter will instruct gmond to load all of the modules that it knows about, read their corresponding

configuration files, call the `init` metric functions of each module, and then display on the console all of the defined metrics. The list of defined metrics should include all of the metrics defined by the new module (as well as all other defined metrics) and the module name in which each one was defined. Finally, once the new module has been loaded and all of the new metrics have been defined, start gmond normally and it should begin to gather and display the new metrics in the web frontend.

Cloning and building a C/C++ module with autotools

Two standalone module packages have been provided to the community to demonstrate the principles of building C/C++ modules. Traditionally, modules have been built by downloading the full Ganglia source tree and then working within the gmond/modules directory. This adds some slight complexity to the development process, especially when the developer wants to track changes coming from the main source code repository. At the time when the Ganglia source code is downloaded, built, and installed on the system, the `make install` command will deploy copies of the C headers that are required to compile a module. Packaged distributions (particularly on Linux and Debian) typically distribute those headers in a dev package and install them into the directory */usr/include*. The standalone `ganglia-modules-(linux|solaris)` packages demonstrate exactly how to build and distribute modules using nothing more than the public API of Ganglia, as provided in one of these dev packages.

You can either add a new module to the ganglia-modules-(linux|solaris) packages or use them as a basis for creating a new standalone project. There are two ways of using these projects as a model, along with some general comments on autotools that are common to both methods. These methods will be shown here.

Adding a module within either project. Thanks to the git source control system, you can easily clone either of these projects from the public repository in order to get started. From that point, you can create a branch and start developing a custom module within your branch. When you work this way, the process is very simple: first make a copy of the example module directory (for example, to create a module called "foo", you would copy the example contents to the "foo" directory). Second, modify the directory contents to your requirements. Finally, at the top level, modify both *configure.ac*, declaring your Makefile in `AC_OUTPUT`, and *Makefile.am*, declaring your new module directory in `SUBDIRS`.

Creating a new project. If you prefer to have your own project be completely standalone, then it is only slightly more difficult. First, start with the tarball distribution of one of the sample projects. In this case, rename one of the module directories to use as your own module. Second, delete all other module directories that are not required by your module. Finally, work through the *configure.ac* file and each instance of *Makefile.am* that you have retained, deleting references to the old modules and creating the necessary description of the module you want to build.

Putting it all together with autotools. No matter which approach you have chosen, once you have modified the *configure.ac* and *Makefile.am* files, it is essential that you re-run the autotools command `autoreconf`. Typically you would run a command such as `autoreconf --install` in the top level directory of your project. After that, you can then run `./configure` and then invoke any of the standard autotools `make` targets. For example, run `make DESTDIR=/tmp/my-module-test install` or `make dist` to make a source distribution tarball that you can upload to Sourceforge or any other community site.

Mod_Python

In the previous sections, we described how to build a gmond metric module using the C/C++ programming language. These sections also described the basic structure of a module along with its various functions that every gmond module must implement. This section will explain how to develop similar modules using the Python programming language. Many, if not most, of the concepts that were previously described for a C/C++ module are just as relevant for a Python module. The programming language may be different, but the anatomy of a gmond metric gathering module remains the same.

Mod_Python is a gmond metric module that doesn't actually gather metrics on its own. Instead, it is a module that was completely designed to act as proxy for other metric gathering modules. Mod_Python was written in the C programming language, and its primary purpose is to provide an embedded Python scripting language environment in which other gmond modules will run. In the previous section, we talked about the modular interface that was introduced in Ganglia 3.1 and also described what it takes to build a gmond module in C. Except for the fact that Mod_Python doesn't actually gather any metrics on its own, it still implements the gmond metric module interfaces in exactly the same way as any other C–based module. By taking advantage of the embedded Python capability, Mod_Python provides gmond with the ability to be extended through metric modules that are written in the Python scripting language rather than the C programming language.

With the introduction of Mod_Python, much of the complexity of implementing a gmond metric module in C has been removed. The gmond module developer no longer has to worry about `mmodule` structures, `MMETRIC_INIT_METADATA` macros, or the intricacies of the Apache Portable Runtime API. All of those elements of a gmond module are hidden behind the easy-to-understand Python scripting language. In fact, one of the very convenient aspects of developing a Python metric module is that you don't even need gmond at all. A gmond Python module can actually be fully designed, implemented, debugged, and tested without ever having to fire up a single instance of gmond.

One very important observation to make at this point is that there is no special bond between the Python scripting language and Ganglia. The fact that Mod_Python exists, and with it the ability to extend gmond's metric gathering ability through Python modules, is merely a convenience. The same thing could have been done, or better yet, can

be done with any embedded scripting language. The Python language is not unique in this area. By implementing a similar module with Perl, Ruby, or PHP, gmond would instantly gain the ability to be extended by implementing gmond metric modules in those languages as well. The main point to remember is that the gmond modular interface was written with future expansion in mind. There are plenty of opportunities to make gmond even more useful and extensible. Mod_Python just happens to be the forerunner to many other possibilities.

Configuring gmond to support Python metric modules

As mentioned previously, Mod_Python is a gmond metric module written in the C programming language. Because it is a module, it needs to be configured just like any other metric module. A base configuration file is included in the Ganglia source code and should have been installed automatically through a standard package install. The name of the dynamically loadable module is `modpython.so`; its corresponding configuration file is called *modpython.conf*. The configuration file for Mod_Python can usually be found in Ganglia's *etc/conf.d* directory of the Ganglia installation. The file contains a very simple and straightforward module configuration that follows the same pattern as every other gmond module. Following is a typical example of the Mod_Python configuration:

```
modules {
    module {
       name = "python_module"
       path = "modpython.so"
       params = "<directory_path_for_python_modules>"
    }
 }
 include ('../etc/conf.d/*.pyconf')
```

The most significant part of the Mod_Python configuration is the `params` directive. The path that is assigned to this directive will be passed down to the Mod_Python module and used to search for and locate the Python modules that should be loaded. Mod_Python will search this location for any file with a *.py* extension. For each Python module that it finds, it will attempt to also locate a matching module configuration block with a module name that corresponds with the Python module filename. The actual configuration for a Python module is described in "Configuring a Python metric module" on page 94. Any Python module that is found in this location and has a matching module configuration will be loaded and assumed to be a metric gathering module. The recommended file extension of a Python module configuration file is *.pyconf*. However, there is nothing about a Python module that requires any specific configuration file extension.

Writing a Python metric module

The first thing to understand when starting to implement a Python metric module is that there are three functions that every Python module must include and implement:

`metric_init(params)`, `metric_handler(name)`, and `metric_cleanup()`. The following is a stripped-down template of what a typical Python metric module might look like. As you can see, there really isn't much to one of these types of modules:

```
import sys, os
descriptors = list()

def My_Callback_Function(name):
    '''Return a metric value.'''
    return <Some_Gathered_Value_Here>

def metric_init(params):
    '''Initialize all necessary initialization here.'''
    global descriptors

    if 'Param1' in params:
        p1 = params['Param1'])
    d1 = {'name': 'My_First_Metric',
        'call_back': My_Callback_Function,
        'time_max': 90,
        'value_type': 'uint',
        'units': 'N',
        'slope': 'both',
        'format': '%u',
        'description': 'Example module metric',
        'groups': 'example'}

    descriptors = [d1]
    return descriptors

def metric_cleanup():
    '''Clean up the metric module.'''
    pass

#This code is for debugging and unit testing
if __name__ == '__main__':
    params = {'Param1': 'Some_Value'}
    metric_init(params)
    for d in descriptors:
        v = d['call_back'](d['name'])
        print 'value for %s is %u' % (d['name'],  v)
```

In order to give you a better understanding of the required elements of every Python module, we'll start out by describing each of the three functions that every Python module must implement. The first required function is `metric_init(params)`. The metric init function serves exactly the same purpose as the init function in a C/C++ metric module. However, unlike a C/C++ metric module init function, the Python init function must be called `metric_init` and must take a single parameter.

The `metric_init(params)` function will be called once at initialization time. Its primary purpose is to construct and return a dictionary of metric definitions, but it can also perform any other initialization functionality required to properly gather the intended metric set. The `params` parameter that is passed into the init function contains a

dictionary of name/value pairs that were specified as configuration directives for the module in the `module` section of the *.pyconf* module configuration file. Each metric definition returned by the init function as a dictionary object, must supply at least the following elements:

```
d = {'name': '<name>',          #Name used in configuration
 'call_back': <handler_function>, #Call back function queried by gmond
 'time_max': int(<time_max>),     #Maximum metric value gathering interval
 'value_type': '<data_type>',     #Data type (string, uint, float, double)
 'units': '<label>',              #Units label
 'slope': '<slope_type>',         #Slope ('zero' constant values, 'both' numeric values)
 'format': '<format>',            #String formatting ('%s', '%u','%f')
 'description': '<description>'}  #Free form metric description
```

All of the data elements that are required in a metric definition (i.e., the elements listed previously) are the same data elements that are described as command-line parameters of the gmetric command-line utility (see the section "Extending gmond with gmetric" on page 97). The only exception is the `call_back` function.

The `call_back` element designates a function that will be called whenever the data for this metric needs to be gathered. The callback function is the second required function that must be implemented in a gmond metric module. We will explain more about the callback function in just a moment, but for now just remember that the name of this function can be anything, but whatever name is assigned to the callback function must match the name given in the metric definition.

In addition to the required elements, the metric definition may contain additional elements. The additional elements will be ignored by gmond itself, but the name/value pairs will be included in the metric data packet as extra data. You can consider any additional elements as extra metadata for a metric. Metadata can be used to further identify or describe the metric and will be available to anything that consumes the metric XML produced directly by gmond or from gmetad. For example, if your metric definition contained an extra element `'my_key': '12345'`, the name/value pair would be included as an `EXTRA_ELEMENT` in the gmond XML dump. Because gmetad is the primary consumer of the gmond XML data, these `EXTRA_ELEMENT`s will also be stored by gmetad and available in the XML data that it produces.

There are a few special elements that can be used to enhance the web frontend or cause gmond to treat a specific metric as if it originated from a different host. These special elements are `GROUP`, `SPOOF_HOST`, and `SPOOF_NAME`. The concept of spoofing will be discussed in "Spoofing with gmetric" on page 99.

The `GROUP` element is used to categorize a metric in the web frontend. If the `GROUP` element is specified in a metric definition, the corresponding value may contain one or more group names separated by commas. Because the `GROUP` element along with its value is passed as an `EXTRA_ELEMENT` in the XML data, when the web frontend sees the group value, it uses that information to appropriately display the metric graph under the specified group header. Also keep in mind that the `GROUP` element may contain more than one group value. If additional group values are specified, the web frontend will

display the same metric group under each additional group designation as well. This can be very useful if the metric can be intuitively categorized in multiple ways. Ultimately, it allows the user of the web frontend to view the metric graph alongside other metrics within the same categories.

As mentioned earlier, the second function that must be implemented by all gmond Python modules is the metric handler function or the function that was referred to as the "callback" function. Unlike a C/C++ metric module, a Python metric module must implement at least one handler function, but it may also choose to implement more than one if needed. The handler definitions must be defined similar to the following:

```
def My_Metric_Handler(name):
```

The value of the name parameter will be the name of the metric that is gathered. This is the name of the metric that was defined in the metric definition returned by the init function described previously. By passing the metric name as a parameter to the callback function, a handler function gains the ability to process more than one metric and to determine which metric is currently being gathered. The callback function should implement all of the necessary code that is required to determine the identity of the metric to be gathered, gather the metric value, and return the value to gmond. The return value from the callback function must match the data type that was specified in the corresponding metric definition.

Finally, the third function that must be implemented in all gmond Python metric modules is the `cleanup` function.

```
def metric_cleanup():
```

This function will be called once when gmond is shutting down. The `cleanup` function should include any code that is required to close files, disconnection from the network, or any other type of clean up functionality. Like the `metric_init` function, the `cleanup` function must be called `metric_cleanup` and must not take any parameters. In addition, the `cleanup` function must not return a value.

As described previously, building a Python metric module requires the implementation of the init function and the cleanup function, at least one metric definition, and at least one callback function. Outside of those requirements, the metric module is free to do whatever it needs in order to appropriately gather and return the supported metric data.

Debugging and testing a Python metric module

One of the most convenient aspects of implementing a gmond module in the Python language is that Ganglia is not required during development. The entire implementation, debugging, and testing of the module can be done outside of the Ganglia environment. Thus, there is no requirement for any special development tools outside of the standard and familiar Python development tools that you are used to using. One of the best tools that we have found for developing Python metric modules is the Eric Python IDE. Eric is an open source IDE and Python language editor that was written

entirely in the Python language. The Eric Python IDE provides you with all of the features that you would expect from an IDE, including source code autocompletion, syntax highlighting, source code indenting help (which is a must in Python), and, most importantly, a full-featured multithread-capable debugger. The IDE also gives you a command-line window that is attached to the currently running Python script. The nice thing about the command-line windows is that when the IDE is stopped at a break point, you can use the command-line window to further inspect variables, construct new source code statements, or basically just do anything you want with Python.

In the module template that was shown in "Mod_Python" on page 89, there are several lines of code at the end of the template that enable you to test and debug your module outside of gmond. If you are not already familiar with writing Python code, the following statements will allow you to run your module as a standalone Python script:

```
#This code is for debugging and unit testing
  if __name__ == '__main__':
     params = {'Param1': 'Some_Value'}
     descriptors = metric_init(params)
     for d in descriptors:
        v = d['call_back'](d['name'])
        print 'value for %s is %u' % (d['name'],  v)
```

The `if` statement at the beginning of the block will be evaluated as true only if the Python script is being run by the Python interpreter directly. The rest of the code block defines the parameters that will be passed to the metric init function and iterates through each metric definition, calls the specified callback function with the metric name as a parameter, and finally prints out the metric value that was gathered. In other words, this very small block of code actually simulates the interaction that gmond would have with the module. Once you have implemented, tested, and debugged your Python metric module outside of gmond, you can be confident that your module will perform correctly when loaded by gmond. Of course, there is always the possibility that a problem might arise when your module is run inside the gmond-embedded Python environment. In order to debug your module under those conditions, there is always the `print` statement. Just remember to start gmond in debug mode.

Configuring a Python metric module

The configuration of a gmond Python module is really no different than any other module. In fact, the following configuration example should look very familiar. Most Python modules will not require any specialized configuration at all. One difference between the configuration of a Python module and C/C++ module is that the Python module configuration does not include a `path` directive. Modules written in the C/C++ language must include a path to the location of the dynamically loadable module or the module name relative to the library path (see the previous discussion on configuring C/C++ modules). However, as all Python modules are loaded and managed by Mod_Python rather than by gmond itself, the path that points to the location of the Python module is defined in the configuration of Mod_Python. In fact, the Python

module path is passed into Mod_Python as a simple parameter to Mod_Python's metric init function. Passing parameters into a Python module is supported in exactly the same way as well. This allows any module specific configuration to be specified in the configuration file and passed to the module at initialization time.

```
modules {
     module {
        name = "example"
        language = "python"
        param <param_1> {
           value = Whatever
        }
        param <param_2> {
           value = NewValue
        }
     }
  }
```

At the risk of duplicating how to configure a gmond metric module, we'll go through each of the directives shown in the previous example for configuring a Python metric module. The `module` section must contain a `name` and a `language` directive. The value of the `name` directive must match the file name of the Python module's *.py* file. When Mod_Python loads each Python metric module, it first searches through the Python module directory that was specified in the Mod_Python configuration for each file that includes the *.py* file extension. Once it finds and confirms that a *.py* file is actually a Python module, it then searches for a module configuration block whose name matches the name of the Python script file. All metric modules must specify the language in which they were written. Therefore, the value of the `language` directive for a Python module must be `"python"`. The final part of the module configuration in specifying any parameter values that should be passed into the `metric_init()` function. This part of the module configuration can take multiple `param` blocks. Each `param` block must include a name and a value. The name of the `param` block represents the key that corresponds to the parameter value in the dictionary object that is ultimately passed to `metric_init()`. Finally, the `value` directive specifies the parameter value. One thing to note is that the value of a parameter will always be passed to the metric module as a string within a dictionary object. If the module requires some data type other than a string, the string value of the parameter will have to be cast to the correct data type before it is used.

Deploying a Python metric module

In the previous section, we described how to implement, debug, and test your Python metric module. The next step is to actually deploy the module within a running instance of gmond. The nice thing about deploying a module into the Ganglia environment is that it is as simple as copying a file to a specific directory.

To deploy your newly developed Python module, copy your module's *.py* file to the Ganglia Python module directory. The directory path is specified in the configuration

of Mod_Python, which was described in "Configuring gmond to support Python metric modules" on page 90. The next step is to configure your Python module by creating a *.pyconf* configuration file. Then make sure that the module's *.pyconf* file has been copied to the Ganglia's *etc/conf.d* directory as well. In order to verify that the new module is properly loaded and configured within the gmond environment, start an instance of gmond using the `-m` parameter. With the `-m` parameter, a new instance of gmond will not be started, but this parameter will instruct gmond to load all of the modules that it knows about, read their corresponding configuration files, call the `metric_init()` functions of each module, and then display on the console all of the defined metrics. The list of defined metrics should include all of the metrics defined by the new module as well as all other defined metrics and the module name in which each one was defined. Once you have confirmed that the new module loads successfully, start gmond in normal operation mode. gmond should begin to gather and report the new metrics, and they should appear in the web frontend.

Spoofing with Modules

Spoofing is a concept that allows an instance of gmond running on one host to report the metrics that it gathers as if they were coming from an instance of gmond running on another host. In other words, gmond can fool the rest of Ganglia into thinking that the metrics that it is gathering are really coming from somewhere else. Spoofing was a concept originally designed and implemented as part of the gmetric utility. The idea of being able to report metrics as if they originated somewhere else was so popular in gmetric, that it only seemed natural to extend that idea into gmond modules as well.

Spoofing a metric within a gmond Python module is a matter of adding extra metadata elements to the metric description. Each metric definition, as previously described, may contain extra elements, which indicate to gmond that special handling of the metric is required. These extra elements include `SPOOF_HOST` and `SPOOF_NAME`. By adding `SPOOF_HOST` and `SPOOF_NAME` to a metric definition, gmond will treat the metric as a spoofed metric rather than an actual metric.

Because the concept of spoofing original came from the gmetric utility, the format of the `SPOOF_HOST` and `SPOOF_NAME` values also follow the same format as defined by gmetric. The `SPOOF_HOST` extra element specifies the IP address and the hostname of the host for which the metric should be reported. The format of the `SPOOF_HOST` value must be the IP address followed by the hostname separated by a colon (`ip_address:host_name`). When gmond sees this extra element, it will automatically replace the originating IP address and hostname with the values that are specified by this element. The `SPOOF_NAME` extra element is used to indicate to gmond that the metric definition should assume the name of a different metric. In other words, if spoofing is being used to gather the boot time of a remote host, the `SPOOF_NAME` can be set to `boot_time` to indicate that this metric is actually an alias of the standard `boot_time` metric. This concept makes a little more sense when you consider that each spoofed metric must also have a unique name.

Let's take the example of the `boot_time` metric. If you have a metric module that gathers the boot time of not only the host on which it is currently running but also several other remote hosts, the dictionary of metric definitions that is returned by this module must include a metric definition for the local `boot_time` metric as well as each remote host `boot_time`. Because gmond requires that every metric defined by a module have a unique metric name, there would be no way to define three different metrics all with the same `boot_time` metric name. Therefore, in order to uniquely identify each metric by its name, a common practice when defining a spoofed metric is to include the hostname as part of the metric name (`boot_time:my_host`). But naming a metric in this way would cause each of the remote host boot time metrics to show up in the web frontend as separate metrics that don't actually correspond to the boot time of the host. In order to fix this problem, use the `SPOOF_NAME` extra element to tell gmond that the metric definition is actually an alias for the standard `boot_time` metric.

Just to make sure that you got all of that, let's summarize. Specifying `SPOOF_HOST` as part of the metric definition tells gmond that this metric is a spoofed metric. The format of its value should be the remote host IP address followed by the hostname separated by a colon. Specifying `SPOOF_NAME` as part of the metric definition tells gmond that the spoof name is really an alias for another metric. Finally, the name of each spoofed metric must be unique. In addition to that, you will need to remember that when your metric callback function is called, the name parameter that is passed in will be the unique name of the metric and not the `SPOOF_NAME`. By passing in the unique name, this helps your callback function determine not only the metric it needs to gather but also the remote host that it should gather the metric from.

Extending gmond with gmetric

The gmetric utility, which is distributed with the monitoring core, as well as being available in multiple native variants for various programming languages, allows us to submit metrics values without having to rely on either the scheduler present in gmond, or the constraints of the native shared library or Python modules on which it depends.

gmetric packets can be debugged and examined using the gmond-debug instance, for which code is hosted on github. Running this utility will dump out the values for each packet as it is received and parsed.

Running gmetric from the Command Line

gmetric configures its metric transmission based on the settings of the local *gmond.conf* file (or another similarly formatted file, if one is specified). Beyond that, there are a number of command-line options to configure the behavior of gmetric, as listed in Table 5-2.

Table 5-2. gmetric command-line arguments

Short Option	Long Option	Description
`-h`	`--help`	Prints list of supported command-line parameters and exits. This argument should not be used with any other options.
`-V`	`--version`	Prints the current version of gmetric and exits. This argument should not be used with any other options.
`-c STRING`	`--conf=STRING`	Specifies the *gmond.conf* format configuration file to use for configuring send channels. By default, this will be the default installation location of *gmond.conf*, so if you are using the configuration specified in the active *gmond.conf* file, this option can be omitted.
`-n STRING`	`--name=STRING`	Specifies the string name that is attached to the submitted metric. By default, gmond will script most nonalphanumeric characters, with the notable exception of underscore, so you should restrict the character set of submitted metric names; otherwise, your names will be sanitized with underscores.
`-v STRING`	`--value=STRING`	Specifies the current value for the metric that you are submitting.
`-t STRING`	`--type=STRING`	Specifies the type of metric being submitted, which is used eventually by gmetad to aid in construction of the RRD files that will hold this metric. Acceptable values are `string` for string metric values, `int8` and `uint8` for 8-bit signed and unsigned integers, `int16` and `uint16` for 16-bit signed and unsigned integers, `int32` and `uint32` for 32-bit signed and unsigned integers, `float` for floating-point numbers, and `double` for double-precision numbers.
`-u STRING`	`--units=STRING`	Specifies a textual value that is used in display of the metric values to quantify the units of measurement used. This value is not used for any calculator or interpolation of the values, and is completely arbitrary. By default, no value is specified. An example unit value would be `C` for degrees centigrade or `kb` to indicate that the current metric indicates kilobytes.
`-s STRING`	`--slope=STRING`	Specifies the slope/derivative type of the metric being submitted. The possible values are `zero` for a zero slope metric, `positive` for an increment-only metric, `negative` for a decrement-only metric, and `both` for an arbitrarily changing metric. The default value is `both`. Using the value `positive` for the slope of a new metric will cause the corresponding RRD file to be generated as a `COUNTER`, with delta values being displayed instead of the actual metric values.
`-x INT`	`--tmax=INT`	Specifies the maximum time in seconds between gmetric metric submission. Defaults to 60 seconds.
`-d INT`	`--dmax=INT`	Specifies the lifetime of this metric in seconds. After the lifetime of the metric has expired, it will be represented as a NaN value and appear as a gap in its representative graph.
`-g STRING`	`--group STRING`	Specifies the arbitrary textual group name of this metric, which is used by Ganglia's web interface to group metrics. It is very useful when large

Short Option	Long Option	Description
		numbers of metrics are present for a host or cluster, as they can be viewed grouped by this value. Defaults to having no group specified.
`-C`	`--cluster=STRING`	Specifies a cluster name "hint", which is presented with the metric. The current version of gmetad does not respect this extra data, so it is necessary to use a *gmond.conf* configuration that submits data for the desired cluster. This behavior will most likely change in future, but for now, this flag does not have any effect.
`-D STRING`	`--desc=STRING`	Specifies an arbitrary textual description of the metric. It is not used for processing or interpretation of values, and it is blank by default.
`-t STRING`	`--title=STRING`	Specifies a display name, which is used by the Ganglia web UI to represent the metric. It defaults to being blank, which means that the metric name is used instead.
`-S STRING`	`--spoof=STRING`	Specifies a spoofed identification for the metric. This means that the metric will be identified by this information, rather than the local host identification assigned by the gmond instance(s) to which gmetric is transmitting this data. It uses a format of `IP+:+hostname`.
`-H`	`--heartbeat`	Indicates that a Ganglia heartbeat should be sent for a spoofed host. Heartbeat packets are usually sent by gmond instances, so that their current hosts are thought to be alive by upstream instances of gmond and gmetad. This argument should be used in tandem with the `-S`/`--spoof` option to fake a heartbeat packet for a host that does not directly correspond to a running gmond instance. When used, the metric-specific options, such as `-n`, `-v`, and so on, should not be specified.

Spoofing with gmetric

"Spoofing" a gmetric value involves submitting a "spoofed" hostname and IP address. It essentially allows packets to be submitted for hosts other than the gmond instance receiving the packets. The format is:

```
IP address + ":" + hostname
```

The hostname should match the reverse lookup of the IP address being presented, or your results may vary unexpectedly. You can use the fully qualified domain name of the host if you need to match what is being presented by another gmond instance. (This approach is covered in much greater depth in "Spoofing with Modules" on page 96.)

A large caveat is that due to the current cluster model used by gmond, you cannot submit spoofed metrics for a host that is not in the same cluster as the gmond instance to which you are submitting your gmetric packets. This issue can be overcome by simply using a separately specified *gmond.conf* file that points at the appropriate gmond instance (if you are using the standard monitoring core gmetric instance) or directly submitting the metrics to a gmond instance in the appropriate cluster (if you are not using the monitoring core gmetric instance).

An example of spoofing for a cluster using a VIP, assuming that the name of the machines are the VIP name plus `p1` or `p2` that could be used with regular gmetric metrics parameters, is:

```
#!/bin/bash
# gmetric-spoof.sh

SPOOF_HOSTNAME=$( hostname -s | sed -e 's/p[12]$//g;' )
SPOOF_IP=$( host $SPOOF_HOSTNAME | cut -d' ' -f4 | head -1 )
# Pull the hostname again to get FQDN
SPOOF_HOSTNAME=$( host $SPOOF_HOSTNAME | cut -d' ' -f1 | head -1 )
/usr/bin/gmetric -S ${SPOOF_IP}:${SPOOF_HOSTNAME} -H
/usr/bin/gmetric -S ${SPOOF_IP}:${SPOOF_HOSTNAME} $*
```

How to Choose Between C/C++, Python, and gmetric

With such a diverse range of options, what factors should you consider when choosing how to add a custom metric to Ganglia? To answer this question, let's take a look at what each method brings to the table as well as some of the complexities. The implementation of a C/C++ API would normally be chosen when there is a demand for minimal impact on the rest of the system. Because the C and C++ programing languages are lower-level compiled languages, the resulting modules tend to occupy a much smaller memory footprint on the system. The implementation of a C/C++ module would be most beneficial particularly when the metric-gathering code is intended to be run frequently and must perform complex functions in a very small period of time. This consideration is very relevant for servers running real-time systems, such as market data, Voice over IP (VoIP), or high-frequency trading. The development of a C/C++ module makes sense when there is preexisting C/C++ source code available that already lends itself to gathering and reporting metrics. For example, a database vendor may have provided a shared library and C headers for gathering metrics from the database. Implementing a C/C++ module in this instance is highly recommended.

The gmetric solution offers a very simple approach to introduce new metrics into the Ganglia system quickly. This is one of the main reasons why it should be chosen, especially when the need for simplicity is paramount and the requirements for performance are not a high priority. For example, when a system administrator is running a long upgrade task, he may only need to run the task once. In this case, there are no real-time performance concerns. In order to monitor the upgrade process, the admin may choose to write a shell script that invokes gmetric to gather the metric data during the upgrade. In such cases, the shell script is often fewer than 10 lines of code but may spawn multiple processes each time it executes.

One further consideration of gmetric: if a metric changes rarely, but the application generating the metric knows when it changes, then it can be appropriate to have the application invoke gmetric. This solution avoids the need for gmond (either a C module or Python module) to poll repetitively for a value that changes rarely.

Implementing a Python module is probably the most common choice for two reasons. It is simpler and quicker to develop a module using the Python language than it is using C or C++. This ease of use is particularly relevant when writing a module that must work on multiple platforms. The second reason for using Python is that it is much more efficient than gmetric, as it runs in the same address space as the gmond process. The implementation of a Python module is often chosen as a default when there is no compelling reason to use C/C++ or gmetric.

XDR Protocol

XDR is a binary protocol that is used by not only gmetric but also gmond itself to pass metric packets from one instance to another. The XDR protocol can also be used to insert metric packets into the metric stream by a third-party utility. In fact, gmetric is a good example of how the XDR protocol can be used in this manner. gmetric, being an external utility, uses the XDR protocol to submit metrics directly to a gmond instance in the form of a binary packet. The metric information is submitted as a series of two UDP packets: one containing metadata regarding the metric in question, and a second packet containing the metric value. Table 5-3 explains the lower-level format of XDR integer and string values. See Table 5-4 and Table 5-5 for descriptions of the metadata and value formats.

Table 5-3. Types

Type	Representation
INT	Integer values in XDR are represented as a series of four sequential 8-bit values that, taken together, form a single 32-bit integer value. The value is ordered high byte to low byte.
STRING	String values in XDR are represented as an integer XDR "length" value, being the length of the target string followed by a series of characters forming the string. These values are padded to a string length that is a multiple of 4 by 0 byte values. Null string values are represented as a zero-length value, followed by four 0 bytes.

Table 5-4. Metadata packet

Name	Type	Value
Packet type	INT	"gmetadata_full", represented as 128
Hostname	STRING	Source hostname for this metric. This should be represented in "spoof" form (IP address + ":" + hostname) if the packet represents spoofed data.
Metric name	STRING	Textual name of the metric presented. Any nonalphanumeric characters will be translated into underscore characters by gmond in modern versions of the monitoring core.
Spoof	INT	1 if the packet is being spoofed, 0 if it is not a spoofed packet.
Type representation	STRING	One of the following: unknown, string, uint16, int16, uint32, int32, float, double.
Metric name	STRING	Repetition of the earlier metric name field.

Name	Type	Value
Units	STRING	Textual name of the units being used for this metric.
Slope	INT	• 0: zero slope • 1: positive slope (creates a COUNTER style metric) • 2: negative slope • 3: both (should be the default for most metrics) • 4: unspecified
tmax	INT	Maximum time in seconds between gmetric values submitted. The minimum value of this should be 60.
dmax	INT	Lifetime in seconds of this metric. 0 indicates an unlimited lifetime.
Extra data qualifier	INT	This "magic" value specifies how many repetitions of the two following values are in the packet. Most packets contain at least a GROUP value, if not a SPOOF_NAME one. This value can be 0 if there are no additional extra data values being passed.
Extra data name (repeats)	STRING	Additional data name to be submitted with this packet. Common keys are "GROUP" and "SPOOF_NAME".
Extra data value (repeats)	STRING	Corresponding value for extra data.

Table 5-5. Value packet

Name	Type	Value
Packet type	INT	133 (128 + 5)
Hostname	STRING	Should match the hostname in the preceding metadata packet.
Metric name	STRING	Should match the metric name in the preceding metadata packet.
Spoof	INT	Should match the spoof value in the preceding metadata packet.
Format string	STRING	"%s". Note: ideally, this should be the printf/scanf-style format of the value being passed, but in reality, all are presently passed as string values, and are then converted to their specified types by gmond.
Metric value	STRING	Reported metric value.

Packets

A metadata packet is an XDR packet that contains the definition of an individual metric. Before a metric can be understood and viewed by the Ganglia monitoring system, its metadata must have been communicated throughout the system by a metadata packet. When an instance of gmond is started, the first thing it does is send a metadata packet over the network for each metric for which it intends to provide a value. If the configuration directive, `send_metadata_interval`, is set to a positive value, the instance of gmond will resend each metric's metadata according to the interval value.

A value packet contains only enough information to communicate a metric value. In order to reduce the amount of data that an instance of gmond produces, the actual

metric values are communicated through a much smaller packet, which allows gmond to communicate the much larger metric metadata on an as-needed basis while maintaining a very small footprint when communicating metric values.

Implementations

In addition to gmetric, there are several other metric-generating utilities that are available. Each of these utilities have generate metric data and insert the data into the metric stream through the use of the XDR protocol. Some of these metric generating utilities are listed in Table 5-6.

Table 5-6. Implementations

Software	Language	URL
Embeddedgmetric	C/C++	*http://code.google.com/p/embeddedgmetric*
Ganglia::Gmetric::PP	Perl	*http://search.cpan.org/~athomason/Ganglia-Gmetric-PP-1.04/lib/Ganglia/Gmetric/PP.pm*
gmetric4j	Java	*https://github.com/ganglia/gmetric4j*
gmetric-java	Java	*https://github.com/ganglia/ganglia_contrib/tree/master/gmetric-java* Note: doesn't support wire format for Ganglia 3.1 or greater
gmetric-python	Python	*https://github.com/ganglia/ganglia_contrib/tree/master/gmetric-python*
go-gmetric	Go	*https://github.com/jbuchbinder/go-gmetric*
jmxetric	Java/JMX	*http://code.google.com/p/jmxetric/*
node-gmetric	Node.js	*https://github.com/jbuchbinder/node-gmetric*
Ruby Gmetric	Ruby	*https://github.com/igrigorik/gmetric*

Java and gmetric4j

Various pure-Java implementations of the XDR protocol and gmetric functionality are available, including gmetric4j, jmxetric, gmetric-java, and the GangliaContext and related classes embedded within the Hadoop project.

Here we cover gmetric4j, as it can be used as a standalone in a wide variety of contexts (an application server or an Android application). It should also be noted that gmetric-java does not currently support the current Ganglia wire format; it supports only the format used before v3.1. gmetric4j supports both the new and old wire formats.

Deploying gmetric4j (in any context) involves two basic steps:

- Create subclasses of `info.ganglia.GSampler` to sample values for gmetric4j.
- Create an instance of `info.ganglia.GMonitor`, add your sampler(s), and invoke `GMonitor.start()`.

If there is no local gmond instance running on the machine where gmetric4j is deployed, then the machine will not be sending a heartbeat metric. gmetric4j includes the class `info.ganglia.CoreSampler`, which is capable of spoofing the heartbeat. Add this class to your GMonitor instance just as you would add any other custom sampler of your own:

```
info.ganglia.GMonitor g;
g = new GMonitor();
// Set up the communications properties
g.setGmetric(new GMetric("239.2.11.71", 8649, UDPAddressingMode.MULTICAST));
// Add info.ganglia.CoreSampler for heartbeat
g.addSampler(new CoreSampler());
// Add a custom GSampler instance, MySampler
g.addSampler(new MySampler());
g.start();
```

Real World: GPU Monitoring with the NVML Module

High-performance coprocessors like the Graphics Processing Unit (GPU) are growing in popularity within HPC clusters. In addition, GPUs designed for the cloud will soon debut. In HPC clusters, CPUs can offload data parallel workloads to the accelerators. The GPU/CPU hybrid architecture is desirable for its overall performance, high performance per watt, and ease of programming. In the cloud, virtualized GPUs will provide thin clients such as smartphones and tablets and access to a high-performance graphics experience. Virtual GPUs will enable applications ranging from computer games to 3D computer-aided design to run in the cloud.

The official NVML module is a valuable tool for cluster administrators who manage GPUs in an HPC cluster or datacenter environment. This module offers access to various GPU metrics that are needed to ensure high GPU availability and performance.

Installation

The installation process for the NVML module requires the installation of multiple packages on each gmond instance. The Python interpreter, along with pyNVML and the NVML plug-in, must be installed. It is best to ensure that all nodes are set up the same way using a common shell script, executed via a parallel SSH client or configuration management tool.

The first step in installing the NVML module is to install the NVIDIA display driver. If the machine is already set up to run CUDA, then the needed NVIDIA driver is already installed. The following command will verify that `nvidia-smi` is installed and working:

```
$ nvidia-smi -q
```

The `nvidia-smi` utility queries GPU metrics from the NVML library. Both the `nvidia-smi` utility and the NVML library were installed with the NVIDIA display driver package. Check the NVML feature matrix for a description of the supported features that

are available for each GPU. The NVML plug-in uses the same interface that `nvidia-smi` uses, so metrics that are unsupported in `nvidia-smi` are also unsupported in the NVML plug-in.

The NVML plug-in requires Python 2.5 or an earlier version with the ctypes library installed. Run the following command to ensure that you have the appropriate version of Python installed:

```
$ python -V

 Python 2.5
```

Once the required version of Python is installed, you can proceed to download and install pyNVML, the Python interface to the NVML library. pyNVML can be installed via the `easy_install` tool:

```
$ sudo easy_install nvidia-ml-py
```

or by downloading and installing the package manually here:

```
$ sudo python setup.py install
```

The next step is to download and install the NVML module. The *README* file that is included with the download contains the installation instructions.

Now that the module is installed, gmond must be restarted so that the new configuration can take effect. gmond will load the NVML module, which will allow the GPU metrics to be viewed in the node view of the Ganglia web interface, as shown in Figure 5-1.

Metrics

The NVML plug-in provides a variety of GPU metrics. These metrics include the GPU count and the NVIDIA driver version. For each GPU discovered on the system, the NVML modules expose the maximum and current clock speeds utilization information for the GPU memory and SM, temperature, fan speeds, power draw, ECC mode, used and total GPU memory, performance state, and identifiers such as the PCI bus ID, the GPU UUID, and the brand.

Reporting GPU metrics in Ganglia enables a cluster administrator to better monitor the GPUs in systems that they manage. Utilization information can provide a coarse-grained metric for assessing a cluster scheduler's efficiency. Temperature and fan speed information can provide insight into how effective the system cooling is working.

Configuration

The default configuration contained in the file *conf.d/nvidia.pyconf* exposes all GPU metrics. However, this default configuration can easily be modified to fit your needs by editing the module's configuration file. Please refer to Chapter 2 as well as the

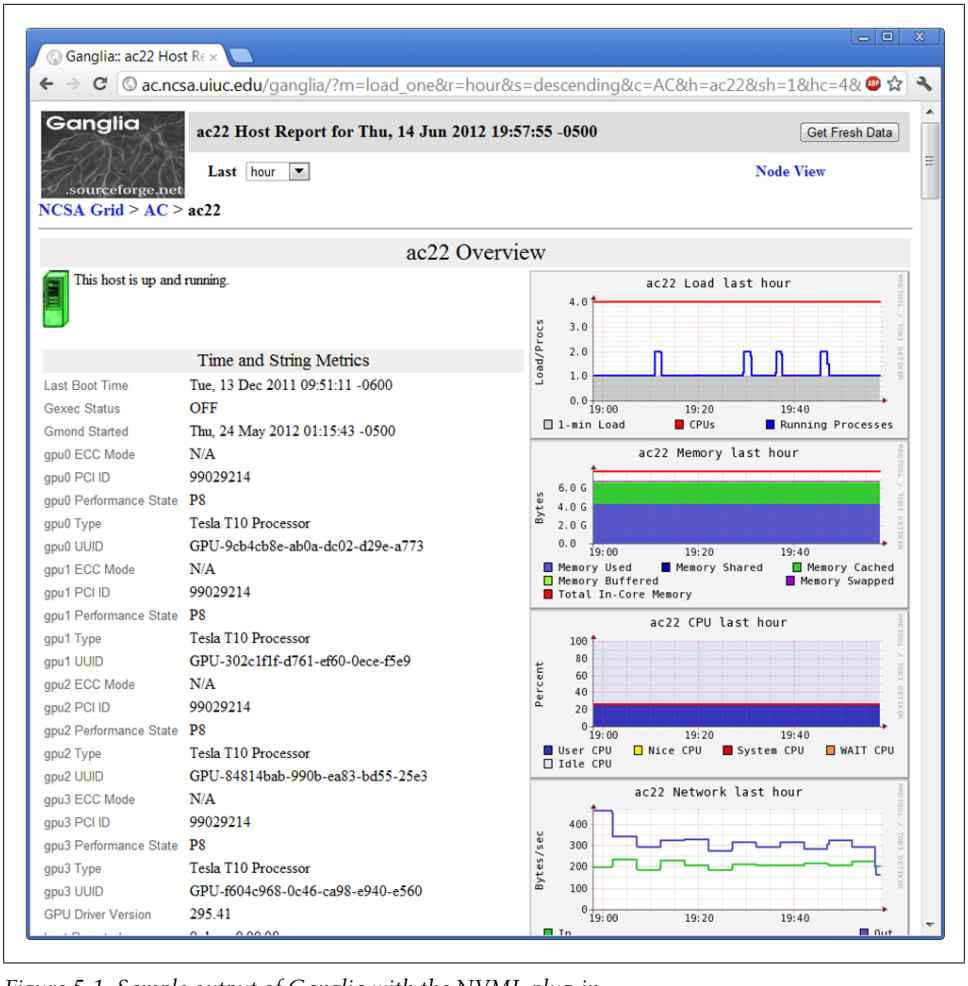


Figure 5-1. Sample output of Ganglia with the NVML plug-in

previous module configuration sections of this chapter for further information about how to modify the configuration of a gmond module.

For HPC nodes, there is often a desire to minimize the overhead related to the collection of metrics. Some GPU metrics are more expensive to query than others. For example, querying fan speed can be 100 times slower than querying the performance state. Disabling unwanted metrics or decreasing the frequency of queries to a particular collection group can help reduce undesirable overhead.

CHAPTER 6

Troubleshooting Ganglia

Bernard Li and Daniel Pocock

Overview

Sooner or later, you may encounter a problem with the Ganglia infrastructure. Because it is a distributed architecture, it is not always obvious which component is at fault. Sometimes, the fault may be completely outside the scope of the Ganglia system, such as a DNS issue, a faulty network card, or even a poorly configured web browser that results in a user mistakenly asserting that the Ganglia reports are broken.

This chapter aims to provide a systematic way of categorizing the faults, investigating them, identifying which component is responsible, rectifying the issue, and, if necessary, communicating details of the issue to the Ganglia community for discussion on the mailing list or registration in the bug database.

Known Bugs and Other Limitations

There are a number of known bugs and other limitations in the Ganglia system. For example, Ganglia is dependent on the system clock, and meaningful data will not be collected and reported if the cluster machines, data collectors, and web server machines do not have clock synchronization. This is a limitation of the Ganglia design, but it is not considered a bug.

Another known issue, fixed only just before the publication of this book (in Ganglia 3.3.7 and beyond), is that Ganglia was not working on a Solaris zone or container environment (this issue can also be worked around by disabling the network module).

To save time troubleshooting, you may wish to peruse the list of open bug reports in the Ganglia bug database maintained by the Ganglia community. If you encounter a similar issue in your environment, you will then avoid wasting time diagnosing it.

Useful Resources

In this section, we will describe helpful online/offline resources to help you troubleshoot issues with Ganglia.

Release Notes

Read the release notes, published on the download page. It usually includes important information particular to the release (or previous releases). A large portion of issues reported to the mailing lists are already known to the developers and noted in the release notes.

Manpages

Manpages are a given, but a lot of users some times forget they exist. The manpages are a great source of information pertaining to a particular component such as *gmond.conf* options and syntax. These manuals should be consulted to double-check that you have configured everything correctly.

Wiki

Our wiki has a lot of examples and information on different Ganglia implementations. Check to see whether what you are trying to do has been documented and whether you are doing it correctly. If it is not documented, consider adding it after you have figured everything out, be it an issue you were having or some special setup you have developed. User contribution plays a large role in the success of the project. The wiki resides in its permanent home here.

IRC

Ganglia developers and users usually hang out in #ganglia on *http://freenode.net*. If you run into issues or just feel like talking shop with other users, you are encouraged to join us. If you do post a question, please be patient, as not everybody is watching the channel all the time (not to mention in the same time zone), and you will get a response eventually. If not, or if the folks on IRC cannot answer your questions, you can post them to the mailing lists described next.

Mailing Lists

There are two main mailing lists that users can contribute to, namely: ganglia-general and ganglia-developers. If you encounter an issue, it is worthwhile to search the mailing-list archives to see if somebody else has encountered something similar.

The current mailing-lists are hosted at SourceForge and the search functionalities are subpar. The Ganglia developers recommend using the Mail Archive for searches. Simply search for "ganglia" when you arrive at the main page, which will bring up links to the two mailing lists that can then be searched. If the issue you encountered has not been previously discussed, consider starting a discussion thread and/or filing a bug in our bug tracker, discussed next. In order to post to a mailing list, you will need to first subscribe to it. Instructions are available at our SourceForge site.

Bug Tracker

Our main bug tracker is located here. Before filing a bug, it is worthwhile to do a quick keyword search to determine whether the issue is already in the system. If you have already done so but nothing came up, please file a new issue in the bug tracker. Please be as precise as possible when describing the issue and provide information that you believe is relevant for developers to troubleshoot the issue, such as the version of Ganglia you are using, the operating system, special configuration options, and so on. Remember, more information is better than not enough information.

Monitoring the Monitoring System

It's always a good idea to be proactive and anticipate problems. After all, that is why most people deploy Ganglia—to monitor the rest of their systems and be alerted before they have a crisis.

Monitoring the Ganglia infrastructure itself is also a good idea: doing so can help you study any problems that arise as the network grows.

Ganglia does not automatically monitor itself, but it is not hard to make it do so. Here are some of the key things you can do to monitor Ganglia with Ganglia:

- Enable mod_gstatus in the *gmond config* file. At a bare minimum, configure this module to report the running Ganglia version. This information can be useful for detecting agents that have not been updated.
- For any node that is acting as an aggregator of gmond packets, consider enabling the multicpu module. The gmond process is single-threaded, and the multicpu module will help identify when a core is always at capacity.
- For a gmetad server, consider installing and enabling the mod_io module to monitor disk IO levels on the disk storing the RRD files. Also monitor the filesystem capacity of those disks.
- See Chapter 7. Consider setting threshold alerts for the metrics mentioned in this section.

General Troubleshooting Mechanisms and Tools

In this section, we provide some general strategies for troubleshooting Ganglia and a discussion of various tools that are helpful.

netcat and telnet

Both gmond and gmetad communicates via TCP sockets in XML. Often the simplest way to figure out whether the daemons are working correctly is to examine the XML stream, which can be accomplished by running netcat against the ports that the daemons are listening on (by default, 8649 for gmond, 8651 for gmetad noninteractive, and 8652 for gmetad interactive). If netcat is not available, you could use the telnet tool, but it would not be possible to pipe the output for further processing.

If it is your first time examining XML outputs from gmond or gmetad (or XML, for that matter!), it might seem a bit daunting, because you do not know whether what you are seeing is "correct." However, you could still check for a few obvious errors:

- Can you see the hostnames (or IP addresses) of the hosts you expect?
- Can you see the metric values from those hosts?
- Do you see values that you do not expect (e.g., a `HOST` tag containing an IP address rather than a hostname in the `NAME` attribute)?
- Are tags properly closed?
- Are there actual contents in the XML or is everything simply open/close tags?
- Is the output truncated?
- Are the outputs what you expect; for instance, is the metric showing up as an integer when you are expecting a floating-point number?

Current versions of Ganglia (up to 3.4 at the time of writing) always emit the XML in a particular way (one tag per line, no indenting), which makes it very easy to scan with grep. Here are some common netcat/grep commands against gmetad port 8651. These commands also work with port 8649 on a gmond instance:

```
# observe a list of CLUSTER tags only:
netcat localhost 8651 | grep ^.C

# observe a list of CLUSTER and HOST tags:
netcat localhost 8651 | grep ^.[CH]

# show all hosts and show which hosts have the multicpu module active:
netcat localhost 8651 | egrep '^.H|^.M.*NAME=.multicpu_user'
```

The first 51 lines of XML output of both gmond and gmetad are always the same, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE GANGLIA_XML [
   <!ELEMENT GANGLIA_XML (GRID|CLUSTER|HOST)*>
```

```
   <!ATTLIST GANGLIA_XML VERSION CDATA #REQUIRED>
   <!ATTLIST GANGLIA_XML SOURCE CDATA #REQUIRED>
<!ELEMENT GRID (CLUSTER | GRID | HOSTS | METRICS)*>
   <!ATTLIST GRID NAME CDATA #REQUIRED>
   <!ATTLIST GRID AUTHORITY CDATA #REQUIRED>
   <!ATTLIST GRID LOCALTIME CDATA #IMPLIED>
<!ELEMENT CLUSTER (HOST | HOSTS | METRICS)*>
   <!ATTLIST CLUSTER NAME CDATA #REQUIRED>
   <!ATTLIST CLUSTER OWNER CDATA #IMPLIED>
   <!ATTLIST CLUSTER LATLONG CDATA #IMPLIED>
   <!ATTLIST CLUSTER URL CDATA #IMPLIED>
   <!ATTLIST CLUSTER LOCALTIME CDATA #REQUIRED>
<!ELEMENT HOST (METRIC)*>
   <!ATTLIST HOST NAME CDATA #REQUIRED>
   <!ATTLIST HOST IP CDATA #REQUIRED>
   <!ATTLIST HOST LOCATION CDATA #IMPLIED>
   <!ATTLIST HOST REPORTED CDATA #REQUIRED>
   <!ATTLIST HOST TN CDATA #IMPLIED>
   <!ATTLIST HOST TMAX CDATA #IMPLIED>
   <!ATTLIST HOST DMAX CDATA #IMPLIED>
   <!ATTLIST HOST GMOND_STARTED CDATA #IMPLIED>
<!ELEMENT METRIC (EXTRA_DATA*)>
   <!ATTLIST METRIC NAME CDATA #REQUIRED>
   <!ATTLIST METRIC VAL CDATA #REQUIRED>
   <!ATTLIST METRIC TYPE (string | int8 | uint8 | int16 | uint16 | int32 | uint32 |
                          float | double | timestamp) #REQUIRED>
   <!ATTLIST METRIC UNITS CDATA #IMPLIED>
   <!ATTLIST METRIC TN CDATA #IMPLIED>
   <!ATTLIST METRIC TMAX CDATA #IMPLIED>
   <!ATTLIST METRIC DMAX CDATA #IMPLIED>
   <!ATTLIST METRIC SLOPE (zero | positive | negative | both | unspecified)
                          #IMPLIED>
   <!ATTLIST METRIC SOURCE (gmond) 'gmond'>
<!ELEMENT EXTRA_DATA (EXTRA_ELEMENT*)>
<!ELEMENT EXTRA_ELEMENT EMPTY>
   <!ATTLIST EXTRA_ELEMENT NAME CDATA #REQUIRED>
   <!ATTLIST EXTRA_ELEMENT VAL CDATA #REQUIRED>
<!ELEMENT HOSTS EMPTY>
   <!ATTLIST HOSTS UP CDATA #REQUIRED>
   <!ATTLIST HOSTS DOWN CDATA #REQUIRED>
   <!ATTLIST HOSTS SOURCE (gmond | gmetad) #REQUIRED>
<!ELEMENT METRICS (EXTRA_DATA*)>
   <!ATTLIST METRICS NAME CDATA #REQUIRED>
   <!ATTLIST METRICS SUM CDATA #REQUIRED>
   <!ATTLIST METRICS NUM CDATA #REQUIRED>
   <!ATTLIST METRICS TYPE (string | int8 | uint8 | int16 | uint16 |
                           int32 | uint32 | float | double | timestamp) #REQUIRED>
   <!ATTLIST METRICS UNITS CDATA #IMPLIED>
   <!ATTLIST METRICS SLOPE (zero | positive | negative | both | unspecified)
                           #IMPLIED>
   <!ATTLIST METRICS SOURCE (gmond) 'gmond'>
]>
```

The interesting part comes after:

```
<GANGLIA_XML VERSION="3.1.7" SOURCE="gmond">
<CLUSTER NAME="AC" LOCALTIME="1340596942" OWNER="Jeremy Enos" LATLONG="unspecified"
         URL="unspecified">
<HOST NAME="ac28" IP="192.168.1.28" REPORTED="1340596933" TN="9" TMAX="20" DMAX="0"
      LOCATION="unspecified" GMOND_STARTED="1337840143">
<METRIC NAME="gpu3_type" VAL="Tesla T10 Processor" TYPE="string" UNITS="" TN="270"
        TMAX="90" DMAX="0" SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU3 Type"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu3 Type"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="mem_total" VAL="8191780" TYPE="float" UNITS="KB" TN="270" TMAX="1200"
        DMAX="0" SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="memory"/>
<EXTRA_ELEMENT NAME="DESC" VAL="Total amount of memory displayed in KBs"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="Memory Total"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu0_pci_id" VAL="99029214" TYPE="string" UNITS="" TN="270" TMAX="90"
        DMAX="0" SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU0 PCI ID"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu0 PCI ID"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="proc_run" VAL="1" TYPE="uint32" UNITS=" " TN="36" TMAX="950" DMAX="0"
        SLOPE="both">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="process"/>
<EXTRA_ELEMENT NAME="DESC" VAL="Total number of running processes"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="Total Running Processes"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu0_mem_speed" VAL="297" TYPE="uint32" UNITS="MHz" TN="36" TMAX="90"
        DMAX="0" SLOPE="both">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU0 Memory Speed"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu0 Memory Speed"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu1_ecc_mode" VAL="N/A" TYPE="string" UNITS="" TN="36" TMAX="90"
        DMAX="0" SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU1 ECC Mode"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu1 ECC Mode"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu1_mem_used" VAL="3840" TYPE="uint32" UNITS="KB" TN="36" TMAX="90"
        DMAX="0" SLOPE="both">
```

```
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU1 Used Memory"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu1 Memory Used"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu0_sm_speed" VAL="799" TYPE="uint32" UNITS="MHz" TN="36" TMAX="90"
       DMAX="0" SLOPE="both">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU0 SM Speed"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu0 SM Speed"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gpu2_mem_speed" VAL="297" TYPE="uint32" UNITS="MHz" TN="36" TMAX="90"
       DMAX="0" SLOPE="both">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU2 Memory Speed"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu2 Memory Speed"/>
</EXTRA_DATA>
</METRIC>
<METRIC NAME="gexec" VAL="OFF" TYPE="string" UNITS="" TN="199" TMAX="300" DMAX="0"
       SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="core"/>
<EXTRA_ELEMENT NAME="DESC" VAL="gexec available"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="Gexec Status"/>
</EXTRA_DATA>
</METRIC>
...
</HOST>
```

The output is truncated for brevity, but each host in this cluster has 103 metrics, and it repeats for each host. The end of the XML stream looks like this:

```
<METRIC NAME="gpu2_mem_total" VAL="4194112" TYPE="uint32" UNITS="KB" TN="272"
       TMAX="90" DMAX="0" SLOPE="zero">
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU2 Total Memory"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu2 Memory Total"/>
</EXTRA_DATA>
</METRIC>
</HOST>
</CLUSTER>
</GANGLIA_XML>
```

The previous example is taken from a gmond in a multicast environment. For unicast environment, only the collector gmond will have information on all hosts, as in this example.

The XML stream from gmetad looks very similar, except that it will have additional `<GRID> </GRID>`tags such as the following:

```
<GANGLIA_XML VERSION="3.1.7" SOURCE="gmetad">
<GRID NAME="NCSA" AUTHORITY="http://acfs/ganglia/" LOCALTIME="1340597372">
<CLUSTER NAME="AC" LOCALTIME="1340597360" OWNER="Jeremy Enos" LATLONG="unspecified"
         URL="unspecified">
<HOST NAME="ac07" IP="192.168.1.7" REPORTED="1340597359" TN="12" TMAX="20" DMAX="0"
      LOCATION="unspecified" GMOND_STARTED="1337840142">
```

The end of the stream looks like:

```
<EXTRA_DATA>
<EXTRA_ELEMENT NAME="GROUP" VAL="gpu"/>
<EXTRA_ELEMENT NAME="DESC" VAL="GPU0 Utilization"/>
<EXTRA_ELEMENT NAME="TITLE" VAL="gpu0 GPU Utilization"/>
</EXTRA_DATA>
</METRIC>
</HOST>
</CLUSTER>
</GRID>
</GANGLIA_XML>
```

Logs

gmond and gmetad do not log many things to syslog; however, these logs should always be checked when there is a problem. gmetad occasionally logs messages about failures to update RRD files. If there is a problem, and the logs provide no clues at all, see the next section.

The component that does log to syslog is the web frontend, which logs to the web server logs (for instance, */var/log/httpd/error_logs* for Apache running on Linux). If you are seeing a blank page when you point your web browser to your gweb page, you should check for errors in the error logs. Both errors in PHP code and in RRD file generation will show up in the web server error logs.

Running in Foreground/Debug Mode

All three main components of Ganglia—gmond, gmetad, and even the web frontend —have debug modes that could be triggered to provide verbose output when the programs are being executed.

For the daemons (gmond, gmetad), this is accomplished by specifying `-d` on the command line. By default, gmond/gmetad run as daemons (background processes). If a debug value greater than 0 is provided, the daemon will start in the foreground.

In debug mode, a lot more messages are printed during operation such that you can see precisely what is happening. Because there is a lot of output, it is recommended that you pipe the output to a temporary file and then examine the log file after the fact for any special notices that are displayed while the issue you are observing is happening.

For instance, here's a sample command for piping the output of running gmond in debug mode to the file */tmp/gmond.log*:

```
# gmond -d 10 > /tmp/gmond.log 2>&1
```

And here's the content of the corresponding log file:

```
loaded module: core_metrics
loaded module: cpu_module
loaded module: disk_module
loaded module: load_module
loaded module: mem_module
loaded module: net_module
loaded module: proc_module
loaded module: sys_module
loaded module: python_module
loaded module: multicpu_module
/usr/lib64/ganglia/python_modules/tcpconn.py:33: DeprecationWarning: The popen2↩
  module is deprecated.  Use the subprocess module.
  import os, sys, popen2
udp_send_channel mcast_join=239.2.11.71 mcast_if=NULL host=NULL port=8649

Discovered device /
Discovered device /boot

        metric 'tcp_established' being collected now
        metric 'tcp_established' has value_threshold 1.000000
        metric 'tcp_listen' being collected now
        metric 'tcp_listen' has value_threshold 1.000000
        metric 'tcp_timewait' being collected now
        metric 'tcp_timewait' has value_threshold 1.000000
        metric 'tcp_closewait' being collected now
        metric 'tcp_closewait' has value_threshold 1.000000
{...}
```

For the web frontend, you can specify debug mode by adding `&debug=n` at the end of the URL, where `n` could be any number greater than or equal to 0.

If you are trying to troubleshoot why a particular graph is not being generated, you can do so by first getting the URL of the graph, either by right-clicking the graph placeholder or just by looking at the source. Once you have the URL, you can then put that in the browser and add `&debug=3` at the end. This change will force the RRDtool command that is used to generate the graph to be displayed on the browser. You can then cut and paste the command into a command prompt on the gmetad server and execute it. It should then tell you exactly why the graph was not generated. It could be because of permission issues or syntax errors with the RRDtool command.

strace and truss

If the log messages and the debug mode output don't provide a clear indication of what is wrong, the next step is to try strace.

strace (truss on Solaris) is a tool that lets you hook onto a running process and intercept its system calls and signals. When one of the Ganglia programs fails with a segmentation fault or consumes 99 percent CPU, you can use strace to determine what system calls

are being executed while this is happening. The system calls will often show interesting details about which files the process accessed just before the problem occurred. If gmond or gmetad is misbehaving and you are trying to find out what it is doing, attaching to the running processes using strace could provide hints when running in debug mode (described later in this chapter) is not sufficient.

valgrind: Memory Leaks and Memory Corruption

A basic gmond that is in deaf mode and has no custom metrics should not be using more than 8 MB of RAM. A gmond process that receives metrics from other processes (over multicast or acting as a UDP aggregator) may grow its memory usage in a manner that is directly proportional to the number of hosts/metrics received. Any other growth in memory usage should be seen as a sign that gmond or one of the metric modules is misbehaving.

Check the process's RSS by running top or ps. If you observe it going up at a steady rate, try to run strace against it to see whether there are any clues as to what is causing the memory consumption. If you are using metric modules, try disabling them one by one to see if you can isolate the culprit. Finally, if all else fails, you can run tools such as valgrind and see what sort of information you could gather. If you believe this particular issue has not yet been reported, it is a good idea to file a bug in our bug tracker—and don't forget to include the valgrind output in the bug report.

Memory corruption is another possibility, particularly if a third-party metric module is behaving badly. valgrind should detect code that is accessing the wrong memory. If such errors are detected, please share them in a bug report.

iostat: Checking IOPS Demands of gmetad

A heavily loaded gmetad server can create a lot of IO, as described in Chapter 3. Observing IO levels during normal operation is highly recommended. If gaps are observed in all the graphs, it is often a symptom of IO saturation, and iostat can confirm this.

iostat is a common Linux command-line utility for checking disk IO levels. iostat can show the volume of IO (number of IOPS and MB/s). It can also show statistics about the IO queue performance, such as the average time an IO request is queued and the utilization rate of the block device.

Here is an example of running `iostat` with a logical volume:

```
$ iostat -k 1 -x dm-14
Linux 3.2.0-0.bpo.2-amd64 (srv1)        26/06/12        _x86_64_        (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           3.64    0.01    2.81    1.55    0.00   92.00

Device:         ...  svctm  %util
dm-14           ...   3.63   2.20
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          4.71    0.00    3.66    0.00    0.00   91.62

Device:         ...  svctm  %util
dm-14           ...   0.00   0.00
```

A value of `%util` approaching 100 percent indicates that the block device is saturated. In that case, it is necessary to either reduce the workload or increase the IO capacity of the block device. See "gmetad Storage Planning and Scalability" on page 44 for a discussion of scalability planning and common strategies and things to avoid.

Restarting Daemons

Occasionally, Ganglia gets into some strange state—for instance, the correct number of hosts are reported but it is not the correct total number of cores. Sometimes these issues can be resolved by restarting the daemons. Whenever restarting processes, be sure to do so in the order recommended in "Starting Up the Processes" on page 41.

gstat

gstat is a nifty command-line program that comes standard with Ganglia. It allows you to quickly determine things such as which hosts are down and how many cores each host has, as well as their respective load.

If you invoke the command without any options, you will be presented with a quick summary of your cluster:

```
CLUSTER INFORMATION
       Name: AC
      Hosts: 40
Gexec Hosts: 0
 Dead Hosts: 1
  Localtime: Sat Jun 30 19:03:05 2012

There are no hosts running gexec at this time
```

To get more details, specify the option `-a1`:

```
$ gstat -a1
CLUSTER INFORMATION
       Name: AC
      Hosts: 40
Gexec Hosts: 0
 Dead Hosts: 1
  Localtime: Sat Jun 30 19:05:12 2012

CLUSTER HOSTS
Hostname                     LOAD                         CPU              Gexec
 CPUs (Procs/Total) [     1,     5, 15min] [  User,  Nice, System, Idle, Wio]

ac43.local    24 (    0/  423) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.2,  ...
```

```
ac42.local    24 (    0/  421) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.1,  ...
ac33    24 (    0/  399) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.1,  99.9, ...
ac48.local    24 (    0/  425) [  0.00,  0.04,  0.11] [   0.0,   0.0,   0.1,  ...
ac44.local    24 (    0/  421) [  0.01,  0.01,  0.00] [   0.0,   0.0,   0.1,  ...
ac45.local    24 (    1/  425) [  1.05,  1.02,  1.00] [   4.3,   0.0,   0.1,  ...
ac46.local    24 (    1/  454) [  1.10,  1.07,  1.01] [   4.1,   0.0,   0.3,  ...
acfs    16 (    4/  585) [  0.53,  0.36,  0.35] [   6.0,   0.0,  12.6,  79.9, ...
ac20     4 (    0/  150) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.5,  99.4, ...
ac18     4 (    0/  150) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.5,  99.4, ...
{...}
```

This option gives you a list of all the hosts in the cluster with detail information regarding number of cores, number of processes running, load, and so on. Because `-1` was specified, it prints a host and its respective details per line.

To get the same list with the IP addresses of each host instead of their hostnames printed, use `-an1`:

```
$ gstat -a1n
CLUSTER INFORMATION
       Name: AC
      Hosts: 40
Gexec Hosts: 0
 Dead Hosts: 1
  Localtime: Sat Jun 30 19:11:21 2012

CLUSTER HOSTS
Hostname                     LOAD                          CPU              Gexec
 CPUs (Procs/Total) [     1,     5, 15min] [  User,  Nice, System, Idle, Wio]

192.168.1.44    24 (    0/  421) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.1,  ...
192.168.1.43    24 (    0/  423) [  0.00,  0.01,  0.00] [   0.0,   0.0,   0.2,  ...
192.168.1.42    24 (    0/  421) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.1,  ...
192.168.1.33    24 (    0/  399) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.1,  ...
192.168.1.48    24 (    0/  425) [  0.00,  0.00,  0.07] [   0.0,   0.0,   0.1,  ...
192.168.1.46    24 (    1/  454) [  1.00,  1.02,  1.00] [   4.1,   0.0,   0.3,  ...
192.168.1.45    24 (    2/  426) [  1.02,  1.01,  1.00] [   4.1,   0.0,   0.1,  ...
192.168.1.250    16 (    6/  585) [  0.36,  0.33,  0.34] [   8.8,   0.0,  19.7,  ...
192.168.1.15     4 (    0/  150) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.6,  ...
192.168.1.24     4 (    0/  150) [  0.00,  0.00,  0.00] [   0.0,   0.0,   0.5,  ...
{...}
```

To list just the dead hosts, use the `-d` option:

```
$ gstat -d
CLUSTER INFORMATION
       Name: AC
      Hosts: 40
Gexec Hosts: 0
 Dead Hosts: 1
  Localtime: Sat Jun 30 19:06:21 2012

DEAD CLUSTER HOSTS
                        Hostname   Last Reported
                            ac49   Mon May 28 16:53:01 2012
```

gstat comes in handy when you are trying to troubleshoot issues with Ganglia and you don't have direct access to the web frontend. By default, it tries to talk to the gmond running on localhost, but you can specify another running gmond by specifying `-i`.

Common Deployment Issues

There are a few common deployment issues that you should be aware of when deploying Ganglia.

Reverse DNS Lookups

The first time gmond receives a metric packet from any other node, it must do a name lookup to find the hostname corresponding to the packet's source address.

If these lookups are satisfied by */etc/hosts*, then it can be quite fast. If the lookups must be handled by DNS, this can slow down the process. As it is a single-threaded design, this scenario can have undesirable consequences.

When a gmond process first starts, it is likely that it will have to do such name lookups for all packets it receives in the first one or two minutes of operation. If a large number of hosts are reporting packets simultaneously, and if DNS is slow or even completely unavailable, this issue can have a severe impact and metrics will not be reported at all until the name lookups are complete and everything starts running normally.

Therefore, for those hosts that receive metrics (e.g., mute nodes or UDP aggregators), you should have multiple name servers defined and/or populate */etc/hosts* if convenient.

Time Synchronization

It is essential that all hosts participating in the Ganglia monitoring system have a synchronized clock. Proactively deploying network time protocol (NTP) is highly recommended.

If you see a message similar to the following in your web server logs, it is highly likely that the hosts running gmond are not time-synced:

```
alleviateFeb 22 05:33:22 localhost.localdomain /usr/sbin/gmetad[2782]:
RRD_update (/var/lib/ganglia/rrds/...metric.rrd): illegal attempt to
update using time 1329950002 when last update time is 1329950002
(minimum one second step)
```

Mixing Ganglia Versions Older than 3.1 with Current Versions

In version 3.1, a new wire format for the multicast/UDP packets was introduced. Therefore, hosts running 3.1 or later can't be used in the same cluster as hosts running 3.0 or earlier.

Specifically, if you are trying to upgrade your Ganglia installation, upgrade gmetad to the latest version first because it can communicate with gmond 3.1 and newer as well as 3.0 or earlier. Afterwards, you can upgrade each cluster one at a time to the current version, but remember that each cluster cannot have a mixture of pre-3.1 gmonds and gmonds running 3.1 or newer.

SELinux and Firewall

If you are new to Ganglia and feel that you have followed all the installation instructions to the letter, but for some reason you are not getting any graphs on the web frontend, you might want to check whether your OS has SELinux enabled by default.

Ganglia developers generally recommend disabling SELinux on systems running Ganglia daemons/web frontend, as it hinders normal operations. If you are attempting to run Ganglia on systems that require SELinux, it is possible to create a security profile that allows Ganglia to work. However, how to write such a SELinux security profile is outside the scope of this book.

Ganglia daemons communicate with each other via network (TCP/UDP) sockets. The web frontend also needs to communicate with gmetad via the interactive port. If a firewall needs to be in place on servers running the Ganglia services, please ensure that the ports are opened on the firewall. Default ports are 8649 for gmond, 8651 for gmetad noninteractive, and 8652 for gmetad interactive.

Typical Problems and Troubleshooting Procedures

In this section, we list and categorize common issues encountered with a Ganglia installation. We categorize based on when the user will first notice the issue. For instance, one might have misconfigured gmond to the extent that certain nodes are not shown in the web interface. Even though technically the issue lies in gmond, because the user will first come across the issue via the web interface, this particular issue will be listed under "Web issues."

Web Issues

There are a number of well-known problems that can occur when using gweb. We've listed them here to help you quickly get back up and running.

Blank page appears in the browser

Check the Apache access log: did the browser connect to the web server? If not, it could be a problem with the browser itself or a web proxy.

Does the Apache error log contain any errors? Look for errors about file permissions, missing PHP modules, and the like. Try monitoring the log with `tail -f` while you

reload the blank page; doing so will let you see exactly which messages are related to the failure.

Try adding `?debug=3` to the end of the URL and check the error log by using `tail -f`.

Browser displays white page with error message

Is it an error about failing to connect to 127.0.0.1:8652? This error typically means that gmetad is not running or not listening on the correct port.

Otherwise, please see the tips for the previous problem and look for more details in the various log files. Check both the web server logs and the system logs (grep for any errors mentioning gmetad).

Cluster view shows uppercase hostname, link doesn't work

Often the host appears twice, with the name in uppercase and lowercase, or some other differing variations of the hostname. Clicking some of the hosts shows a blank page and no metrics.

Older versions of Ganglia treated hostnames in a case-sensitive manner, which is incorrect. In some environments, the hostname is uppercase in the host's file and lowercase in DNS. Ganglia thus becomes confused. As of Ganglia 3.3, hostnames are converted to lowercase, RRD files are created with lowercase filenames, and lowercase hostnames should be used in URLs to access the metrics. A config option allows the legacy behavior to remain, but if you incorrectly have that option enabled in either or both *gmetad.conf* and *config.php*, you may have trouble.

Host appears in the wrong cluster

In *gmond.conf*, you have defined the name of the Cluster to be cluster2, yet the host is still displayed under cluster1. This is a common misconception of how Ganglia and gmond work. gmond clusters via the port defined for `udp_send_channel` but not the name of the cluster. The cluster name is used to generate the XML, which in turn shows up in the web frontend. If you would like the host to show up under a different cluster, use different ports for `udp_send_channel`. By default, the port number for gmond is 8649. Don't use ports 8651 and 8652, as these are the default ports used by gmetad.

Host appears multiple times in web, different variations of the hostname (or IP address)

Usually, the different hostnames/IP addresses correspond to different interfaces on the host: the agent randomly picks a source IP address on the host for sending out metric packets, which can produce unexpected results. If gmond is restarted, sometimes it will not be sending on the same interface/source IP that it used previously.

The `bind_hostname` parameter can be used to lock it to the correct interface.

Some hosts appear with shortname instead of FQDN

The hostnames are inserted into the XML stream by the gmond host receiving the multicast or UDP metric packets. It does a reverse lookup against the source address of the packet. Sometimes that host has shortnames in */etc/hosts* and it uses them instead of the FQDNs from DNS. Therefore, consider adapting */etc/hosts* to contain both FQDN and shortname for each IP address.

One or more hosts don't appear in the web interface

Make sure gmond is running on the host in question (see the section "netcat and telnet" on page 110).

If the hosts have only recently been added to the network, check whether the RRD files for the hosts have been created yet. If the gmetad server has a full filesystem, it will fail to create RRD files.

If using multicast, check the interface where packets are sent (a packet sniffer such as `tcpdump` might help). One gotcha is that if you wanted multicast traffic to go through `eth1` as opposed to the default `eth0`, you need to add a route explicitly:

```
# route add -host 239.2.11.71 dev eth1
```

Hosts don't appear/data stale after UDP aggregator restarted

Occasionally, it is necessary to restart your collector gmond. If, as soon as you do that, you notice that other gmonds stopped reporting new metric data, you should make sure that `send_metadata_interval` in *gmond.conf* is set to a nonzero value.

Dead/retired hosts still appearing in the Web

Let's say you have a 10-node cluster, one node has a hardware failure and cannot be repaired. You may have replaced the node with another one with a different IP address, or you do not want the dead host to show up any more. For unicast mode, all you need to do is restart the collector. For multicast node, all nodes in the cluster will need to be restarted. Subsequently, gmetad will also need to be restarted. This step should flush the dead node out of the system.

Alternatively, you could set `host_dmax` to a nonzero number. This will flush out the node automatically after the specified number of seconds have passed. For unicast mode, this value needs to be set only on the collector. For multicast mode, the value needs to be set on all gmonds' configuration.

This issue is also relevant when monitoring dynamic environments such as cloud resources (for example, Amazon EC2). Hosts are constantly brought up and shut down and the IP addresses of hosts are generally not from a constant pool. You may end up with a lot of dead hosts in your cluster if this option is not properly set.

Wrong CPU count or other metrics are missing

You noticed that "Hosts up" is correct but "CPUs Total" is less than expected. If you drill down to the host view, some hosts are missing certain metrics and graphs. In that case, you might want to reload gmond on the hosts in question. If that does not work, try to do a systemic restart of all the daemons.

Fonts in graphs are too big or too small

Graphs by default are generated by RRDtool. In certain situations, the fonts of the graphs may be too large or too small, making it difficult to read the text. In that case, try to install different TrueType fonts on your system. Alternatively, try upgrading the version of RRDtool, as newer versions have better font management.

As of RRDtool v1.3, `fontconfig` is used to access system fonts. By default, it will use the font DejaVu Sans Mono. If that's not available, it will try Bitstream Vera Sans Mono, Monospace, and finally Courier. Make sure that `fontconfig` provides one of these fonts. Use `fc-list` to see which fonts are installed on your system.

Spikes in graphs

Sometimes it is possible to have unexpected spikes in your graphs that will throw the scale totally off. If you are certain the spikes are not normal, you can remove them from the RRDtool database using the contributed script *removespikes.pl*, which is usually shipped in the Ganglia release tarball under *contrib/*. If it is not available, you can get it from the github repository.

Certain Broadcom Network Interface Controllers (NICs) are known to cause spikes due to hardware bugs. If the spikes are in your network graphs and are in the range of petabytes/sec, consider rebuilding Ganglia using the following flag:

```
make CPPFLAGS=-DREMOVE_BOGUS_SPIKES
```

This flag is known to alleviate the issue under Linux.

Custom metrics don't appear

So you have written your first gmond metric module in Python and have confirmed by testing that it is working as expected on the host you would like to collect data. You have installed the module and the corresponding `pyconf` in the right location, but no matter how many times you restart gmond, you are still not seeing the graph on the web frontend.

Traditionally, gmond is executed by an unprivileged user, such as `nobody` or `ganglia`, which has limited access. Your Python module will also be executed by this user: therefore, it is important to check whether that user can run the script without any issue (such as accessing certain files).

It is also a good idea to run gmond with the `-m` parameter. The `-m` parameter will instruct gmond to load each module and display a list of all of the metrics that it knows about. Check the output listing to make sure that your new metrics are included.

Custom metric's value is truncated

You have just injected a new metric into Ganglia, which is a long string. However, you noticed that it is being truncated.

By default the value of a metric is stored in a 32-byte structure. Anything beyond will be truncated. It is possible to increase this value by recompiling Ganglia; alternatively, you could also split the value into multiple metrics.

Gaps appear randomly in the graphs

Gaps that appear randomly in the graphs are often a sign that some component is overloaded. Verify that no other network congestion issues exist. The UDP packets sent by gmond are likely to be dropped by congested routers and switches.

On the gmond that receives the packets (either via multicast or acting as a UDP aggregator), verify that the UDP receive buffer is big enough. Versions of Ganglia 3.4.0 and later have a "buffer" parameter in *gmond.conf* that can be used to make the buffer larger.

Verify that the gmond process handling the TCP polls from gmetad is not overloaded. If the gmond is using 100 percent of CPU, that is a sign that it is overloaded. Split the cluster, reduce the number of metrics from each member node, reduce the transmission rate from the member nodes, or put that gmond on a more powerful CPU.

Verify that the gmetad process is not overloading the CPU. If the gmetad is using 100 percent of CPU, that is a sign that it is overloaded. Split the workload over multiple gmetads on different physical hosts.

Confirm that the IO device storing the RRDs is not overloaded. See "iostat: Checking IOPS Demands of gmetad" on page 116 and also review Chapter 3.

If gaps are showing up only on graphs of a new metric added via gmetric or the gmond metric interface, you may have specified an incorrect slope for the metric.

For most cases, use `slope=both`, which will cause the underlying RRD file to be of type `GAUGE` and thus each data point collected will be graphed. However, if the metric you are collecting is a rate of change, then specify `slope=positive` such that the RRD file will be of type `COUNTER`. For more information regarding the difference between the two types, refer to the documentation for RRDtool.

Some host is completely missing from the cluster

Confirm that the host is up and the gmond process is running. If the gmond process won't start, see the sections about troubleshooting gmond in debug mode.

Use a packet sniffer (tcpdump or wireshark) to verify that the host is transmitting packets.

Try restarting the gmond: this will cause it to retransmit its metadata.

If using UDP unicast, use netcat to check the UDP aggregator for the cluster: does it hear the host?

If using multicast, use netcat to check the XML from a host in the cluster that is not deaf: does it have XML for the missing host?

Use netcat to check the XML from the gmetad: does it have XML for the missing host?

Check if RRD files exist under */var/lib/ganglia/rrds*; verify that they don't have size 0. If the filesystem fills up, the files are sometimes created with size 0.

gmetad hierarchy and federation; some grids don't appear on the Web

Check the `trusted_hosts` setting in every *gmetad.conf*. The ACL can be tested by executing netcat between the gmetad hosts. Make sure that the gmetad that is aggregating the grids is configured to poll the correct port on the lower level gmetad: it should poll port 8651, not 8649 or 8652.

gmetad Issues

This section contains a list of common gmetad problems and their solutions.

Empty (size = 0) RRD files created

This often happens if the filesystem is full.

gmetad takes a long time to start

If the network is large, this could be a scalability issue; see the discussion in "Acute IO Demand During gmetad Startup" on page 46.

gmetad segmentation fault writing to RRD

gmetad crashes with a segmentation fault. The stack trace or strace output shows that it was writing to an RRD. This may be due to a buggy version of RRDtool. Unfortunately, the buggy version of RRDtool was distributed in Fedora 14 for a while (see Ganglia bug 287). Using RRDtool v1.4.4 or greater may fix this issue; otherwise, please try the other troubleshooting tools/techniques.

gmetad doesn't poll all nodes defined in data_source

In *gmetad.conf*, the `data_source` definition can list multiple nodes for a single cluster. However, gmetad doesn't automatically failover/poll the second node when the first is

down. This is a known issue. In many cases, it is necessary to completely restart gmetad to force it to resume polling the node that went down.

Some people have a cron job configured to restart gmetad every few hours just to avoid this issue.

RRA definition changed in gmetad.conf, but RRD files are unchanged

The RRA definition in *gmetad.conf* is used only when new RRD files are created. If the RRA definition is changed—for example, to increase the retention period of the data—gmetad will not apply the change to existing RRD files.

Two options are available for handling data in existing RRD files:

- Archive old RRD files and let gmetad create new RRD files based on updated RRA definition.
- Use `rrddump` to dump existing RRD files to XML format, massage them to conform to the new RRA definition, and reimport back to RRD files using `rrdrestore`. For more information about these tools, please refer to the RRDtool documentation.

rrdcached Issues

The most common issues you will face when using rrdcached relate to errors about the number of open files or file descriptors. Use the `ulimit` command to increase the permitted number of open files or file descriptors for the rrdcached process. The process must be able to open all the RRDs for all the metrics concurrently.

gmond Issues

This section contains a list of common gmetad problems and their solutions.

gmond fails to start or localhost issues

When `bind_hostname = true` (the default in recent versions of gmond), gmond will try to bind to the IP address associated with the hostname.

On some machines, the */etc/hosts* file contains an entry mapping the hostname to the localhost address (127.0.0.1). In this situation, gmond will fail. Try fixing the host's file.

gmond uses a lot of RAM

This is not always a memory leak. If the gmond memory usage grows to 2 GB or more and then gmond crashes, it is probably a memory leak or you have an extremely large number of hosts sending metrics to the gmond. It could also be a sign of a denial-of-service attack (someone sending random metrics to fill up the memory). See "valgrind: Memory Leaks and Memory Corruption" on page 116.

If the gmond memory usage is high (more than 100 MB) but constant, it is quite possible that this is just the normal amount of memory needed to keep the state information for all the metrics it is receiving from other nodes in the cluster.

gmond doesn't start properly on bootup

Verify that the `init` script is installed and has the executable bit set. Verify that the symlink from */etc/rcX.d* exists for the run-level. Verify that the host has an IP address before the gmond `init` script is invoked. If the system obtains an IP address dynamically, it is possible that DHCP is not completed before the attempt to start gmond, and so gmond fails to run. If network manager is in use (typically on desktop workstations), there is often no DHCP IP address until the user has logged in. Ganglia v3.3.7 introduced a new configuration option, `retry_bind`, that can be used to tell gmond to wait for the IP address rather than aborting if it is not ready.

UDP receives buffer errors on a machine running gmond

If you notice UDP receive buffer errors/dropped packets on a machine running gmond, you may find gmond itself to be the culprit. Check */proc/net/udp* to see how many packets are being dropped by the gmond process. If gmond is dropping packets, increase the size of the UDP receive buffer (see the buffer parameter introduced in v3.4.0). If that doesn't help, and if the gmond process is at full capacity (100 percent of a CPU core), consider reducing the rate of metric packets from all gmonds in the cluster, or break the cluster into multiple clusters.

CHAPTER 7
Ganglia and Nagios

Vladimir Vuksan, Jeff Buchbinder, and Dave Josephsen

It's been said that specialization is for insects, which although poetic, isn't exactly true. Nature abounds with examples of specialization in just about every biological kingdom, from mitochondria to clownfish. The most extreme examples are a special kind of specialization, which biologists refer to as *symbiosis*.

You've probably come across some examples of biological symbiosis at one time or another. Some are quite famous, like the clownfish and the anemone. Others, like the fig wasp, are less so, but the general idea is always the same: two organisms, finding that they can rely on each other, buddy up. Buddies have to work less and can focus more on what they're good at. In this way, symbiosis begets more specialization, and the individual specializations grow to complement each other.

Effective symbiotes are complementary in the sense that there isn't much functional overlap between them. The beneficial abilities of one buddy stop pretty close to where those of the other begin, and vice versa. They are also complementary in the sense that their individual specializations combine to create a solution that would be impossible otherwise. Together the pair become something more than the sum of their parts.

It would surprise us to learn that you'd never heard of Nagios. It is probably the most popular open source monitoring system in existence today, and is generally credited for if not inventing, then certainly perfecting the centralized polling model employed by myriad monitoring systems both commercial and free. Nagios has been imitated, forked, reinvented, and commercialized, but in our opinion, it's never been beaten, and it remains the yardstick by which all monitoring systems are measured.

It is not, however, a valid yardstick by which to measure Ganglia, because the two are not in fact competitors, but symbiotes, and the admin who makes the mistake of choosing one over the other is doing himself a disservice. It is not only possible, but advisable to use them together to achieve the best of both worlds. To that end, we've included this chapter to help you understand the best options available for Nagios interoperability.

Sending Nagios Data to Ganglia

Under the hood, Nagios is really just a special-purpose scheduling and notification engine. By itself, it can't monitor anything. All it can do is schedule the execution of little programs referred to as plug-ins and take action based on their output.

Nagios plug-ins return one of four states: 0 for "OK," 1 for "Warning," 2 for "Critical," and 3 for "Unknown." The Nagios daemon can be configured to react to these return codes, notifying administrators via email or SMS, for example. In addition to the codes, the plug-ins can also return a line of text, which will be captured by the daemon, written to a log, and displayed in the UI. If the daemon finds a pipe character in the text returned by a plug-in, the first part is treated normally, and the second part is treated as performance data.

Performance data doesn't really mean anything to Nagios; it won't, for example, enforce any rules on it or interpret it in any way. The text after the pipe might be a chili recipe, for all Nagios knows. The important point is that Nagios can be configured to handle the post-pipe text differently than pre-pipe text, thereby providing a hook from which to obtain metrics from the monitored hosts and pass those metrics to external systems (like Ganglia) without affecting the human-readable summary provided by the pre-pipe text.

Nagios's performance data handling feature is an important hook. There are quite a few Nagios add-ons that use it to export metrics from Nagios for the purpose of importing them into local RRDs. These systems typically point the `service_perfdata_command` attribute in *nagios.cfg* to a script that use a series of regular expressions to parse out the metrics and metric names and then import them into the proper RRDs. The same methodology can easily be used to push metrics from Nagios to Ganglia by pointing the `service_perfdata_command` to a script that runs gmetric instead of the RRDtool import command.

First, you must enable performance data processing in Nagios by setting `process_performance_data=1` in the *nagios.cfg* file. Then you can specify the name of the command to which Nagios should pass all performance data it encounters using the `service_perfdata_command` attribute.

Let's walk through a simple example. Imagine a `check_ping` plug-in that, when executed by the Nagios scheduler, pings a host and then return the following output:

```
PING OK - Packet loss = 0%, RTA = 0.40 ms|0;0.40
```

We want to capture this plug-in's performance data, along with details we'll need to pass to gexec, including the name of the target host. Once `process_performance_data` is enabled, we'll tell Nagios to execute our own shell script every time a plug-in returns with performance data by setting `service_perfdata_command=PushToGanglia` in *nagios.cfg*. Then we'll define `pushToGanglia` in the Nagios object configuration like so:

```
define command{
command_name    pushToGanglia
command_line  /usr/local/bin/pushToGanglia.sh
"$LASTSERVICECHECK$||$HOSTNAME$||$SERVICEDESC$||$SERVICEOUTPUT$||$SERVICEPERFDATA$"
}
```

Careful with those delimiters!

With so many Nagios plug-ins, written by so many different authors, it's important to carefully choose your delimiter and avoid using the same one returned by a plug-in. In our example command, we chose double pipes for a delimiter, which can be difficult to parse in some languages. The tilde (~) character is another good choice.

The capitalized words surrounded by dollar signs in the command definition are Nagios macros. Using macros, we can request all sorts of interesting details about the check result from the Nagios daemon, including the nonperformance data section of the output returned from the plug-in. The Nagios daemon will substitute these macros for their respective values at runtime, so when Nagios runs our `pushToGanglia` command, our input will wind up looking something like this:

```
1338674610||dbaHost14.foo.com||PING||PING OK - Packet loss = 0%, RTA = 0.40 ms||0;0.40
```

Our *pushToGanglia.sh* script will take this input and compare it against a series of regular expressions to detect what sort of data it is. When it matches the `PING` regex, the script will parse out the relevant metrics and push them to Ganglia using gexec. It looks something like this:

```
#!/bin/sh
while read IN
do
    #check for output from the check_ping plug-in
    if [ "$(awk -F '[|][|]' '$3 ~ /^PING$/' <<<${IN})" ]
    then

        #this looks like check_ping output all right, parse out what we need
        read BOX CMDNAME PERFOUT <<<$(awk -F '[|][|]' '{print $2" "$3" "$5}'<<<${IN})
        read PING_LOSS PING_MS <<<$(tr ';' ' '<<<${PERFOUT})

        #Ok, we have what we need. Send it to Ganglia.
        gmetric -S ${BOX} -n ${CMDNAME} -t PING_MS -v ${PING_MS}
        gmetric -S ${BOX} -n ${CMDNAME} -t PING_LOSS -v ${PING_LOSS}

    #check for output from the check_cpu plug-in
    elif [ "$(awk -F '[|][|]' '$3 ~ /^CPU$/' <<<${IN})" ]
    then
        #do the same sort of thing but with CPU data
    fi
done
```

This is a popular solution because it's self-documenting, keeps all of the metrics collection logic in a single file, detects new hosts without any additional configuration, and works with any kind of Nagios check result, including passive checks. It does, however, add a nontrivial amount of load to the Nagios server. Consider that any time you add a new check, the result of that check for every host must be parsed against the `pushToGanglia` script. The same is true when you add a new host or even a new regex to the `pushToGanglia` script. In Nagios, `process_performance_data` is a global setting, and so are the ramifications that come with enabling it.

It probably makes sense to process performance data globally if you rely heavily on Nagios for metrics collection. However, for the reasons we outlined in Chapter 1, we don't think that's a good idea. If you're using Ganglia along with Nagios, gmond is the better-evolved symbiote for collecting the normal litany of performance metrics. It's more likely that you'll want to use gmond to collect the majority of your performance metrics, and less likely that you'll want Nagios churning through the result of every single check in case there might be some metrics you're interested in sending over to Ganglia.

If you're interested in metrics from only a few Nagios plug-ins, consider leaving the metric `process_performance_data` disabled and instead writing "wrappers" for the interesting plug-ins. Here, for example, is what a wrapper for the `check_ping` plug-in might look like:

```
#!/bin/sh

ORIG_PLUGIN='/usr/libexec/check_ping_orig'

#get the target host from the H option
while getopts "H:" opt
do
    if [ "${opt}" == 'H' ]
    then
        BOX=${OPTARG}
    fi
done

#run the original plug-in with the given options, and capture its output
OOUT=$(${ORIG_PLUGIN} $@)
OEXIT=$?

#parse out the perfdata we need
read PING_LOSS PING_MS <<<$(echo ${OOUT} | cut -d\| -f2 | tr ";" " ")

#send the metrics to Ganglia
gmetric -S ${BOX} -n ${CMDNAME} -t PING_MS -v ${PING_MS}
gmetric -S ${BOX} -n ${CMDNAME} -t PING_LOSS -v ${PING_LOSS}

#mimic the original plug-in's output back to Nagios
echo "${OOUT}"
exit ${OEXIT}
```

The wrapper approach takes a huge burden off the Nagios daemon but is more difficult to track. If you don't carefully document your changes to the plug-ins, you'll mystify other administrators, and upgrades to the Nagios plug-ins will break your data collection efforts.

The general strategy is to replace the `check_ping` plug-in with a small shell script that calls the original `check_ping`, intercepts its output, and sends the interesting metrics to Ganglia. The imposter script then reports back to Nagios with the output and exit code of the original plug-in, and Nagios has no idea that anything extra has transpired. This approach has several advantages, the biggest of which is that you can pick and choose which plug-ins will process performance data.

Monitoring Ganglia Metrics with Nagios

Because Nagios has no built-in means of polling data from remote hosts, Nagios users have historically employed various remote execution schemes to collect a litany of metrics with the goal of comparing them against static thresholds. These metrics, such as the available disk space or CPU utilization of a host, are usually collected by services like NSCA or NRPE, which execute scripts on the monitored systems at the Nagios server's behest, returning their results in the standard Nagios way. The metrics themselves, once returned, are usually discarded or in some cases fed into RRDs by the Nagios daemon in the manner described previously.

This arrangement is expensive, especially considering that most of the metrics administrators tend to collect with NRPE and NSCA are collected by gmond out of the box. If you're using Ganglia, it's much cheaper to point Nagios at Ganglia to collect these metrics.

To that end, the Ganglia project began including a series of official Nagios plug-ins in gweb versions as of 2.2.0. These plug-ins enable Nagios users to create services that compare metrics stored in Ganglia against alert thresholds defined in Nagios. This is, in our opinion, a huge win for administrators, in many cases enabling them to scrap entirely their Nagios NSCA infrastructure, speed up the execution time of their service checks, and greatly reduce the monitoring burden on both Nagios and the monitored systems themselves.

There are five Ganglia plug-ins currently available:

1. Check heartbeat.
2. Check a single metric on a specific host.
3. Check multiple metrics on a specific host.
4. Check multiple metrics across a regex-defined range of hosts.
5. Verify that one or more values is the same across a set of hosts.

Principle of Operation

The plug-ins interact with a series of gweb PHP scripts that were created expressly for the purpose. See Figure 7-1. The *check_host_regex.sh* plug-in, for example, interacts with the PHP script: "http://your.gweb.box/nagios/check_host_regex.php". Each PHP script takes the arguments passed from the plug-in and parses a cached copy of the XML dump of the grid state obtained from gmetad's `xml_port` to retrieve the current metric values for the requested entities and return a Nagios-style status code (see "gmetad" on page 33 for details on gmetad's `xml_port`). You must functionally enable the server-side PHP scripts before they can be used and also define the location and refresh interval of the XML grid state cache by setting the following parameters in the gweb *conf.php* file:

```
$conf['nagios_cache_enabled'] = 1;
$conf['nagios_cache_file'] = $conf['conf_dir'] . "/nagios_ganglia.cache";
$conf['nagios_cache_time'] = 45;
```

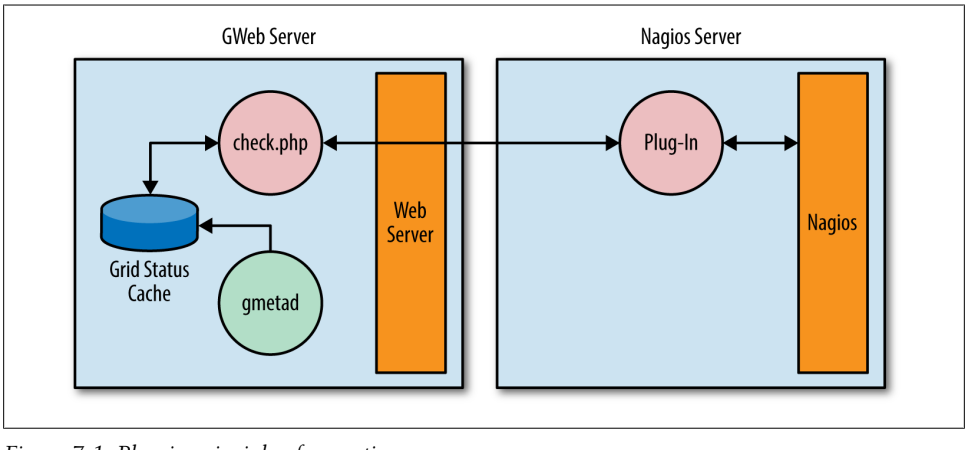
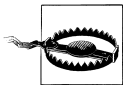


Figure 7-1. Plug-in principle of operation

Consider storing the cache file on a RAMDisk or tmpfs to increase performance.



Beware: Numerous parallel checks

If you define a service check in Nagios to use hostgroups instead of individual hosts, Nagios will schedule the service check for all hosts in that hostgroup at the same time, which may cause a race condition if gweb's grid state cache changes before the service checks finish executing. To avoid cache-related race conditions, use the *warmup_metric_cache.sh* script in the *web/nagios* subdirectory of the gweb tarball, which will ensure that your cache is always fresh.

Check Heartbeat

Internally, Ganglia uses a heartbeat counter to determine whether a machine is up. This counter is reset every time a new metric packet is received for the host, so you can safely use this plug-in in lieu of the Nagios `check_ping` plug-in. To use it, first copy the *check_heartbeat.sh* script from the Nagios subdirectory in the Ganglia Web tarball to your Nagios plug-ins directory. Make sure that the `GANGLIA_URL` inside the script is correct. By default, it is set to:

```
GANGLIA_URL="http://localhost/ganglia2/nagios/check_heartbeat.php"
```

Next, define the check command in Nagios. The threshold is the amount of time since the last reported heartbeat; that is, if the last packet received was 50 seconds ago, you would specify 50 as the threshold:

```
define command {
  command_name  check_ganglia_heartbeat
  command_line  $USER1$/check_heartbeat.sh host=$HOSTADDRESS$ threshold=$ARG1$
}
```

Now for every host/host group, you want the monitored change `check_command` to be:

```
check_command  check_ganglia_heartbeat!50
```

Check a Single Metric on a Specific Host

The `check_ganglia_metric` plug-in compares a single metric on a given host against a predefined Nagios threshold. To use it, copy the *check_ganglia_metric.sh* script from the Nagios subdirectory in the Ganglia Web tarball to your Nagios plug-ins directory. Make sure that the `GANGLIA_URL` inside the script is correct. By default, it is set to:

```
GANGLIA_URL="http://localhost/ganglia2/nagios/check_metric.php"
```

Next, define the check command in Nagios like so:

```
define command {
  command_name  check_ganglia_metric
  command_line  $USER1$/check_ganglia_metric.sh host=$HOSTADDRESS$↩
  metric_name=$ARG1$ operator=$ARG2$ critical_value=$ARG3$
}
```

Next, add the check command to the service checks for any hosts you want monitored. For instance, if you wanted to be alerted when the 1-minute load average for a given host goes above 5, add the following directive:

```
check_command            check_ganglia_metric!load_one!more!5
```

To be alerted when the disk space for a given host falls below 10 GB, add:

```
check_command            check_ganglia_metric!disk_free!less!10
```

Operators denote criticality

The operators specified in the Nagios definitions for the Ganglia plug-ins always indicate the "critical" state. If you use a `notequal` operator, it means that state is critical if the value is *not* equal.

Check Multiple Metrics on a Specific Host

The `check_multiple_metrics` plug-in is an alternate implementation of the *check_ganglia_metric* script that can check multiple metrics on the same host. For example, instead of configuring separate checks for disk utilization on /, */tmp*, and */var*—which could produce three separate alerts—you could instead set up a single check that alerted any time disk utilization fell below a given threshold.

To use it, copy the *check_multiple_metrics.sh* script from the Nagios subdirectory of the Ganglia Web tarball to your Nagios plug-ins directory. Make sure that the variable `GANGLIA_URL` in the script is correct. By default, it is set to:

```
GANGLIA_URL="http://localhost/ganglia2/nagios/check_multiple_metrics.php"
```

Then define a check command in Nagios:

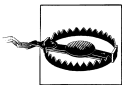
```
define command {
  command_name  check_ganglia_multiple_metrics
  command_line  $USER1$/check_multiple_metrics.sh host=$HOSTADDRESS$ checks='$ARG1$'
}
```

Then add a list of checks that are delimited with a colon. Each check consists of:

```
metric_name,operator,critical_value
```

For example, the following service would monitor the disk utilization for root (/) and */tmp*:

```
check_command check_ganglia_multiple_metrics!disk_free_rootfs,less,↩
10:disk_free_tmp,less,20
```



Beware: Aggregated services

Anytime you define a single service to monitor multiple entities in Nagios, you run the risk of losing visibility into "compound" problems. For example, a service configured to monitor both */tmp* and */var* might only notify you of a problem with */tmp*, when in fact both partitions have reached critical capacity.

Check Multiple Metrics on a Range of Hosts

Use the `check_host_regex` plug-in to check one or more metrics on a regex-defined range of hosts. This plug-in is useful when you want to get a single alert if a particular metric is critical across a number of hosts.

To use it, copy the *check_host_regex.sh* script from the Nagios subdirectory in Ganglia Web tarball to your Nagios plug-ins directory. Make sure that the `GANGLIA_URL` inside the script is correct. By default, it is:

```
GANGLIA_URL="http://localhost/ganglia2/nagios/check_host_regex.php"
```

Next, define a check command in Nagios:

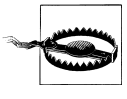
```
define command {
  command_name  check_ganglia_host_regex
  command_line  $USER1$/check_host_regex.sh hreg='$ARG1$' checks='$ARG2$'
}
```

Then add a list of checks that are delimited with a colon. Each check consists of:

```
metric_name,operator,critical_value
```

For example, to check free space on */* and */tmp* for any machine starting with `web-*` or `app-*` you would use something like this:

```
check_command check_ganglia_host_regex!^web-|^app-!disk_free_rootfs,less,↩
10:disk_free_tmp,less,10
```



Beware: Multiple hosts in a single service

Combining multiple hosts into a single service check will prevent Nagios from correctly respecting host-based external commands. For example, Nagios will send notifications if a host listed in this type of service check goes critical, even if the user has placed the host in scheduled downtime. Nagios has no way of knowing that the host has anything to do with this service.

Verify that a Metric Value Is the Same Across a Set of Hosts

Use the `check_value_same_everywhere` plug-in to verify that one or more metrics on a range of hosts have the same value. For example, let's say you wanted to make sure the SVN revision of the deployed program listing was the same across all servers. You could send the SVN revision as a string metric and then list it as a metric that needs to be the same everywhere.

To use the plug-in, copy the *check_value_same_everywhere.sh* script from the Nagios subdirectory of the Ganglia Web tarball to your Nagios plug-ins directory. Make sure that the `GANGLIA_URL` variable inside the script is correct. By default, it is:

```
GANGLIA_URL="http://localhost/ganglia2/nagios/check_value_same_everywhere.php"
```

Then define a check command in Nagios:

```
define command {
  command_name  check_value_same_everywhere
  command_line  $USER1$/check_value_same_everywhere.sh hreg='$ARG1$' checks='$ARG2$'
}
```

For example:

```
check_command check_value_same_everywhere!^web-|^app-!svn_revision,num_config_files
```

Displaying Ganglia Data in the Nagios UI

In Nagios 3.0, the `action_url` attribute was added to the host and service object definitions. When specified, the `action_url` attribute creates a small icon in the Nagios UI next to the host or service name to which it corresponds. If a user clicks this icon, the UI will direct them to the URL specified by the `action_url` attribute for that particular object.

If your host and service names are consistent in both Nagios and Ganglia, it's pretty simple to point any service's `action_url` back to Ganglia's *graph.php* using built-in Nagios macros so that when a user clicks on the `action_url` icon for that service in the Nagios UI, he or she is presented with a graph of that service's metric data. For example, if we had a host called host1, with a service called load_one representing the one-minute load history, we could ask Ganglia to graph it for us with:

```
http://my.ganglia.box/graph.php?c=cluster1&h=host1&m=load1&r=hour&z=large
```

The hiccup, if you didn't notice, is that Ganglia's *graph.php* requires a `c=` attribute, which must be set to the name of the cluster to which the given host belongs. Nagios has no concept of Ganglia clusters, but it does provide you with the ability to create custom variables in any object definition. Custom variables must begin with an underscore, and are available as macros in any context a built-in macro would be available. Here's an example of a custom variable in a host object definition defining the Ganglia cluster name to which the host belongs:

```
define host{
    host_name        host1
    address        192.168.1.1
    _ganglia_cluster    cluster1
    ...
}
```

Read more about Nagios Macros here.

You can also use custom variables to correct differences between the Nagios and Ganglia namespaces, creating, for example, a `_ganglia_service_name` macro in the service definition to map a service called "CPU" in Nagios to a metric called "load_one" in Ganglia.

To enable the `action_url` attribute, we find it expedient to create a template for the Ganglia `action_url`, like so:

```
define service {
   name       ganglia-service-graph
   action_url http://my.ganglia.host/ganglia/graph.php?c=$_GANGLIA_CLUSTER$&↩
              h=$HOSTNAME$&m=$SERVICEDESC$&r=hour&z=large
   register   0
}
```

This code makes it easy to toggle the `action_url` graph for some services but not others by including `use ganglia-service-graph` in the definition of any service that you want to graph. As you can see, the `action_url` we've specified combines the custom-made `_ganglia_cluster` macro we defined in the host object with the hostname and `service desc` built-in macros. If the Nagios service name was not the same as the Ganglia metric name (which is likely the case in real life), we would have defined our own `_gan glia_service_name` variable in the service definition and referred to that macro in the `action_url` instead of the `servicedesc` built-in.

The Nagios UI also supports custom CGI headers and footers, which make it possible to accomplish rollover popups of the `action_url` icon containing graphs from the Ganglia *graph.php*. This approach requires some custom development on your part and is outside the scope of this book, but we wanted you to know it's there. If that sounds like a useful feature to you, we suggest checking out this information.

Monitoring Ganglia with Nagios

When Ganglia is running, it's a great way to aggregate metrics, but when it breaks, it can cause a bit of frustration with regard to locating the cause of that breakage. Thankfully, there are a number of points to monitor, which can help stave off an inconvenient breakage.

Monitoring Processes

Using `check_nrpe` (or even `check_procs` directly), the daemons that support Ganglia can be monitored for any failures. It is most useful to monitor gmetad and rrdcached on the aggregation hosts and gmond on all hosts. The pertinent snippets for local monitoring of a gmond process are:

```
define command {
  command_name    check_gmond_local
  command_line    $USER1$/check_procs -C gmond -c 1:2
  }

define service {
  use                       generic-service
  host_name                 localhost
  service_description       GMOND
  check_command             check_gmond_local
  }
```

Monitoring Connectivity

A more "functional" type of monitoring is monitoring for connectivity on the outbound TCP ports for the varying services. gmetad, for example, listens on ports 8651 and 8652, and gmond listens on port 8649. Checking these ports, with a reasonable timeout, can give a reasonably good idea as to whether they are functioning as expected.

Monitoring cron Collection Jobs

cron collection jobs, which are run by your cron periodic scheduling daemon, are another way of collecting metrics without using gmond modules. Monitoring failures in these scripts, by virtue of their extremely heterogeneous nature and lack of similar structures, has the potential for being a place for fairly serious collection failures. These can, for the most part, be avoided by following a few basic suggestions

Log, but not too much.
: Using the logger utility for bash scripts or any of the variety of syslog submission capabilities available will allow you to be able to see what your scripts are doing, instead of being bombarded by logwatch emails or just seeing collection for certain metrics stop.

Use "last run" files.
: Touch a stamp file to allow other monitoring tools to detect the last run of your script. That way, you can monitor the stamp file for becoming stale in a standard way. Be wary of permissions issues, as test-running a script as a user other than the one who will be running it in production can cause silent failures.

Expect bad data.
: Too many cron jobs are written to collect data, but assume things like "the network is always available," "a file I'm monitoring exists," or "some third-party dependency will never fail." These will eventually lead to error conditions that either break collection completely or, worse, submit incorrect metrics.

Use timeouts.
: If you're using netcat, telnet, or other network-facing methods to gather metrics data, there is a possibility that they will fail to return data before the next polling period, potentially causing a pile-up or resulting in other nasty behavior. Use common sense to figure out how long you should be waiting for results, then exit gracefully if you haven't gotten them.

Collecting rrdcached Metrics

It can be useful to collect metrics on the backlog and processing metrics for your rrdcached services (if you are using them to speed up your gmetad host). This can be done by querying the rrdcached `stats` socket and pushing those metrics into Ganglia using gmetric.

Excessive backlogs can be caused by high IO or CPU load on your rrdcached server, so this can be a useful tool to track down rogue cron jobs or other root causes:

```
#!/bin/bash
# rrdcache-stats.sh
#
# SHOULD BE RUN AS ROOT, OTHERWISE SUDO RULES NEED TO BE PUT IN PLACE
# TO ALLOW THIS SCRIPT, SINCE THE SOCKET IS NOT ACCESSIBLE BY NORMAL
# USERS!

GMETRIC="/usr/bin/gmetric"
RRDSOCK="unix:/var/rrdtool/rrdcached/rrdcached.sock"
EXPIRE=300

( echo "STATS"; sleep 1; echo "QUIT" ) | \
  socat - $RRDSOCK | \
  grep ':' | \
  while read X; do
    K="$( echo "$X" | cut -d: -f1 )"
    V="$( echo "$X" | cut -d: -f2 )"
    $GMETRIC -g rrdcached -t uint32 -n "rrdcached_stat_${K}" -v ${V} -x ${EXPIRE}
    -d ${EXPIRE} | \
  done
```

CHAPTER 8
Ganglia and sFlow

Peter Phaal

Ganglia's gmond agent already has built-in metrics and can be extended using plug-in modules—why do I need to know about sFlow? The short answer is that sFlow agents are available for platforms such as Windows servers and hypervisors that aren't currently supported by gmond. A longer answer requires a basic understanding of how sFlow integrates with Ganglia to extend coverage and improve efficiency.

There are strong parallels between Ganglia's approach to monitoring large numbers of servers and the sFlow standard used to monitor the switches connecting them. The scalability challenge of monitoring the network links mirrors the challenge of monitoring servers because each server has at least one link to the network. However, the constraints are different, leading to divergence in the functional split between generating and consuming metrics.

Network switches perform most of their functionality in hardware and have limited processing and memory resources. While computational resources are scarce, switches are richly connected to the network and excel at sending packets. With sFlow, raw metrics from the switches are sent over the network to a central server, exploiting the relatively abundant network resources to shift processing and state from the switches to software running on the server. Removing state from the switches minimizes the memory footprint and eliminates the need to dynamically allocate memory—both useful properties when embedding the agent in switch firmware.

Unlike server metrics, switch metrics are largely implemented in hardware. For example, byte and packet counts for each switch port are implemented as hardware counters. Standards are critical: traffic passes through many devices often from different vendors and they need to agree on how to quantify the traffic. A core part of sFlow is the specification of standard sets of metrics, allowing each switch vendor to embed the measurements in hardware and produce interoperable results.

Ganglia's binary protocol uses XDR to efficiently encode metrics and send them as UDP packets. However, each packet contains only a single metric and additional packets are

needed to transmit metadata describing the metrics. For example, a host sending the 30 basic metrics every 15 seconds will generate an average of 2 packets per second and a cluster of 1,000 servers will generate 2,000 packets per second of measurement traffic. In contrast, the sFlow protocol encodes standard blocks of metrics as XDR structures, allowing a host to send all 30 metrics in a single packet and requiring only 67 packets per second to monitor the entire thousand node cluster.

Another difference between the Ganglia and sFlow binary protocols is that the default Ganglia configuration multicasts the packets, meaning that every link in the 1,000-node cluster carries 2,000 packets per second and every host needs to process 2,000 packets per second. In contrast, sFlow is a unicast protocol, allowing the network to isolate measurement traffic to individual links. Most links will carry only one sFlow packet every 15 seconds and only the link connecting to the sFlow analyzer will carry the full 67 packets per second. The increased efficiency of the sFlow protocol allows 30,000 servers to be monitored with the same network overhead as gmond requires to monitor 1,000 servers.

Ganglia gmond agents can also be deployed in a unicast configuration. For large clusters, switching to unicast improves scalability by reducing the amount of memory, CPU, and network resources consumed on each host in the cluster.

For most applications, the difference in scalability isn't significant, but the improved efficiency of using sFlow as the measurement transport helps Ganglia monitor the extremely large numbers of physical and virtual servers found in cloud data centers.

Standardizing the metrics helps reduce operational complexity by eliminating configuration options that would be needed for a more flexible solution. Again, this is very important in multivendor networking environments where each configuration option needs to be added to custom device firmware. Generally, sFlow agents export all the metrics they are capable of generating and leave it up to the analyzer to decide which metrics to keep or discard. This approach may seem wasteful, but often the measurements are sent to multiple applications, each of which is interested in different metrics. Maintaining complex, per-application state in the agents consumes significant resources and becomes a challenging configuration task as matching application and agent configuration settings need to be maintained. Shifting the task of measurement selection to the collector frees up agent resources and reduces configuration complexity.

Security of network devices is also of paramount concern and sFlow agents are intrinsically secure against remote intrusion attacks because they send but never receive or process packets.

At this point, you may be wondering how sFlow agents relate to server monitoring, as most of the discussion has been about the challenges of embedding monitoring in network switches. Although it is easy to deploy Ganglia agents and script custom metrics

on a cluster of Linux servers, monitoring a large pool of virtual servers is a different matter. In many ways, hypervisors have more in common with switches than they do with a general-purpose server. The hypervisor acts as a virtual switch, connecting virtual machines to each other and to the physical network. Just like the management processor on a switch, the hypervisor is a tightly controlled, highly secure environment with limited CPU and memory resources. The sFlow agent is designed for embedded environments and is a natural fit for hypervisors.

Instrumenting applications poses similar challenges. For example, `mod_sflow` embeds sFlow instrumentation in the Apache web server. The `mod_sflow` agent has a minimal footprint and a negligible impact on the performance of the web server. The alternative of tailing the web server log files in order to derive metrics has a much greater overhead that can become prohibitive for high traffic servers. Similar to the network, there is a value in defining standard metrics in the application space. For example, the Apache, NGINX, and Tomcat sFlow agents generate the same set of HTTP metrics, allowing web servers to be monitored interchangeably using a variety of performance analysis tools.

Metrics charts are an extremely useful way of summarizing large amounts of information, making them a staple of operations dashboards. For example, each data point in a chart trending HTTP activity may summarize information from hundreds of thousands of HTTP requests. However, metrics can only take you so far; how do you follow up if you see an unusual spike in HTTP requests? With sFlow monitoring, metrics are only one part of the measurement stream; an sFlow agent also exports records describing randomly sampled transactions, providing detailed visibility into transaction attributes, data volumes, response times, and status codes (for more information on sFlow's random sampling mechanism, see Packet Sampling Basics). The examples in "Web load" on page 177 and "Optimizing memcached efficiency" on page 175 illustrate how analysis of sFlow's sampled transactions provides additional detail that complements Ganglia's trend charts.

The remainder of this chapter describes in detail the architecture of a Ganglia and sFlow deployment, standard sFlow metrics, configuration, troubleshooting, and integration with the broader set of sFlow analysis tools. If you would like to see how Ganglia and sFlow monitoring works in practice before diving into the details, see "Tagged, Inc." on page 172.

Architecture

In a classic Ganglia deployment, gmond agents are installed on each host in a cluster; see "gmond: Big Bang in a Few Bytes" on page 4. Each gmond agent performs three tasks:

1. Monitoring the performance of its host and sharing the metrics with other hosts in the cluster by sending multicast messages

2. Listening for updates from other hosts in the cluster in order to monitor cluster state
3. Responding to requests from gmetad for XML snapshots of the cluster state

In an sFlow deployment, sFlow agents replace gmond agents on all the hosts within the cluster; see Figure 8-1. The sFlow agents in each cluster send metrics as unicast messages to a single gmond instance that tracks cluster state and responds to requests from gmetad. As you will see in "Configuring gmond to Receive sFlow" on page 155, deploying gmond as an sFlow collector requires minimal configuration and eliminates dependencies because functionality associated with generating and transmitting metrics is disabled.

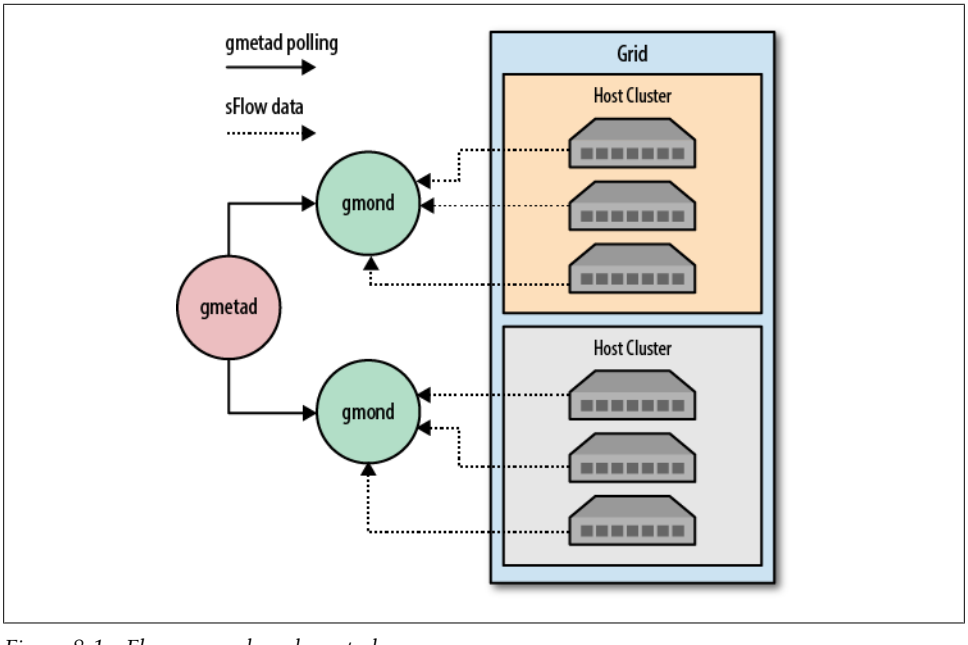


Figure 8-1. sFlow, gmond, and gmetad

The sFlow agents export standard groups of metrics; see the section "Standard sFlow Metrics" on page 147. However, supplementing sFlow's standard metrics with additional custom metrics can be accomplished using gmetric; see "Custom Metrics Using gmetric" on page 160.

A single gmond instance monitoring each cluster does represent a single point of failure. If this is a concern, there are a number of strategies for making the deployment fault tolerant. A second gmond instance can be added to each cluster and the sFlow agents can be configured to send metrics to both the primary and secondary gmond instances. Alternatively, a virtual IP address can be assigned to gmond and used as the destination for sFlow messages and gmetad requests. The virtual IP address can be handed to a secondary system in the event that the primary system fails. Running gmond instances

on virtual machines makes it easy to quickly bring up replacements in the event of a failure.

An entire cluster should be homogeneously monitored using sFlow agents or gmond agents—mixing sFlow and gmond agents within a single cluster is not recommended. However, you can adopt different measurement technologies within a grid, selecting the best strategy for monitoring each cluster. Using sFlow as the agent technology works best for commodity web, memcache, virtual server, and Java clusters where sFlow's standard metrics provide good coverage. For specialized environments, gmond's extensibility and extensive library of modules are likely to be a better option.

Standard sFlow Metrics

Current approaches to server performance monitoring are highly fragmented. Each operating system, server vendor, and application developer creates specific agents and software for performance monitoring, none of which interoperate. Standardizing the metrics simplifies monitoring by decoupling agents from performance monitoring applications, allowing measurements to be made once and shared among different monitoring applications.

Server Metrics

The standard set of sFlow server metrics are a superset of Ganglia's base metrics; see "Base Metrics" on page 75. Base Ganglia metrics are indicated by an asterisk next to the metric name, as seen in Table 8-1.

Table 8-1. sFlow server metrics

Metric Name	Description	Type
machine_type*	Machine Type	system
os_name*	Operating System	system
os_release*	Operating System Release	system
uuid	System UUID	System
heartbeat*	Heartbeat	System
load_one*	One-minute Load Average	load
load_five*	Five-minute Load Average	load
load_fifteen*	Fifteen-minute Load Average	load
proc_run*	Total Running Processes	process
proc_total*	Total Processes	process
cpu_num*	CPU Count	cpu
cpu_speed*	CPU Speed	cpu
boottime*	Last Boot Time	cpu

Metric Name	Description	Type
cpu_user*	CPU User	cpu
cpu_nice*	CPU Nice	cpu
cpu_system*	CPU System	cpu
cpu_ide*	CPU Idle	cpu
cpu_wio*	CPU I/O Wait	cpu
cpu_intr*	CPU Interrupts	cpu
cpu_sintr*	CPU Soft Interrupts	cpu
interrupts	Interrupts	cpu
contexts	Context Switches	cpu
mem_total*	Memory Total	memory
mem_free*	Free Memory	memory
mem_shared*	Shared Memory	memory
mem_buffers*	Memory Buffers	memory
mem_cached*	Cached Memory	memory
swap_total*	Swap Space Total	memory
swap_free*	Free Swap Space	memory
page_in	Pages In	memory
page_out	Pages Out	memory
swap_in	Swap Pages In	memory
swap_out	Swap Pages Out	memory
disk_total*	Total Disk Space	disk
disk_free*	Free Disk Space	disk
part_max_used*	Maximum Disk Space Used	disk
reads	Reads	disk
bytes_read	Bytes Read	disk
read_time	Read Time	disk
writes	Writes	disk
bytes_written	Bytes Written	disk
write_time	Write Time	disk
bytes_in*	Bytes Received	network
pkts_in*	Packets Received	network
errs_in	Input Errors	network
drops_in	Input Drops	network
bytes_out*	Bytes Sent	network
pkts_out*	Packets Sent	network

Metric Name	Description	Type
errs_out	Output Errors	network
drops_out	Output Drops	network

The overlap between Ganglia and sFlow server metrics is no coincidence. One of the major contributions of the Ganglia project was identifying a common set of metrics that summarize server performance and are portable across operating systems; the Ganglia base metrics were used as a starting point for defining the standard sFlow server metrics.

Hypervisor Metrics

The standard set of sFlow hypervisor and virtual machine metrics (Table 8-2) are based on metrics defined by the open source libvirt project. The libvirt project has created a common set of tools for managing virtualization resources on different virtualization platforms, currently including: Xen, QEMU, KVM, LXC, OpenVZ, User Mode Linux, VirtualBox, and VMware ESX and GSX. The sFlow metrics provide consistency between different virtualization platforms and between sFlow- and libvirt-based performance monitoring systems.

Table 8-2. sFlow hypervisor metrics

Metric Name	Description	Type
vnode_mem_total	Hypervisor Memory Total	hypervisor
vnode_mem_free	Hypervisor Free Memory	hypervisor
vnode_cpu_speed	Hypervisor CPU Speed	hypervisor
vnode_cpu_num	Hypervisor CPU Count	hypervisor
vnode_domains	Hypervisor Domain Count	hypervisor
<VM name>.vcpu_state	<VM name>: VM CPU State	vm cpu
<VM name>.vcpu_util	<VM name>: VM CPU Utilization	vm cpu
<VM name>.vcpu_num	<VM name>: VM CPU Count	vm cpu
<VM name>.vmem_total	<VM name>: VM Memory Total	vm memory
<VM name>.vmem_util	<VM name>: VM Memory Utilization	vm memory
<VM name>.vdisk_capacity	<VM name>: VDisk Capacity	vm disk
<VM name>.vdisk_total	<VM name>: VDisk Space	vm disk
<VM name>.vdisk_free	<VM name>: Free VDisk Space	vm disk
<VM name>.vdisk_reads	<VM name>: VM Reads	vm disk
<VM name>.vdisk_bytes_read	<VM name>: VM Bytes Read	vm disk
<VM name>.vdisk_writes	<VM name>: VM Writes	vm disk
<VM name>.vdisk_bytes_written	<VM name>: VM Bytes Written	vm disk

Metric Name	Description	Type
<VM name>.vdisk_errs	<VM name>: VM Disk Errors	vm disk
<VM name>.vbytes_in	<VM name>: VM Bytes Received	vm network
<VM name>.vpkts_in	<VM name>: VM Packets Received	vm network
<VM name>.verrs_in	<VM name>: VM Input Errors	vm network
<VM name>.vdrops_in	<VM name>: VM Input Drops	vm network
<VM name>.vbytes_out	<VM name>: VM Bytes Sent	vm network
<VM name>.vpkts_out	<VM name>: VM Packets Sent	vm network
<VM name>.verrs_out	<VM name>: VM Output Errors	vm network
<VM name>.vdrops_out	<VM name>: VM Output Drops	vm network

Per virtual machine statistics are distinguished in Ganglia by prefixing the statistic by the virtual machine name.

Java Virtual Machine Metrics

The sFlow Java Virtual Machine (JVM) metrics (Table 8-3) are based on the metrics exposed through the Java Management Extensions (JMX) interface, ensuring consistency with existing JMX-based monitoring systems.

Table 8-3. sFlow Java virtual machine metrics

Metric Name	Description	Type
jvm_release	JVM Release	jvm
jvm_vcpu_util	JVM CPU Utilization	jvm
jvm_vmem_total	JVM Memory Total	jvm
jvm_vmem_util	JVM Memory Utilization	jvm
jvm_hmem_initial	JVM Heap Initial	jvm
jvm_hmem_used	JVM Heap Used	jvm
jvm_hmem_committed	JVM Heap Committed	jvm
jvm_hmem_max	JVM Heap Max	jvm
jvm_nhmem_initial	JVM Non-Heap Initial	jvm
jvm_nhmem_used	JVM Non-Heap Used	jvm
jvm_nhmem_committed	JVM Non-Heap Committed	jvm
jvm_nhmem_max	JVM Non-Heap Max	jvm
jvm_gc_count	JVM GC Count	jvm
jvm_gc_ms	JVM GC mS	jvm

Metric Name	Description	Type
jvm_cls_loaded	JVM Classes Loaded	jvm
jvm_cls_total	JVM Classes Total	jvm
jvm_cls_unloaded	JVM Classes Unloaded	jvm
jvm_comp_ms	JVM Compilation ms	jvm
jvm_thread_live	JVM Threads Live	jvm
jvm_thread_daemon	JVM Threads Daemon	jvm
jvm_thread_started	JVM Threads Started	jvm
jvm_fds_open	JVM FDs Open	jvm
jvm_fds_max	JVM FDs Max	jvm

By default, Ganglia assumes that there is a single JVM instance per host. If hosts contain more than one JVM instance, setting `multiple_jvm_instances=yes` in the gmond configuration file causes gmond to prefix each metric and description with a distinct virtual machine name; for example, the `jvm_hmem_initial` metric becomes `<name>.jvm_hmem_initial` with description <name>: JVM Heap Initial. See "Configuring gmond to Receive sFlow" on page 155.

HTTP Metrics

The sFlow HTTP metrics (Table 8-4) report on web server traffic by HTTP method and status class.

Table 8-4. sFlow HTTP metrics

Metric Name	Description	Type
http_meth_option	HTTP Method OPTION	httpd
http_meth_get	HTTP Method GET	httpd
http_meth_head	HTTP Method HEAD	httpd
http_meth_post	HTTP Method POST	httpd
http_meth_put	HTTP Method PUT	httpd
http_meth_delete	HTTP Method DELETE	httpd
http_meth_trace	HTTP Method TRACE	httpd
http_meth_connect	HTTP Method CONNECT	httpd
http_meth_other	HTTP Method other	httpd
http_status_1xx	HTTP Status 1XX	httpd
http_status_2xx	HTTP Status 2XX	httpd
http_status_3xx	HTTP Status 3XX	httpd

Metric Name	Description	Type
http_status_4xx	HTTP Status 4XX	httpd
http_status_5xx	HTTP Status 5XX	httpd
http_status_other	HTTP Status other	httpd

By default, Ganglia assumes that there is a single HTTP sFlow instance per host. If hosts contain more than one HTTP instance, setting `multiple_http_instances=yes` in the gmond configuration file causes gmond to prefix each metric and description with the TCP port number that the instance uses to receive HTTP requests; for example, the `http_meth_option` metric becomes `<port>.http_meth_option` with description <port>: HTTP Method OPTION. See "Configuring gmond to Receive sFlow" on page 155.

In addition, an HTTP sFlow agent exports HTTP operation records for randomly sampled HTTP requests. See Table 8-5.

Table 8-5. sFlow HTTP operation attributes

Attribute Name	Description
http_method	HTTP method (i.e., GET, HEAD, POST, etc.)
http_version	HTTP protocol version (i.e., 1, 1.1, etc.)
http_uri	URI exactly as it came from the client
http_host	Host value from request header
http_referer	Referer value from request header
http_useragent	User-Agent value from request header
http_xff	X-Forwarded-For value from request header
http_authuser	RFC 1413 identity of user
http_mimetype	Mime-Type of the response
http_req_bytes	Content-Length of request
http_resp_bytes	Content-Length of response
http_uS	Duration of the operation (in microseconds)
http_status	HTTP status code
socket_protocol	IP protocol type (e.g., TCP, UDP, etc.)
socket_local_ip	IP address of memcache server
socket_remote_ip	IP address of memcache client
socket_local_port	Server TCP/UDP port number
socket_remote_port	Client TCP/UDP port number

The HTTP operation records contain a superset of the attributes in the widely supported Combined Logfile Format (CLF) commonly used in web server logging. The section "Using Ganglia with Other sFlow Tools" on page 165 describes how sFlow can be converted into CLF for use with logfile analyzers.

memcache Metrics

The sFlow memcache statistics are consistent with the statistics reported by the memcache `STATS` command, ensuring consistency with existing memcache monitoring tools. The sFlow memcache metrics (Table 8-6) are a superset of those reported by the gmond module; see also Table A-4. Metrics that are present in the gmond module are indicated with an asterisk next to the metric name.

By default, Ganglia assumes that there is a single memcache sFlow instance per host. If hosts contain more than one memcache instance, setting `multiple_memcache_instances=yes` in the gmond configuration file causes gmond to prefix each metric and description with the TCP port number that the instance uses to receive memcache requests; for example, the `mc_curr_conns` metric becomes `<port>.mc_curr_conns` with description <port>: memcache Current Connections. See "Configuring gmond to Receive sFlow" on page 155.

Table 8-6. sFlow memcache metrics

Metric Name	Description	Type
mc_curr_conns*	memcache Current Connections	memcache
mc_total_conns	memcache Total Connections	memcache
mc_conn_structs	memcache Connection Structs	memcache
mc_cmd_get*	memcache Command GET	memcache
mc_cmd_set*	memcache Command SET	memcache
mc_cmd_flush	memcache Command FLUSH	memcache
mc_get_hits*	memcache GET Hits	memcache
mc_get_misses*	memcache GET Misses	memcache
mc_delete_misses*	memcache DELETE Misses	memcache
mc_delete_hits*	memcache DELETE Hits	memcache
mc_incr_misses	memcache INCR Misses	memcache
mc_incr_hits	memcache INCR Hits	memcache
mc_decr_misses	memcache DECR Misses	memcache
mc_decr_hits	memcache DECR Hits	memcache
mc_cas_misses	memcache CAS Misses	memcache

Metric Name	Description	Type
mc_cas_hits	memcache CAS Hits	memcache
mc_cas_badval	memcache CAS Badval	memcache
mc_auth_cmds	memcache AUTH Cmds	memcache
mc_auth_errors	memcache AUTH Errors	memcache
mc_bytes_read*	memcache Bytes Read	memcache
mc_bytes_written*	memcache Bytes Written	memcache
mc_limit_maxbytes*	memcache Limit MaxBytes	memcache
mc_accepting_conns	memcache Accepting Connections	memcache
mc_listen_disabled_num	memcache Listen Disabled	memcache
mc_threads	memcache Threads	memcache
mc_conn_yields	memcache Connection Yields	memcache
mc_bytes*	memcache Bytes	memcache
mc_curr_items*	memcache Current Items	memcache
mc_total_items	memcache Total Items	memcache
mc_evictions*	memcache Evictions	memcache
mc_cmd_touch	memcache Command TOUCH	memcache
mc_rejected_conns	memcache Rejected Connections	memcache
mc_reclaimed	memcache Reclaimed	memcache

In addition to the memcache metrics, a memcache sFlow agent also exports memcache operation records for randomly sampled operations. See Table 8-7.

Table 8-7. sFlow memcache operation attributes

Attribute Name	Description
mc_protocol	memcache protocol (i.e., ASCII, BINARY, etc.)
mc_cmd	memcache command (i.e., SET, GET, INCR, etc.)
mc_key	Key used to store/retrieve data
mc_nkeys	Number of keys in request
mc_value_bytes	Size of the object referred to by key (in bytes)
mc_uS	Duration of the operation (in microseconds)
mc_status	Status of command (i.e., OK, ERROR, STORED, NOT_STORED, etc.)
socket_protocol	IP protocol type (e.g., TCP, UDP, etc.)
socket_local_ip	IP address of memcache server
socket_remote_ip	IP address of memcache client
socket_local_port	Server TCP/UDP port number
socket_remote_port	Client TCP/UDP port number

Configuring gmond to Receive sFlow

The bulk of a typical gmond configuration file, *gmond.conf*, is devoted to the metrics that gmond exports for the local host. When gmond is configured as a pure sFlow collector, most configuration settings can be eliminated, resulting in a simple configuration file:

```
/* Configuration settings for a pure sFlow receiver */
/* Delete all udp_send_channel, modules, collection_group and include sections */

globals {
  daemonize = yes
  setuid = yes
  user = nobody
  debug_level = 0
  max_udp_msg_len = 1472
  mute = yes       /* don't send metrics */
  deaf = no        /* listen for metrics */
  allow_extra_data = yes
  host_dmax = 0
  host_tmax = 20
  cleanup_threshold = 300
  gexec = no
  send_metadata_interval = 0
}

cluster {
  name = "unspecified"
  owner = "unspecified"
  latlong = "unspecified"
  url = "unspecified"
}

host {
  location = "unspecified"
}

/* channel to receive gmetric messages */
/* eliminate mcast_join - sFlow is a unicast protocol */
udp_recv_channel {
  port = 8649
}

/* channel to service requests for XML data from gmetad */
tcp_accept_channel {
  port = 8649
}

/* channel to receive sFlow */
/* 6343 is the default sFlow port, an explicit sFlow    */
/* configuration section is needed to override default  */
udp_recv_channel {
  port = 6343
}
```

```
/* Optional sFlow settings */
#sflow {
# udp_port = 6343
# accept_vm_metrics = yes
# accept_jvm_metrics = yes
# multiple_jvm_instances = no
# accept_http_metrics = yes
# multiple_http_instances = no
# accept_memcache_metrics = yes
# multiple_memcache_instances = no
#}

/* end of configuration file */
/* Delete all modules, collection_group and include sections */
```

The deaf and mute global settings instruct gmond to listen for metrics but not send them. All the settings related to local generation of metrics have been removed. For consistency, an sFlow agent should be installed on the host running gmond if local metrics are required.

In the Ganglia architecture, each cluster is monitored by a separate gmond process. If more than one cluster is to be monitored, then it is possible to run multiple gmond processes on a single server, each with its own configuration file. For example, if sFlow agents on the first cluster are sending to port 6343, then sFlow agents on the second cluster should be configured to send to a different port, say 6344. The following settings, in a separate configuration file, configure the second gmond instance to listen on the nonstandard port:

```
...

/* channel to receive gmetric messages */
udp_recv_channel {
  port = 8650
}

/* channel to service requests for XML data from gmetad */
tcp_accept_channel {
  port = 8650
}

/* channel to receive sFlow */
udp_recv_channel {
  port = 6344
}

/* Change sFlow channel to non-standard port 6344 */
sflow {
  udp_port = 6344
}
```

The nonstandard port setting is only required if both gmond processes are running on a single server. If each cluster is monitored by a separate server, then the sFlow agents on each cluster need to be configured to send metrics to the collector for their cluster.

Another alternative is to assign multiple IP addresses to a single server, one per cluster that is to be monitored. In this case, multiple gmond instances are created, each associated with a distinct IP address, and the sFlow agents in each cluster are configured to send metrics to the associated IP address. For example, the following settings configures the gmond instance to listen for sFlow on a specific IP address:

```
/* channel to receive sFlow */
udp_recv_channel {
  port = 6343
  bind = <IP address>
}
```

For more information on configuring Ganglia clusters, see Chapter 2.

Host sFlow Agent

The Host sFlow agent is an open source implementation of the sFlow standard for server monitoring. The Host sFlow agent provides "scalable, multi-vendor, multi-OS performance monitoring with minimal impact on the systems being monitored.

The following example shows how to install and configure the Host sFlow daemon (hsflowd) on a Linux server in order to illustrate how to send sFlow metrics to Ganglia gmond. The Host sFlow website should be consulted for detailed instructions on installing and configuring Host sFlow on other platforms.

First, install the hsflowd software:

```
rpm -Uvh hsflowd_XXX.rpm
```

Alternatively, hsflowd can be installed from sources:

```
tar -xzf hsflowd-X.XX.tar.gz
cd hsflowd-X.XX
make
make install
make schedule
```

Next, edit the hsflowd configuration file, */etc/hsflowd.conf*:

```
sflow{
  DNSSD = off
  polling = 20
  sampling = 512
  sampling.http = 100
  sampling.memcache = 400
  collector{
   ip = <gmond IP address>
```

```
    udpport = <gmond udp_rcv_channel port>
  }
}
```

By default, hsflowd is configured to use DNS Service Discovery (DNS-SD) to automatically retrieve settings (i.e., `DNSSD = on`). Manual configuration (i.e., `DNSSD = off`) is recommended when using hsflowd to send metrics to gmond because it allows each host to be configured to send sFlow to the gmond instance responsible for its cluster. If only one cluster is being monitored, consider using DNS-SD.

Finally, start the hsflowd daemon:

```
service hsflowd start
```

Within a few minutes, metrics for the server should start appearing in Ganglia. If metrics fail to appear, follow the directions in "Troubleshooting" on page 161.

Host sFlow Subagents

The Host sFlow website maintains a list of related projects implementing subagents that extend sFlow monitoring to HTTP, memcache, and Java applications running on the server. These subagents require minimal additional configuration because they share configuration settings with hsflowd (Figure 8-2).

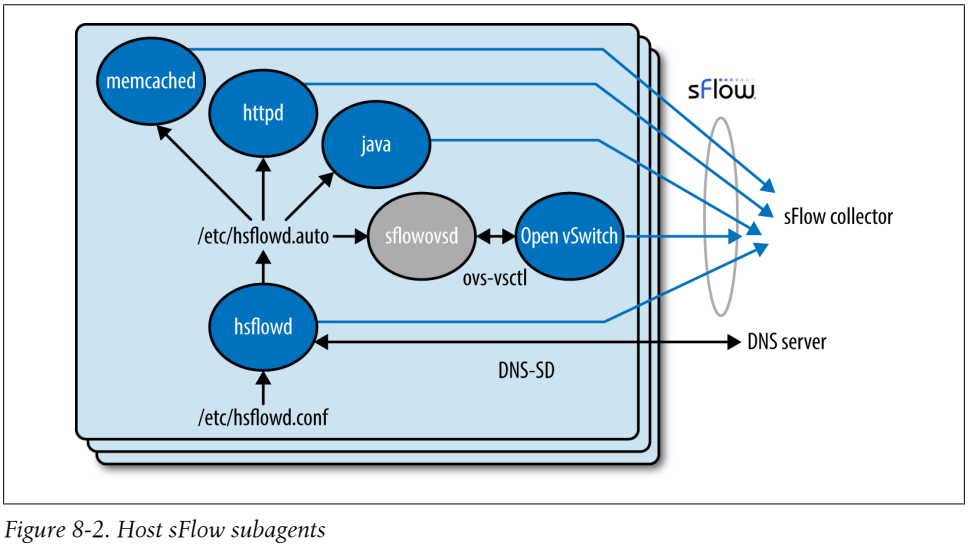


Figure 8-2. Host sFlow subagents

The hsflowd daemon writes configuration information that it receives via DNS-SD or through its configuration file to the */etc/hsflowd.auto* file. Other sFlow subagents

running on the host automatically detect changes to the */etc/hsflowd.auto* file and apply the configuration settings.

The sFlow protocol allows each subagent to operate autonomously and send an independent stream of metrics to the sFlow collector. Distributing monitoring among the agents eliminates dependencies and synchronization challenges that would increase the complexity of the agents.

Extending the functionality of Host sFlow using subagents differs from gmond's use of modules (see the section "Extending gmond with Modules" on page 78), but is very similar in approach to adding custom metrics using gmetric (see the section "Extending gmond with gmetric" on page 97). In fact, gmetric can easily be used to add custom metrics to the standard metrics exported using sFlow (see "Custom Metrics Using gmetric" on page 160).

Each of the sFlow subagents is responsible for exporting a different set of metrics. At the time of writing, the following subagents are available:

hsflowd
: The Host sFlow daemon exports standard host metrics (see "Server Metrics" on page 147), and can also export per-virtual-machine statistics (see "Hypervisor Metrics" on page 149) when run on a Hyper-V, Xen, XenServer, or KVM hypervisor. In addition, network traffic monitoring using iptables/ULOG is supported on Linux platforms.

Open vSwitch
: The Open vSwitch is the default switch in XenServer 6.0, the Xen Cloud Platform, and also supports Xen, KVM, Proxmox VE, and VirtualBox. It has also been integrated into many virtual management systems including OpenStack, openQRM, and OpenNebula. Enabling the built-in sFlow monitoring on the virtual switch offers the same visibility as sFlow on physical switches, providing a unified, end-to-end view of network performance across the physical and virtual infrastructure. The sflowovsd daemon ships with Host sFlow and automatically configures sFlow monitoring on the Open vSwitch using the `ovs-vsctl` command. Similar integrated sFlow support is available for the Microsoft extensible virtual switch that is part of the upcoming Windows Server 2012 version of Hyper-V.

jmx-sflow-agent
: The Java sFlow agent is invoked on the Java command line, easily adding monitoring to existing Java applications and exporting standard metrics (see "Java Virtual Machine Metrics" on page 150).

tomcat-sflow-valve
: This is a Tomcat Valve that can be loaded in an Apache Tomcat web server to export Java metrics (see "Java Virtual Machine Metrics" on page 150) and HTTP metrics (see "HTTP Metrics" on page 151).

mod-sflow
: This is a module for the Apache web server that exports HTTP metrics (see "HTTP Metrics" on page 151).

nginx-sflow-module
: This is a module for the NGINX web server that exports HTTP metrics (see "HTTP Metrics" on page 151).

sflow/memcached
: This project tracks the latest memcache release and includes an embedded sFlow subagent that exports memcache metrics (see "memcache Metrics" on page 153).

Custom Metrics Using gmetric

One of the strengths of Ganglia is the ability to easily add new metrics. Although the Host sFlow agent doesn't support the addition of custom metrics, the Ganglia gmetric command-line tool provides a simple way to add custom metrics (see "Extending gmond with gmetric" on page 97).

For example, the following command exports the number of users currently logged into a system:

```
gmetric -S 10.0.0.1:server1 -n Current_Users -v `who |wc -l` -t int32 -u current_users
```

Using the `-S` or `--spoof` option ensures that the receiving gmond instance correctly associates metrics sent using gmetric with metrics sent by hsflowd. The spoof argument is a colon-separated string containing an IP address and a hostname. The Host sFlow daemon writes information about its configuration and sFlow settings into the */etc/hsflowd.auto* file, including the IP address and hostname used when it sends sFlow data; the gmetric spoof string must match these values.

The gmetric command-line tool is distributed with gmond and picks up settings from the *gmond.conf* file. Because there are no gmond instances running on the hosts in an sFlow deployment, eliminating the dependency on *gmond.conf*, gmond, and gmetric is worthwhile. The *gmetric.py* utility is a simple Python script that can be used a replacement for gmetric.

The following bash script demonstrates how configuration settings can be extracted from the *hsflowd.auto* file and used as arguments for the *gmetric.py* command:

```
#!/bin/bash
# Read configuration settings from hsflowd.auto
while IFS="=" read name value
do
  case "$name" in
    agentIP)
       SPOOF_IP=$value
       ;;
    hostname)
       SPOOF_HOSTNAME=$value
       ;;
```

```
    collector)
       set $value
       HOST=$1
       PORT=$(($2-6343+8649))
       ;;
    *)
       ;;
  esac
done < /etc/hsflowd.auto

# Export one or more custom metrics using gmetric.py
/usr/local/bin/gmetric.py\
 --host $HOST\
 --port $PORT\
 --spoof $SPOOF_IP:$SPOOF_HOSTNAME\
 --name Current_Users\
 --value `/usr/bin/who |/usr/bin/wc -l`\
 --type int32\
 --units current_users
```

This script assumes that *gmetric.py* has been installed as an executable in */usr/local/bin*. The calculation for the gmetric port is based on the numbering convention used in "Configuring gmond to Receive sFlow" on page 155, where the standard sFlow and gmetric ports are shifted by the same constant when creating multiple gmond instances.

Additional custom metrics can be added to the end of the script; ganglia/gmetric is a library of user-contributed gmetric scripts maintained by the Ganglia project on github. Scheduling the script to run every minute using cron allows Ganglia to automatically track the custom metrics.

Troubleshooting

Most problems with sFlow deployments occur because the sFlow datagrams are dropped somewhere between the sFlow agent and gmond. The following steps will help identify where measurements are being dropped.

Are the Measurements Arriving at gmond?

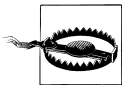
Use a packet capture tool, such as `tcpdump`, to verify that the sFlow packets are arriving at the server running gmond. The following command uses `tcpdump` to check for packets arriving on UDP port 6343:

```
tcpdump -p udp port 6343
```

Check every `udp_recv_channel` specified in *gmond.conf* files in order to verify that metrics are arriving (see "Configuring gmond to Receive sFlow" on page 155).

If the missing metrics are associated with a single host, use `tcpdump` to filter on the specific host. The following command verifies that sFlow data is arriving from host 10.0.0.237:

```
tcpdump -p src host 10.0.0.237 and udp port 6343
```



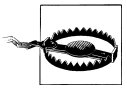
Packet capture using tcpdump occurs before server firewall rules are applied. Don't assume that the fact that packets are being displayed by tcpdump means that the packets are being received by gmond—packets can still be dropped by the firewall. Make sure that incoming sFlow packets are permitted by the local firewall iptables on a Linux system. Typically, UDP port 6343, the default sFlow port, is required—but you will need to ensure that every `udp_recv_channel` and `tcp_accept_channel` configured in the *gmond.conf* file is allowed as an incoming connection in the firewall rules.

A quick way to check whether the firewall is the problem is to temporarily disable the firewall and see if metrics start to appear in Ganglia. However, this procedure should be performed only if the server is in a secure environment where the security risk of turning off the firewall is acceptable.

Next, use `telnet` to connect to the gmond `tcp_accept_channel` and verify that the metrics appear in the XML document. The following command assumes that the default `tcp_accept_channel` setting, 8649, is being used:

```
telnet localhost 8649
```

If metrics are missing from the XML output, the next step is to verify that the metrics are arriving in the sFlow datagrams. The `sflowtool` command is similar to `tcpdump`, decoding and printing the contents of network packets. However, whereas `tcpdump` is a general purpose packet analyzer, `sflowtool` is specifically concerned with sFlow data, performing a full decode of all the sFlow metrics and attributes.



If you are monitoring Windows hosts and the charts aren't appearing in the Ganglia user interface but the data appears to be correct in the XML output, make sure that `case_sensitive_hostnames` is set to `0` in *gmetad.conf*.

The following command demonstrates how `sflowtool` is used in combination with `tcpdump` in order to print out the contents of the sFlow datagrams and verify that specific metrics are being received:

```
tcpdump -p -s 0 -w - udp port 6343 | sflowtool
```

You can use `sflowtool` on its own to receive and decode sFlow. However, when gmond is running, it will have opened the port to listen for sFlow, blocking `sflowtool` from being able to open the port. Using `tcpdump` allows the packets to be captured from the open port and fed to `sflowtool`. The alternative is to stop gmond before running this test.

If you are receiving sFlow data, then `sflowtool` will print out the detailed contents, and you should see output similar to the following:

```
startDatagram =================================
datagramSourceIP 10.1.4.2
datagramSize 432
unixSecondsUTC 1339651142
datagramVersion 5
agentSubId 100000
agent 10.1.4.2
packetSequenceNo 464
sysUpTime 9244000
samplesInPacket 1
startSample ----------------------
sampleType_tag 0:2
sampleType COUNTERSSAMPLE
sampleSequenceNo 464
sourceId 2:1
counterBlock_tag 0:2001
adaptor_0_ifIndex 1
adaptor_0_MACs 1
adaptor_0_MAC_0 000000000000
adaptor_1_ifIndex 2
adaptor_1_MACs 1
adaptor_1_MAC_0 eedb257595e5
counterBlock_tag 0:2005
disk_total 64427231232
disk_free 51740361728
disk_partition_max_used 35.43
disk_reads 20349
disk_bytes_read 403682304
disk_read_time 181676
disk_writes 16994
disk_bytes_written 144289792
disk_write_time 1130328
counterBlock_tag 0:2004
mem_total 1073741824
mem_free 723337216
mem_shared 0
mem_buffers 17854464
mem_cached 192057344
```

```
swap_total 2181029888
swap_free 2181029888
page_in 201324
page_out 70454
swap_in 0
swap_out 0
counterBlock_tag 0:2003
cpu_load_one 0.000
cpu_load_five 0.000
cpu_load_fifteen 0.000
cpu_proc_run 1
cpu_proc_total 96
cpu_num 1
cpu_speed 3200
cpu_uptime 9298
cpu_user 5590
cpu_nice 5570
cpu_system 16120
cpu_idle 9204610
cpu_wio 66020
cpuintr 0
cpu_sintr 0
cpuinterrupts 231288
cpu_contexts 370635
counterBlock_tag 0:2006
nio_bytes_in 2514529
nio_pkts_in 7316
nio_errs_in 0
nio_drops_in 0
nio_bytes_out 1799289
nio_pkts_out 8199
nio_errs_out 0
nio_drops_out 0
counterBlock_tag 0:2000
hostname virtual-vm
UUID 7c270fa3830347a9b6aef60bac8cd16f
machine_type 3
os_name 2
os_release 2.6.18-308.1.1.el5xen
endSample   ----------------------
endDatagram   =================================
```

The output of `sflowtool` consists of simple key/value pairs that are easily processed using scripting languages such as Perl. An example is given in "Using Ganglia with Other sFlow Tools" on page 165.

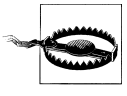
At this point, if the sFlow is being received and metrics are missing, it is likely that the sFlow agent has been incorrectly configured and the metrics aren't being sent.

Are the Measurements Being Sent?

If possible, use sflowtool and tcpdump on the sending machine to verify that measurements are being transmitted. Again, it is possible that local firewall rules on the sending machine are preventing transmission of the sFlow datagrams. See "Are the Measurements Arriving at gmond?" on page 161.

Make sure to verify that the destination IP and port correspond to the *gmond.conf* file settings at the receiving end.

Check the configuration of the sFlow agent. Verify that the sFlow agent's counter polling interval is configured and set to a reasonable value—a polling interval of 30 seconds is typical for sFlow deployments.



There are two types of sFlow data: periodic counters and randomly sampled packets/transactions. Ganglia gmond is able to process only counter data, so it is possible that sFlow sample records are being transmitted to gmond, but without counter records, Ganglia will not show any data.

If the measurements are being sent but not received, work with your network administrator to identify any firewall or ACL setting in intermediate switches, routers, or firewall that may be dropping the sFlow datagrams.

Using Ganglia with Other sFlow Tools

Ganglia is reporting an increase in HTTP traffic to your web servers—how do you know whether you are the target of a denial of service attack, or a marketing promotion has just gone viral? You are seeing an increase in cache misses to your memcached cluster—how can you fix the problem? If you are using sFlow agents to generate the metrics, you have the data to answer these types of question at your fingertips.

The metrics that Ganglia displays are only part of the information contained in an sFlow stream. For example, the `mod_sflow` agent running in Apache servers also randomly samples HTTP operations, sending records that include attributes such as URL, user-agent, response time, client socket, and bytes transferred; see Table 8-5. Accessing the sampled HTTP operations allows you to dig deeper into a trend and identify the source of the increased traffic.

The following sflowtool output shows the HTTP counters and operation samples reported by `mod_sflow`:

```
startDatagram =================================
datagramSourceIP 10.0.0.150
datagramSize 132
unixSecondsUTC 1339652714
datagramVersion 5
agentSubId 0
```

```
agent 10.0.0.150
packetSequenceNo 30526
sysUpTime 3981481944
samplesInPacket 1
startSample ----------------------
sampleType_tag 0:2
sampleType COUNTERSSAMPLE
sampleSequenceNo 19510
sourceId 3:80
counterBlock_tag 0:2002
parent_dsClass 2
parent_dsIndex 1
counterBlock_tag 0:2201
http_method_option_count 0
http_method_get_count 55755
http_method_head_count 4
http_method_post_count 1359
http_method_put_count 0
http_method_delete_count 0
http_method_trace_count 0
http_methd_connect_count 7
http_method_other_count 0
http_status_1XX_count 0
http_status_2XX_count 54577
http_status_3XX_count 2211
http_status_4XX_count 314
http_status_5XX_count 23
http_status_other_count 0
endSample   ----------------------
endDatagram   =================================
startDatagram =================================
datagramSourceIP 10.0.0.150
datagramSize 264
unixSecondsUTC 1339652714
datagramVersion 5
agentSubId 0
agent 10.0.0.150
packetSequenceNo 30527
sysUpTime 3981483944
samplesInPacket 1
startSample ----------------------
sampleType_tag 0:1
sampleType FLOWSAMPLE
sampleSequenceNo 28550
sourceId 3:80
meanSkipCount 2
samplePool 57126
dropEvents 2
inputPort 0
outputPort 1073741823
flowBlock_tag 0:2100
extendedType socket4
socket4_ip_protocol 6
socket4_local_ip 10.0.0.150
socket4_remote_ip 10.0.0.70
```

```
socket4_local_port 80
socket4_remote_port 63729
flowBlock_tag 0:2206
flowSampleType http
http_method 2
http_protocol 1001
http_uri /index.php
http_host 10.0.0.150
http_useragent curl/7.21.4
http_request_bytes 0
http_bytes 0
http_duration_uS 1329
http_status 403
endSample   ----------------------
endDatagram   ================================
```

The following example demonstrates how the sflowtool output can be used to generate additional metrics. The Perl script uses sflowtool to decode the HTTP request data and calculates average response time over a minute, printing out the results:

```
#!/usr/bin/perl -w
use strict;
use POSIX;

my $total_time = 0;
my $total_requests = 0;
my $now = time();
my $start = $now - ($now % 60);

open(PS, "/usr/local/bin/sflowtool|") || die "Failed: $!\n";
while( <PS> ) {

  my ($attr,$value) = split;

  # process attribute
  if('startDatagram' eq $attr) {
    $now = time();
    if($now - $start >= 60) {
      if($total_requests > 0) {
        printf "%d %d\n", $start, int($total_time/$total_requests);
      }
      $total_time = 0;
      $total_requests = 0;
      $start = $now - ($now % 60);
    }
  }
  if('http_duration_uS' eq $attr) {
    $total_time += $value;
    $total_requests++;
  }
}
```

The output of the script follows. The first column is the time (in seconds since the epoch) and the second column is the average HTTP response time (in microseconds):

```
[pp@pcentos ~]$ ./http_response_time.pl
1339653360 2912326
1339653420 3002692
1339653480 1358454
1339653540 3983638
```

This example demonstrated how sampled transactions can be used to generate new metrics. The metrics can be fed back to Ganglia using gmetric; see "Custom Metrics Using gmetric" on page 160.

This is just one simple example. There are many metrics that can be calculated based on the detailed information in transaction samples. Using sflowtool to pretty-print sFlow records is a good way to see the information that is available and experiment with calculating different metrics.

You don't have to write your own analysis scripts. The sFlow data can be converted into different forms for compatibility with existing tools. For example, the following command uses sflowtool to convert the binary sFlow HTTP operation data into ASCII CLF so that the operations can be visually inspected or exported to a web log analyzer such as Webalizer:

```
sflowtool -H | rotatelogs log/http_log
```

You can also use protocol reporting tools such as Wireshark with sFlow, using sflowtool to convert sFlow into the standard PCAP format:

```
wireshark -k -i <(sflowtool -t)
```

In addition, there are a broad range of native sFlow analysis options that provide complementary functionality to Ganglia's. The sFlow.org website contains a list of open source and commercial sFlow analysis tools—including Ganglia, of course!

When using multiple sFlow analysis tools, each tool needs to receive copies of the sFlow packets. There are two main approaches to replicating sFlow:

Source replication
: Configure each sFlow agent to send sFlow packets to multiple destinations. Because sFlow is a unicast protocol, this involves resending packets to each of the destinations and this increases measurement traffic on the network. The additional traffic is generally not an issue, as each sFlow agent generates a small number of packets.

Destination replication
: Some sFlow analyzers have built-in replication and packet forwarding capabilities, and there are commercial and open source UDP replication tools available, including `sflowtool`.

Finally, the case study "Tagged, Inc." on page 172 demonstrates how Ganglia and sFlow are used in a large deployment, illustrating the techniques described in this chapter and providing examples that demonstrate the importance of performance monitoring to the operations and developer teams.

CHAPTER 9
Ganglia Case Studies

Daniel Pocock, Bernard Li, Alex Dean, and Peter Phaal

The Ganglia project started out in 1999 with the aim of monitoring grid computing infrastructure: largely homogeneous clusters of similar compute nodes, typically in the academic and research community. The project founders (including Matt Massie) designed the system to be lightweight and efficient.

In this chapter, we present a range of case studies that demonstrate just how widely respected Ganglia has become—not just within the original audience, but in the wider world of industry.

The SARA case study is just one example of Ganglia at home in the environment it was designed for, albeit on the other side of the Atlantic.

The fact that Ganglia is being used to monitor 24 x 7 e-commerce enterprises is a sign of just how robust it is. Some of these include Etsy and Quantcast, both of which have shared an insight into just how Ganglia keeps their business running. The social networking trend is one of the most widely talked about revolutions in communications today, and it is no surprise to find Ganglia has had its finger in that pie, too, as it is the tool of choice at Tagged.

Stepping back from extremes of multicore CPU deployments, Ganglia has also proven itself to be truly adaptable and versatile in the face of dramatic change. It is estimated that more people will be accessing the Web from smartphones than from desktop PCs by the time you are reading this book. In this new world, CPUs spend more time sleeping than the average housecat; network connectivity stops and starts and nodes rarely hold the same IP address for more than an hour. Ganglia's lightweight protocol, which functions reliably as unidirectional UDP traffic, has proven itself to be ready for business at this level, too, as demonstrated by its presence as an embedded agent in the Lumicall app for Android. It is this versatility that may well be the most significant acknowledgment of how perceptive Ganglia's founders were when designing an efficient monitoring protocol.

Some of the most interesting case studies may be those that we can't publish at all. It is known that due to Ganglia's bare-bones efficiency, in that it was written in low-level C with the source code available for all to see, it was chosen for a number of algorithmic trading systems, where every process on a server is closely scrutinized to maintain a competitive edge.

Tagged, Inc.

Tagged.com is a social networking site with over 350 million registered users and 5 billion page views a month. This case study describes how Ganglia and sFlow (Chapter 8) are used to monitor the performance of the Apache, memcached, and Java services that make up the site (thanks to Dave Mangot and Tagged for providing the information for this case study).

Site Architecture

Tagged's scale-out, tiered site architecture, shown in Figure 9-1, is typical of social networking websites.

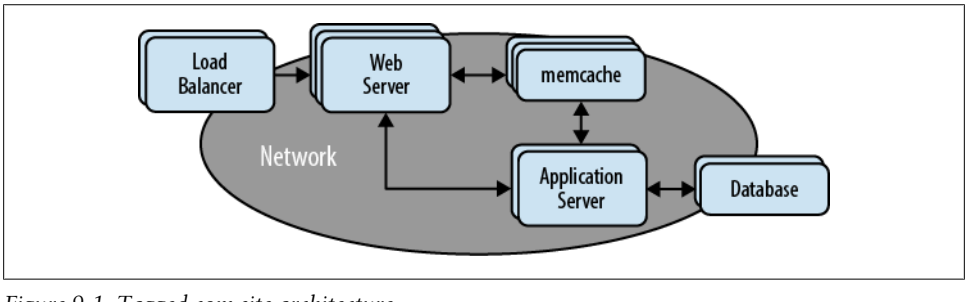


Figure 9-1. Tagged.com site architecture

Incoming web requests are handled by load balancers that distribute the requests to clusters of web servers. Application logic is implemented through a combination of scripted pages in the web tier interacting with clusters that make up the application tier. The application tier interacts with the database tier to maintain persistent user data. The scale-out design of each tier ensures that the site can handle user growth by adding additional servers to the clusters.

The memcache tier is a critical component in most social networking sites. It performs two major functions:

1. Caching information retrieved from the database tier in order to reduce database load
2. Providing the scatter/gather functionality needed to rapidly query each user's friends and their current status

Tagged has over 7 terabytes of memory in their memcache pool, and the effect of the cache on site performance is dramatically illustrated in "Optimizing memcached efficiency" on page 175.

Ganglia is well suited to monitoring Tagged's infrastructure, presenting a comprehensive, near real-time view of the performance of the clusters making up the site. Tagged had been using Ganglia for quite some time but we're starting to see network performance issues with gmond as they scaled up the site. Solving these performance issues was an important motivation in deciding to migrate from gmond to sFlow agents.

One of the great things about replacing the gmond processes is the increased efficiency of the monitoring infrastructure. With gmond, every metric sends a packet across the wire. If you sample every 15 seconds, gmond sends a packet every 15 seconds for each metric that you monitor. With sFlow agents, you can sample every 15 seconds, but the agent will batch those metrics into a single packet to send across the wire. Tagged is now able to collect more metrics, more often, with fewer packets. On a big network like Tagged, anything that can lower packets per second is a big win. The difficult part was converting from multicast to unicast. The switch to sFlow was used as an opportunity to templatize all the Puppet CMDB configurations for this purpose. Now Tagged has a system that they really love.

Monitoring Configuration

The diagram in Figure 9-2 shows the elements making up Tagged's monitoring system. The diagram has been simplified—in reality there are multiple clusters within each of the service tiers.

Host sFlow agents, described in "Host sFlow Agent" on page 157, are installed on each server in the Web, memcached, and Application clusters. The Host sFlow agents report the standard server metrics as described in "Server Metrics" on page 147 to gmond instances running on two collection servers, gmond01 and gmond02. Each gmond server has multiple gmond instances, each listening on a different UDP port for measurements from each cluster; see "Configuring gmond to Receive sFlow" on page 155.

Custom metrics are generated using scripts and sent using the gmetric command to the gmond instance responsible for the cluster; see the section "Custom Metrics Using gmetric" on page 160. In addition, sFlow analysis software running on sflow01 processes sampled transaction data, generating additional metrics that are exported to Ganglia using gmetric; see "Using Ganglia with Other sFlow Tools" on page 165.

Puppet is used extensively by Tagged for managing server configurations. In this case, *gmond.conf*, *hsflowd.conf*, and *gmetad.conf* files are all generated using Puppet ERB templates. Coordination between the *hsflowd.conf* and *gmond.conf* settings is needed to ensure that sFlow and gmetric messages are sent to the correct gmond instance. The *gmetad.conf* files should also be coordinated with the *gmond.conf* files on gmond01 and

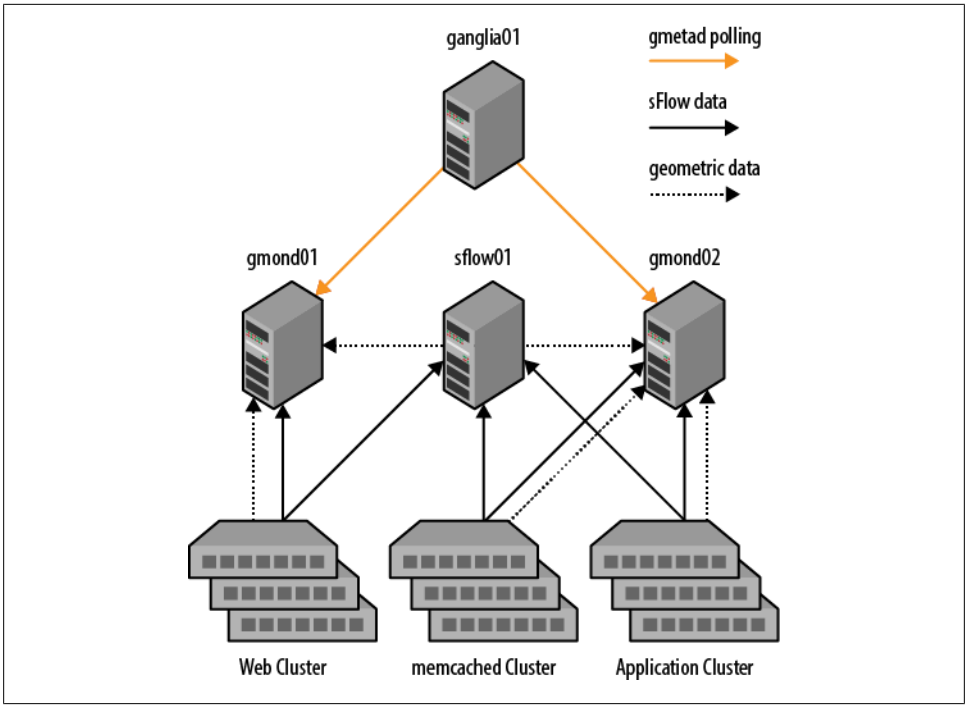


Figure 9-2. Monitoring system

gmond02 to ensure that clusters are correctly polled and labelled, which is achieved through a Puppet ERB template in conjunction with a custom Puppet function.

Apache

Apache stats would normally be calculated based on the output of `mod_status`, typically using a script to make HTTP requests to the Apache */server-status/* page and parse the results. Using `mod_sflow` instead, HTTP metrics (see "HTTP Metrics" on page 151) are streamed directly into Ganglia. No additional configuration is needed because `mod_sflow` reads configuration settings from the Host sFlow agent (see "Host sFlow Subagents" on page 158).

memcached

memcached has been a black box for systems administrators and developers for many years. There are a limited number of statistics you can get using the *STATS* interface to memcached, and those statistics are about the memcached instance as a whole, not about individual keys. Some of the `STATS` commands (like `SIZES`) will actually lock up the entire cache while it scans the items, leaving your memcached instance unusable for several seconds. memcached/sFlow integrates sFlow instrumentation within the memcached instances providing the ability to see details that previously could be ob-

tained only by loading a kernel module like Gear6 Advanced Reporter. memcached/sFlow is a simple open source patch to the memcached source (hopefully the patch will be pulled into the memcached base soon), with no kernel recompilation required.

memcached/sFlow provides the regular metrics (see the section "memcache Metrics" on page 153), such as hit rates that you would get from the `STATS` command, without having to telnet to the memcached instance every 15 seconds.

memcached/sFlow reads configuration settings from the Host sFlow agent, so no additional configuration is needed (see "Host sFlow Subagents" on page 158).

Java

Before moving to sFlow, Tagged used to get Java virtual machine metrics using Zenoss, which uses a dedicated JMX poller to retrieve information from a designated list of hosts. This approach unfortunately doesn't scale to any large environment. Consider Netflix in Amazon EC2, which has potentially thousands of machines disappearing and appearing on the network in minutes but can refresh the host list on the poller only so often. The alternative of using a gmond module (see "Extending gmond with Modules" on page 78) would require starting a JVM every time it needs to check a metric, because a Java client is needed to retrieve metrics from a JVM using the JMX protocol.

With sFlow instrumentation of the JVM, data is pushed from the JVMs to gmond, with no polling necessary. The jmx-flow-agent runs as a `-javaagent` argument to the Java command line. Each JVM automatically sends its metrics (see "Java Virtual Machine Metrics" on page 150) to gmond the moment the JVM starts.

Again, the jmx-sflow-agent reads configuration settings from the Host sFlow agent, so no additional configuration is needed (see "Host sFlow Subagents" on page 158).

Examples

The following examples highlight some of the areas where Ganglia and sFlow provide visibility into critical site services to the operations and developer teams at Tagged.

Optimizing memcached efficiency

Using Ganglia to monitor the cold start of a memcached cluster provides a compelling demonstration of the leverage that a memcached cluster provides. The two charts shown in Figure 9-3 show overall bytes in and out of a cluster as it starts up. In the figure, the chart on the left shows initial traffic to the cold cache and the chart on the right shows the same cluster minutes later once it has warmed up. Comparing the two charts gives a clear illustration of the importance of the memcached clusters in scaling the site. When the cache is cold, performance is limited by the database tier, data is being written into the cache and the read performance delivers around 10 MB/s. Once the cache has warmed up, read performance jumps to 1.7 GB/s—a 170x improvement in throughput!

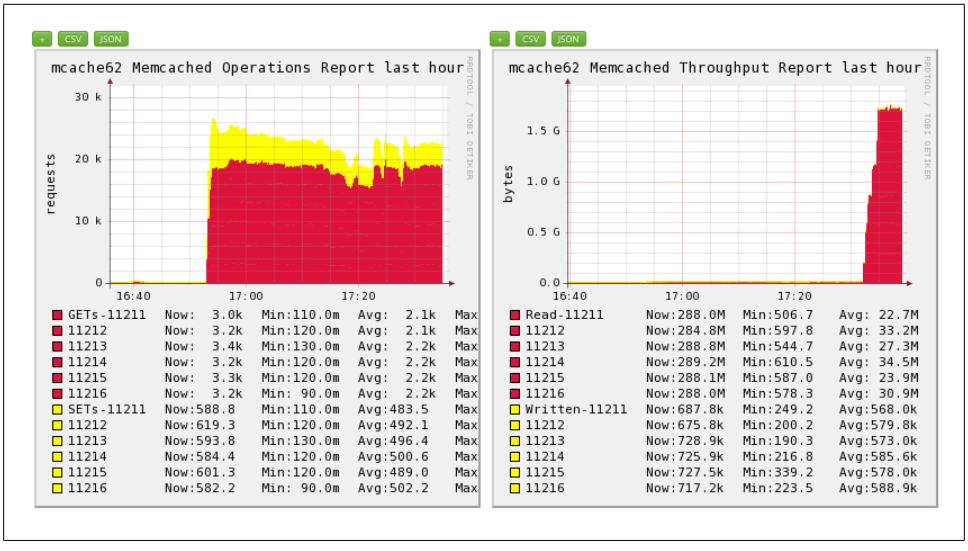


Figure 9-3. Cold start of memcached cluster

The jump in throughput is so dramatic that the rescaling of the vertical axis on the right-hand chart makes the values shown on the left-hand chart appear as a thin line with a near zero value.

The performance of the entire site is critically dependent on how effectively the memcache clusters protect the database tier. Because the primary bottleneck limiting performance is the database tier, any improvement in the cache hit rates reduces load on the databases and improves scalability. For example, suppose that we can improve the cache hit rate from 95 to 96 percent. This might not seem like much, but when you consider that cache misses represent database queries, this seemingly trivial 1 percent improvement in hit rate is actually a 20 percent reduction in the miss rate (going from a 5 percent miss rate to a 4 percent miss rate), resulting in a proportional decrease in the load on the databases and a significant increase in overall site capacity and performance.

The Ganglia chart shown in Figure 9-4 trends the overall efficiency of the memcached cluster and is one of the critical performance metrics tracked for the site.

Although Ganglia does an excellent job of trending the performance counters exported by sFlow, one of the benefits of using sFlow as the measurement technology is that counters aren't the only information being exported. In addition to periodically exporting the standard memcache counters to Ganglia, the sFlow agents embedded in the memcached servers also randomly sample memcache operations.

For example, the table in Figure 9-5 is updated every minute to show the keys associated with the most cache misses. Tracking misses helps identify problems that can have a

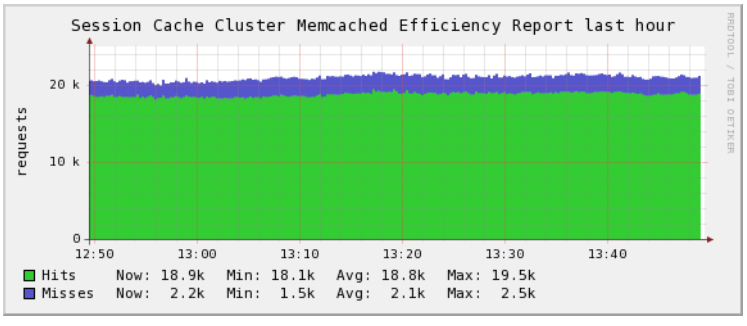


Figure 9-4. Session cache cluster efficiency

Memcache Key	Operations
tag	3.57K
WORKING	820.00
cacheable	760.00
0apiCache	760.00
0teamconnection	273.33
buddylist	246.67
0rateLimit	146.67
0registration	133.33
simpleuserdata	113.33
0cid	73.33
0alerts	60.00
_	53.33
0team	46.67
0photo	40.00
0birthdays	33.33

Figure 9-5. Missed keys

significant impact on performance, such as a mistyped key name in a PHP script. The data also identifies "hot keys" that, if deleted, could result in a stampede of requests to the database—a cache miss storm.

Web load

Measurements from the web tier provide an overview of the entire site, tracking response times and request rates to each of the applications running on the site.

The charts in Figure 9-6 show combined CPU, network, HTTP request type, and HTTP status code metrics from a static web cluster. The CPU load is low, mostly waiting for the disk, as you would expect when serving static content. Also notice that all the requests are HTTP `GET` operations.

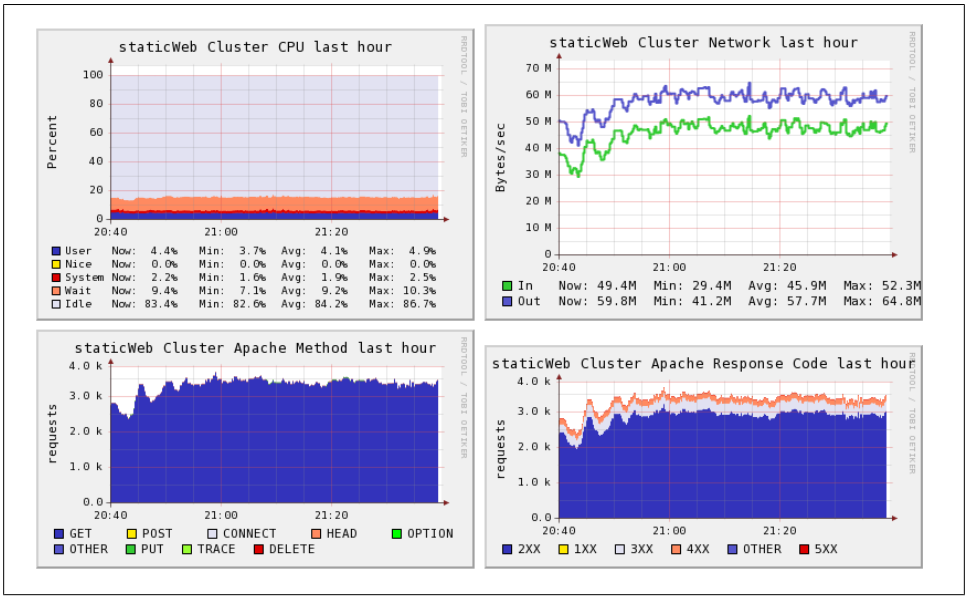


Figure 9-6. Static web cluster performance

The Ganglia chart in Figure 9-7 trends overall request rates and types to a web server cluster serving nonstatic content for the Tagged.com site. An interesting point to note is the large number of HTTP `POST` operations, which is typical of a Web 2.0 site where a large proportion of the traffic is AJAX requests from JavaScript applications running in client browsers.

Again, while Ganglia is used to trend sFlow performance counters from the web cluster, sFlow agents also sample HTTP operations, providing details such as URL, user agent, client address, and response time. This additional detail is used to identify performance problems with specific services.

The table in Figure 9-8 clearly shows that uploading photos is the slowest operation, which is unsurprising, as uploading a photo involves a significant data transfer and many users have ADSL connections with poor upstream bandwidth. This data also allows Tagged to track response time for popular content, such as the Pets game. This information helps the operations and development teams work together to deliver faster response times, keeping users happy and increasing revenue.

Java performance

The Ganglia chart in Figure 9-9 trends the amount of heap memory allocated by an application over the course of a week. It turns out that this application periodically needed to be restarted, but no one knew why. With sFlow-instrumented Java, fine-grained detail of heap utilization can be tracked. We can see that the garbage collector

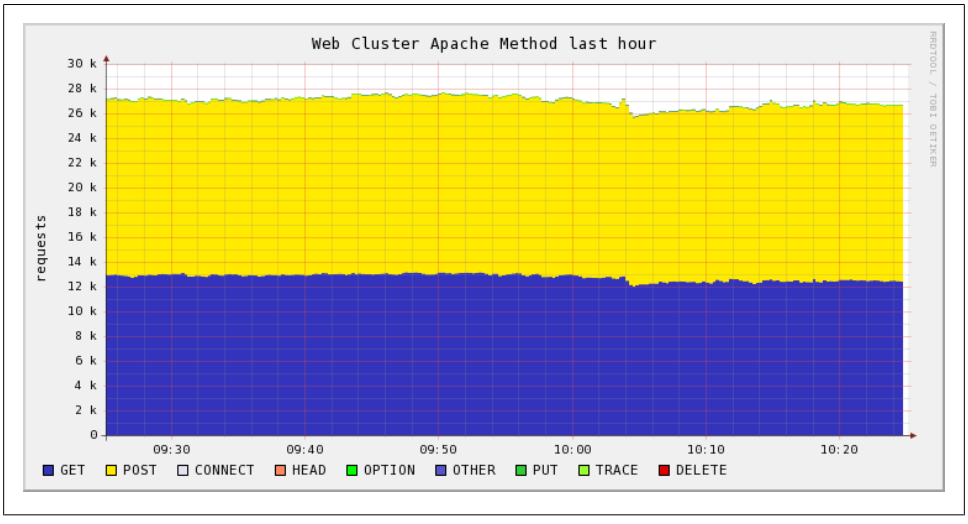


Figure 9-7. Requests to nonstatic web cluster

	URI Path	Duration
	/upload_photo_iframe.html	8.83
	/post_comment.html	1.43
	/topic.html	1.31
	/pets_sold.html	1.03
	/pets_browse.html	725.93m
	/compose_message.html	541.98m
	/messages.html	526.69m
	/home.html	494.59m
	/pets.html	469.24m
	/edit_profile.html	453.19m
	/profile.html	430.15m
	/api	420.92m
	/meetme.html	333.72m
	/meezers	330.73m
	/Desconhecido	264.18m

Figure 9-8. URLs by average duration

is unable to bring the heap on this application back to a steady state, and that over time, it grows and grows until the machine starts swapping. After a restart, the garbage collection is able to keep up again.

Monitoring isn't limited to Java heap memory; there are many additional Java metrics that we can track with Ganglia (see "Java Virtual Machine Metrics" on page 150), allowing Tagged to keep a close eye on the many Java applications running on the site.

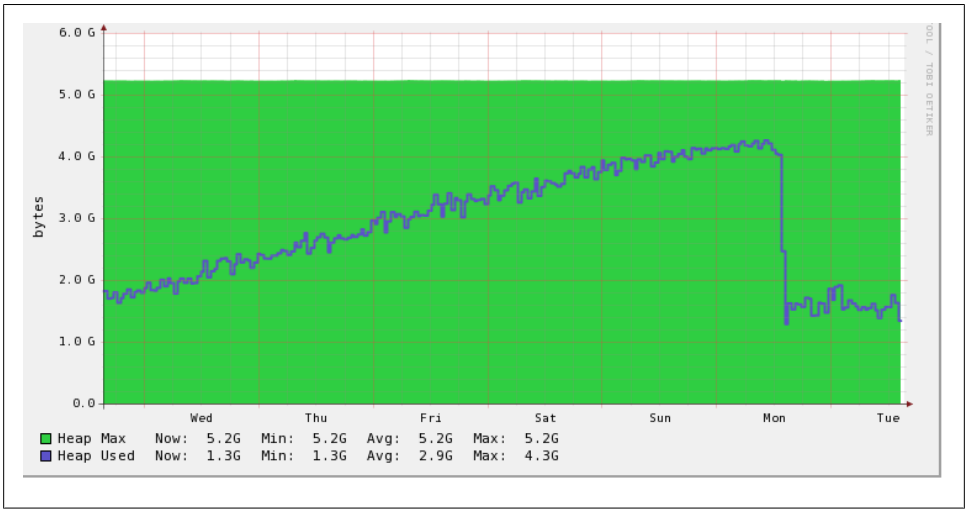


Figure 9-9. Java heap memory

SARA

Ramon Bastiaans, SARA

SARA is a supercomputing center in the Netherlands, founded by and in cooperation with a number of universities.

Amongst other things, SARA provides computational, storage, and networking resources for the Dutch research community and institutions. Their research fields are very diverse and vary from meteorology, chemistry, and astronomy to economics and psychology.

Researchers, students, and teachers may receive access to one or more of the facilities provided by SARA through national subsidiaries or can approach SARA directly. Amongst these facilities are a National Supercomputer "Huygens," a National Computational Cluster "LISA," and various grid computational and storage services.

For more detailed and complete information on SARA, please visit their website.

Overview

We started out using Ganglia to monitor our National Beowulf cluster "Rainbow" around 2002. At the time, this system consisted of about 200 "mini tower" personal computers (Figure 9-10).

Now, 10 years later, we monitor just about every system with Ganglia. At the time of writing, there are about 20 different systems, 1,400 hosts, and more than 18,000 CPUs.

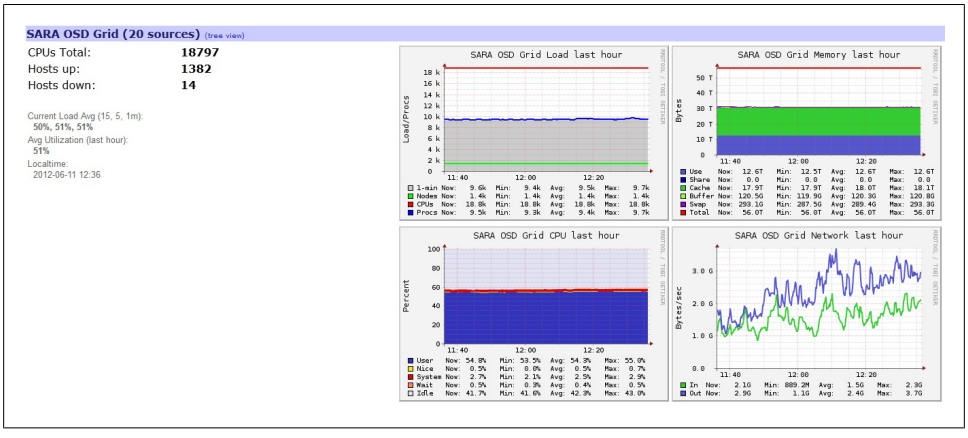


Figure 9-10. The SARA OSD grid with 1,400 hosts

Advantages

Ganglia provides a number of advantages to both system administrators as well as our users.

Operational

For system administrators and consultants, Ganglia provides a few benefits:

- System health at a glance
- Usage and trending
- History

We have a large TV setup in our office that rotates graphs at an interval, displaying several different systems (Figure 9-11).

Ganglia allows us to keep a watch on the general status of the systems we manage. For example, whenever the "running processes" does not match the "1-minute load," or the "gray area" goes higher than the "blue line," we can conclude we have a rogue machine in the cluster. This usually means that a machine is getting overloaded, is crashing, or has some other issues that are out of the ordinary and needs attention. It also allows us to see any peaks instantly.

It also enables us to see how well (or poorly) the system is being used. Low usage or load might indicate a problem with batch job scheduling or inefficient usage of the system. This might result in one of our consultants contacting a user about optimizing his batch jobs. Obviously, it also allows us to see past values for certain statistics.

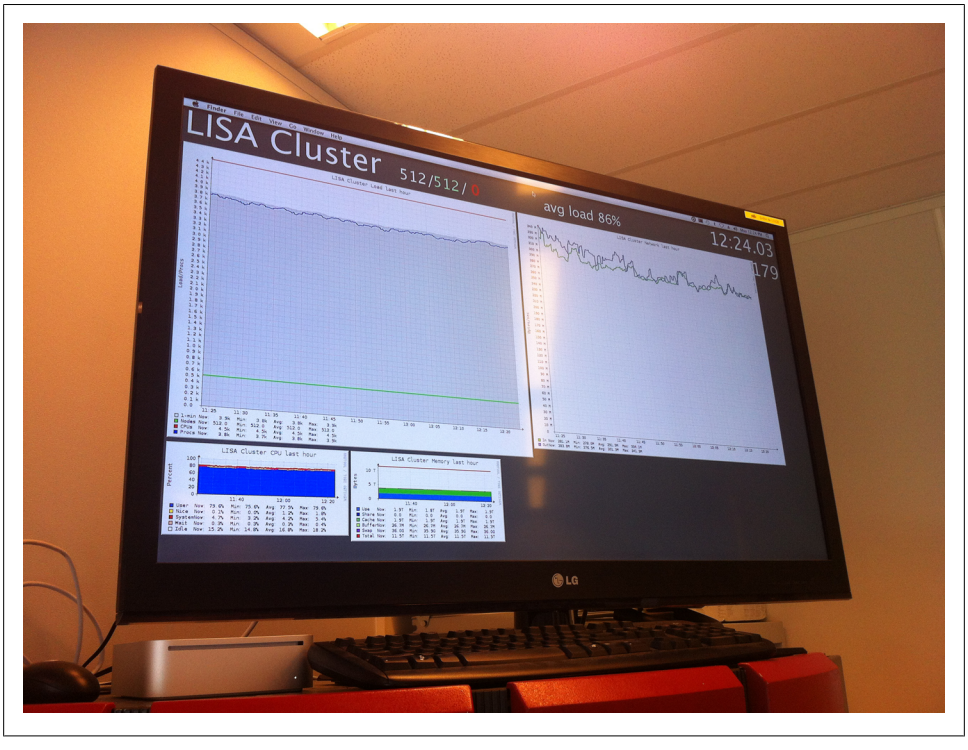


Figure 9-11. Live dashboard for the 512-node SARA LISA cluster

Users

For users, benefits include:

- System availability
- User job health

Our customers or users of the systems can use our Ganglia website (which is public) to see how "busy" the system is and how their job is performing. For example, a user might like to know how much memory his job is using or why the job is not consuming CPU time. In this case, the user could access our Ganglia website to see these statistics.

Customizations

- (batch) job monitoring
- Custom metrics
- Custom graphs and reports

On our computational systems, we use batch job resource management and scheduling software. Thus, users provide a sort of shell script that (amongst other things) contains

a definition of the resources they need (CPUs, memory, etc.) and the tasks they would like to run.

The batch system in conjunction with the scheduler decides when the job is actually run. This decision is based on various things such as resource availability, network fabric, fair share and backfilling mechanisms, and many more. Thus, users can't always be sure when their calculation is run.

We have developed an add-on to Ganglia called Job Monarch that interfaces with the resource manager and reports in which batch jobs are running in the cluster, on which nodes, and which resources they have reserved in the batch system (Figure 9-12). Alternatively, this add-on can also store an archive of jobs for historical reference.

For more information on Job Monarch and SARA's other open source projects, please visit here.

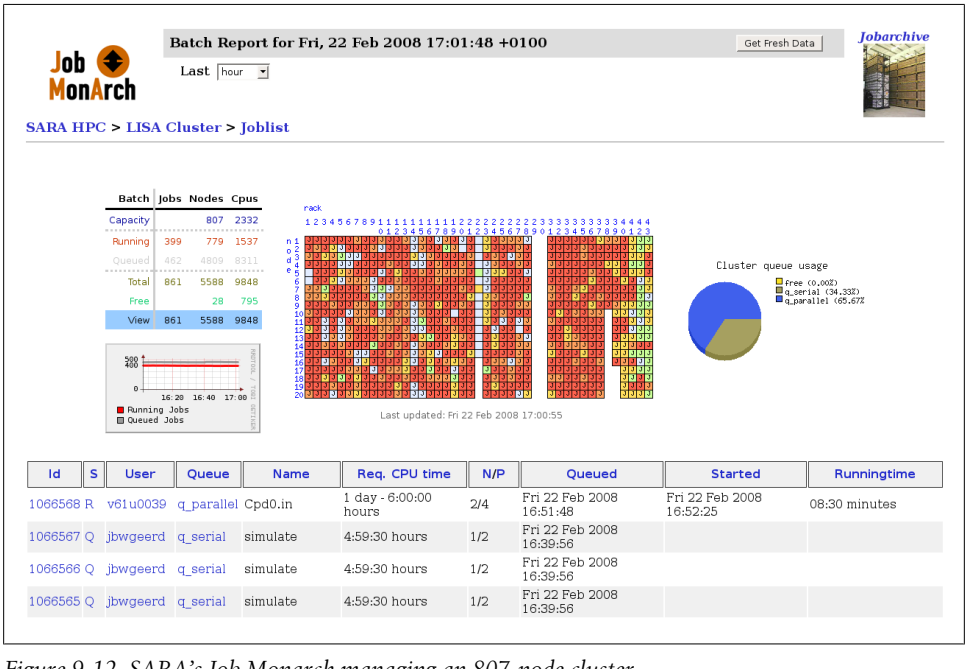


Figure 9-12. SARA's Job Monarch managing an 807-node cluster

Metrics

For operational usage, we report many extra metrics:

- Temperatures
- BIOS/firmware levels
- Support tag/serials

These allow us to see immediately when any machine is out of sync with the latest firmware. Additionally, we report things such as support service tags or serial numbers so that we can easily report any issue.

We use the IPMI and Dell Open Manage software to poll this information from a shell script and then submit the metrics through the use of gmetric.

Additionally, some system-specific metrics are relevant for a particular system:

- Tape drive occupation
- Number of SSH logins
- LDAP connections
- NFS connections/traffic

Thanks to the gmetric utility, the possibilities are pretty much endless. In the latest Ganglia versions, some of these metrics now have modules that report their values including NFS.

Custom graphs

Although custom metrics are nice to have, some metric values make sense only in a proper context in relation to other metrics. In this case, custom graphs or reports can be useful. With older versions of Ganglia, this type of reporting could be achieved using *graph.d* PHP scripts; as of the latest Ganglia releases with JSON definitions, it is fairly easy.

Challenges

Challenges including the following:

- Multicast/network
- Memory usage

One of the key features of Ganglia is its usage of multicast/UDP to submit its statistics. This approach has some advantages but can also be challenging in a complex network layout. Some clusters might have complex network layouts with various hops and switches. Most of the time, UDP multicast is thus not an option.

In addition, Ganglia's monitoring daemon by default receives and sends all information to all machines in the same system. That is no problem if you have only a couple of machines, but once you have hundreds of machines, this is no longer very useful. The situation then arises that a simple compute node has all monitoring metrics of hundreds of machines in memory.

This setup not only starts to consume large amounts of memory but is also not very useful or necessary for the way Ganglia works. Once we got to the point that users'

batch job calculations started to suffer from lack of available memory, we decided to alter our configuration.

By removing the UDP receive channel, or in later versions setting gmond to deaf mode, the compute nodes no longer receive all metrics from all machines, and their memory usage decreased.

Central collector unicast receiver

One of our other approaches to the network issue is to set up a central collector daemon and let all Ganglia daemons send in unicast mode. Unicast does travel through complex network layouts whereas multicast does not. We use this setup in combination with the aforementioned deaf compute nodes.

However, the issue of memory usage remains in this setup. For a cluster containing a few hundred machines, like ours, the memory usage can exceed 1 GB. So make sure that you have enough memory for your central collector.

Server RRD IO

When we started out using Ganglia with only a few machines, the server side of Ganglia was under a tiny load and could handle it easily. However, the more machines we added, the more we realized that the server and its disk IO were starting to become a bottleneck. Because every machine stores about 50 metrics or more and every single metric is 1 file on disk, for 1,800 machines this adds up to almost 100,000 files on disk.

While visiting a conference, I learned that one performance bottleneck was the Linux kernel readahead ability. This kernel feature tries to predict IO actions and accelerate performance by reading ahead. This feature does not work for RRD as the IO access is fairly random. Installing a new RRDtool version that contains a call to disable readahead for RRDs helped a lot for this particular issue. This problem was fixed in version 1.2.24 and 1.3.x. (See this article by David Plonka, Archit Gupta, and Dale Carder.)

Even with readahead disabled, all the RRD activity can still be cumbersome on the Ganglia server side. Because our Ganglia server has enough memory, we now use a RAM disk for the RRD files. Obviously, RAM disks can be dangerous in terms of the loss of files, but as long as you are aware of this risk, you can watch out for it.

We now use a 2 GB RAM disk with Linux tmpfs. The advantage of tmpfs is that it resizes the actual memory usage for the RAM disk, based on utilization. We created an *init.d* script that copies out and writes back the RRDs upon system boot and shutdown. Additionally, we write all RRDs to disk at least once per hour. This way, we can be sure that the loss of metrics is only for the last hour, should the server or the RAM disk somehow fail and the RAM disk contents be lost.

Conclusion

All in all, we are happy with our Ganglia setup, which translates to most of our systems using it.

Although there can be some scaling issues, using some of the techniques described here can help quite a bit. We are confident that our current setup can handle much more, using techniques such as central collector daemons and RAM disks for the server side.

Reuters Financial Software

Thomson Reuters is a well-known provider of real-time financial data, news reporting, business data, and related software products to help customers make best use of Reuters data.

Reuters Financial Software (RFS) and the Risk Management division (which was part of a private equity deal during 2012) produce software to fully automate the operations of the treasury, capturing trade events, monitoring portfolios in real time, detecting risks, and processing payments in the back office.

Ganglia in the QA Environment

Before new versions of software products are released to the clients, it is essential that they pass through a testing regime.

Testing involves two things:

Functional testing
: For example, entering a foreign exchange (FX) spot trade and verifying that it flows to the back office for payment.

Nonfunctional testing
: For example, verifying that the price server process does not use up more memory than the previous release.

It is the nonfunctional testing requirement that has been satisfied by the use of Ganglia, RRDtool, and Nagios.

Because the technical staff have a strong background in development and system administration, the use of these tools provides a convenient way to obtain flexibility.

Market data overload

During the recent financial crisis, there have been times when trading activity has been significantly in excess of normal levels. This typically happens on the days when banks like Bear Stearns and Lehman Brothers collapsed in the United States, or when England saw a bank run on the Northern Rock. For example, a bank handling 300 deals per

hour may have traded 1,000 deals per hour as clients try to remove (perceived) risky stocks from their portfolios.

These surges in trading subsequently lead to a surge in the volume of market data (prices) transmitted from the exchanges. Each time a deal is concluded on an exchange, the price is transmitted to all interested parties through the various networks (Reuters, Bloomberg, and some other minor players). This surge in data comes back to all banks, not just those that are trading more actively than usual.

The Kondor+ application, the flagship front-office product from RFS, receives the market data through a single server process (known as KVS) and distributes the prices to the trader workstations in the bank.

One client noticed that their KVS had crashed during the surge in data and the trader workstations subsequently stopped showing the current prices. The client raised a support request, demanding a full explanation and a solution.

Analysis and reproducing the problem

The client had already checked basic metrics. Network cards were operating way below capacity, and CPU was not overloaded.

Therefore, it became necessary for the engineering team at RFS to conduct a more detailed study and simulation of the problem. To create such a simulation, it is necessary to emulate the bank's environment: a database must be created containing a similar number of deals and portfolios.

Fortunately, creating the replica environment was straightforward, and reproducing the problem was also done very quickly, proving that the client would likely face the same problem again if trading volumes went through the roof.

It is at this stage that monitoring became a factor. Not only was it necessary to monitor the CPU and NIC, but it became necessary to focus on the individual process, the CPU core it uses, and its memory footprint. Furthermore, it was possible to modify the application to provide some metrics about its functional load (number of price updates per second, and the time that each price update is queued).

All of the data can be combined and superimposed onto a single graph using the monitoring tools, Ganglia for the system, and some custom scripts around rrdtool to rrdupdate values from the application's log.

As the underlying format of all data is rrdtool, it is easy to rrdgraph all the data together as required. It was very quickly established that KVS response times become slower and slower as the market data volume increases. Memory consumption was not unusual, but a CPU core was overloaded (due to a single thread). The graphs show a linear relationship between the rate of incoming price ticks and the processing delay—up to a point. When the market data level exceeds a certain threshold, the relationship between the tick rate and processing time ceases to become linear and demonstrates an exponential increase. Soon afterward, the process crashes.

Validating the solution

Using various profiling tools, some inefficient code paths were identified and optimized. Such a solution requires testing to confirm its validity.

Once again, the simulations were performed and it was observed that the exponential behavior no longer occurred within the trading volumes expected by the client.

The custom rrdtool graphs provide a convenient way to show the client evidence of the testing and also allow them to forecast the market data volumes that they can safely handle in the future.

Ganglia in a Major Client Project

Upgrading a Kondor+ client from an old version of the product to the current version is a major project. One particular client desired to upgrade Kondor+ and the full suite of related products in a single weekend—a tremendously complicated exercise.

Such projects often involve anywhere from six to twelve months of planning and testing before a real upgrade is attempted. This process validates that the bank will be able to complete the upgrade safely.

Upgrading takes too long

In most banks, upgrades are performed on Saturdays, and postupgrade tests are performed on Sundays. There is always a fall-back plan in case a problem is found on a Sunday so that the bank will always be open for business as usual on Monday morning.

This schedule imposes an important nonfunctional test criterion on the project: how long does it take to run the upgrade?

For the project in question, in which the client was upgrading all components of the Kondor Suite on the same weekend, the simulation upgrade required almost 40 hours. Such a long upgrade would not be permitted, as it would not allow enough time for bank staff to test the system on Sunday.

Analysis and studying the problem

This problem required analysis from many perspectives. For example, a specialist from the database vendor was called to verify the configuration of the database server. Application support staff pored over the log files looking for update queries that ran too slowly. System staff look for trends in metrics, such as CPU, IOPS, memory usage, or network IO, that might be correlated with the problem.

As the bank had purchased new servers for the project, there was no monitoring tool available—staff from the bank headquarters were not planning to install their commercial monitoring tool until the project want live.

Consequently, the Thomson Reuters consultants proposed the use of Ganglia as a stopgap measure to enable monitoring of the new servers and some application-specific metrics during the testing phase of the project.

Given the nature of the project (an upgrade), it should not be a big surprise that many of the processes ran on a single thread (using a single CPU core), which is often a bottleneck. In such phases, the project is CPU-bound.

Another common feature of upgrade projects is that all the data in the database is scanned, reconstructing tables one by one where the schema has changed and recalculating values that have new meaning. Behind the scenes, the database server has to perform an enormous amount of work updating indexes during such operations. All of the database activity is typically IO-bound, so the SAN performance is the major bottleneck during these phases of the upgrade.

It was not expected that Ganglia would solve all the problems on this project. In fact, it was only expected to help answer a couple questions:

1. How many hours of the upgrade are CPU-bound, and how many hours are IO-bound? (How often is it bound by one CPU?)
2. During those phases when the upgrade is IO-bound, is the IO performance of the SAN satisfactory?

Using Ganglia for the analysis

To help answer these questions, Ganglia binaries are used from the OpenCSW project for Solaris. In particular, the package `ganglia_modules_solaris` is used to obtain per-core CPU metrics and per-LUN (logical unit number) metrics from the SAN.

The per-CPU metrics are very valuable for visualizing the periods when the upgrade process is bound by a single CPU.

However, the per-LUN metrics (reporting IOPS from the SAN) are slightly more challenging because the database actually stripes across multiple LUNs. Fortunately, with rrdtool as the backend for Ganglia, it is possible to manually define a graph that aggregates metrics for all the LUNs backing each database. Such a graph is shown in Figure 9-13. Notice that the average service time for each LUN is graphed separately (colored lines), while the throughput (MB/s) for all LUNs is summed to create the shaded area graph. Spikes in the shaded area graph show periods when there is a need for significant IO throughput. It is interesting to note that the LUNs don't all demonstrate the same service times during those peaks in throughput; this is a very interesting revelation that encouraged further analysis of the SAN architecture.

Results

The results showed that the project was more often blocked by IO than by CPU, making it clear that remediation should be focused on the IO system itself, or on improving (or

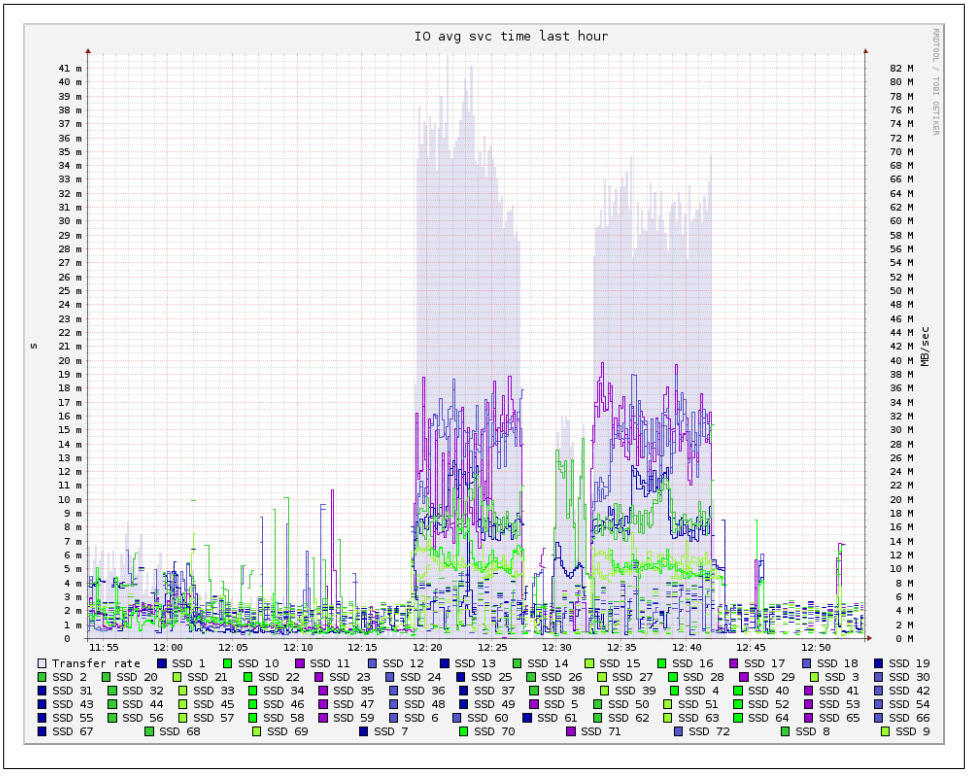


Figure 9-13. IO observations during trial upgrade

eliminating) some of the IO-intensive queries. Furthermore, the visualization with RRDtool made it clear that a particularly long phase of the upgrade was IO bound all the way through.

Lumicall (Mobile VoIP on Android)

Lumicall is an advanced open source mobile VoIP app for Android published as part of the OpenTelecoms.org initiative.

Mobile VoIP started as a niche area, but recent trends suggest that it has the potential to become the dominant paradigm for voice communications. Demand for mobile VoIP is driven by the desire of consumers and businesses to break free of the straightjacket created by the telephone companies. The paradigm of existing mobile communications is largely insecure (as demonstrated by the wholesale level of phone hacking recently exposed in the UK), overpriced (as demonstrated by the need for EU regulators to intervene and set limits on roaming charges), and largely inflexible when compared to modern unified communications solutions.

Nobody expects telephone companies to reform; rather, the increasing power of smartphones (such as those running Android) is enabling consumers and businesses to deploy VoIP apps that work around the cumbersome GSM model. For example, when a Belgian goes to France, he no longer pays roaming charges: he simply locks his phone onto Wi-Fi and uses mobile VoIP.

Although consumers have been quick to experiment with apps like Lumicall and Viber, businesses are still one step behind. One of their key concerns is the need to maintain the quality and consistency of phone calls.

Monitoring Mobile VoIP for the Enterprise

For VoIP communications, a change in network quality (or a change in application performance due to some other app on the phone hogging the CPU) can lead to an almost immediately perceptible change in audio quality.

It is often easy for the user to be disturbed by such issues in audio quality, yet it can be troublesome for an IT department to take responsibility and diagnose the problem, particularly if the user is in a remote location.

In such cases, a comprehensive monitoring solution at the level of the handset is necessary. The solution can gather key metrics about the audio quality (for example, the percentage of packets that are not received) and secondary metrics (such as wireless strength) that may help understand how a quality problem has been induced.

Ganglia Monitoring Within Lumicall

Because the Android platform is based on Java, it is relatively easy to deploy the gmetric4j code into an Android app, which is exactly what has been done with Lumicall.

One particular issue with mobile users is that they often move between different wireless networks and mobile data networks, almost always behind network address translation (NAT). This makes it impossible to probe them continuously using a protocol such as SNMP. However, Ganglia's UDP unicast mode requires communication in only one direction (from handset back to base), so it is an instant success. The same is likely to be true for almost any mobile app, not just a VoIP app like Lumicall.

A more subtle issue is the tendency of Android phones to sleep. Many Java applications, including the original gmetric4j code, rely on regular threading code and use the `Thread.sleep()` method to perform background tasks. This code compiles and executes on Android without any conspicuous error. However, it is obvious that when the phone sleeps, the metrics stop updating. Consequently, the most recent version of gmetric4j allows the timing code to be replaced by a user-supplied implementation, and Lumicall leverages this mechanism to drive all background activity using the Android AlarmManager, a special class that can wake the CPU when a `sleep()` interval elapses. This code can be found in the class `org.lumicall.android.ganglia.AndroidGScheduler`.

Implementing gmetric4j Within Lumicall

The GMonitor instance, the main module of gmetric4j, is encapsulated within an Android service, `GMonitorService`. The service is defined in the Android manifest:

```
<service
    android:name="org.lumicall.android.ganglia.GMonitorService"
    android:enabled="true"/>
```

and started from elsewhere in the application:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    startService(new Intent(this, GMonitorService.class));
```

The specific methods for Android (Wi-Fi and GSM metrics) and some VoIP quality metrics are then defined in the `org.lumicall.android.ganglia.*Sampler` classes. A cut-down version of the `WifiSampler` class, just extracting the Wi-Fi metric, is presented here:

```
public class AndroidSampler extends GSampler {
private Context context;

  public AndroidSampler(Context context) {
    super(0, 5, "lumicall");
    this.context = context;
  }

  @Override
  public void run() {
    Publisher gm = getPublisher();
    publishWifi(gm);
  }

  protected void publishWifi(Publisher p) throws Exception {
    WifiManager wm = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);
    WifiInfo wi = wm.getConnectionInfo();
    double rssi = Double.NaN;
    if(wi.getBSSID() != null) {
      rssi = wi.getRssi();
    }
    p.publish(process, "wifi_rssi",
      Double.toString(rssi), GMetricType.FLOAT, GMetricSlope.BOTH, "");
  }
}
```

The output of this metric is demonstrated by the RSSI graph in Figure 9-14.

Some mobile devices spend little time in Wi-Fi coverage areas, so it is also important to understand the quality of the mobile/cell network. The class `org.lumicall.android.ganglia.TelephonySampler` gathers this metric (see Figure 9-15).

One of the core metrics for a mobile VoIP application such as Lumicall is the latency of packets during a call; Figure 9-16 shows a surge in latency just before 19:20.

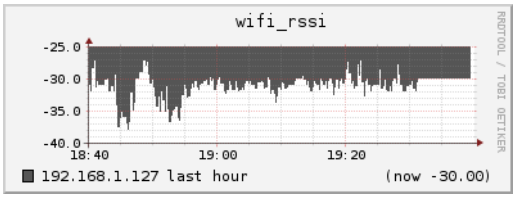


Figure 9-14. Lumicall RSSI metric (Wi-Fi signal level, –dBm)

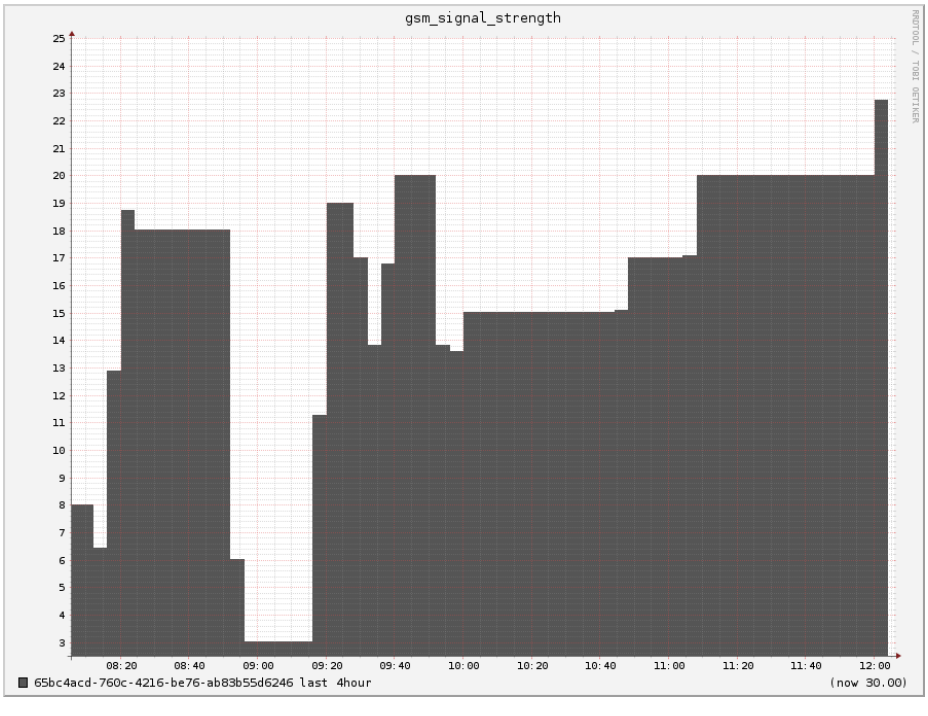


Figure 9-15. Lumicall GSM signal strength metric

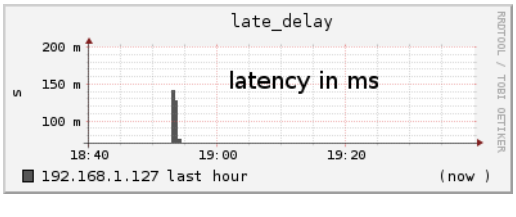


Figure 9-16. Lumicall VoIP latency metric

At the end of the day, does all this monitoring exhaust the phone's battery? Ganglia can monitor that, too! Figure 9-17 shows that the battery still holds a full day's charge.

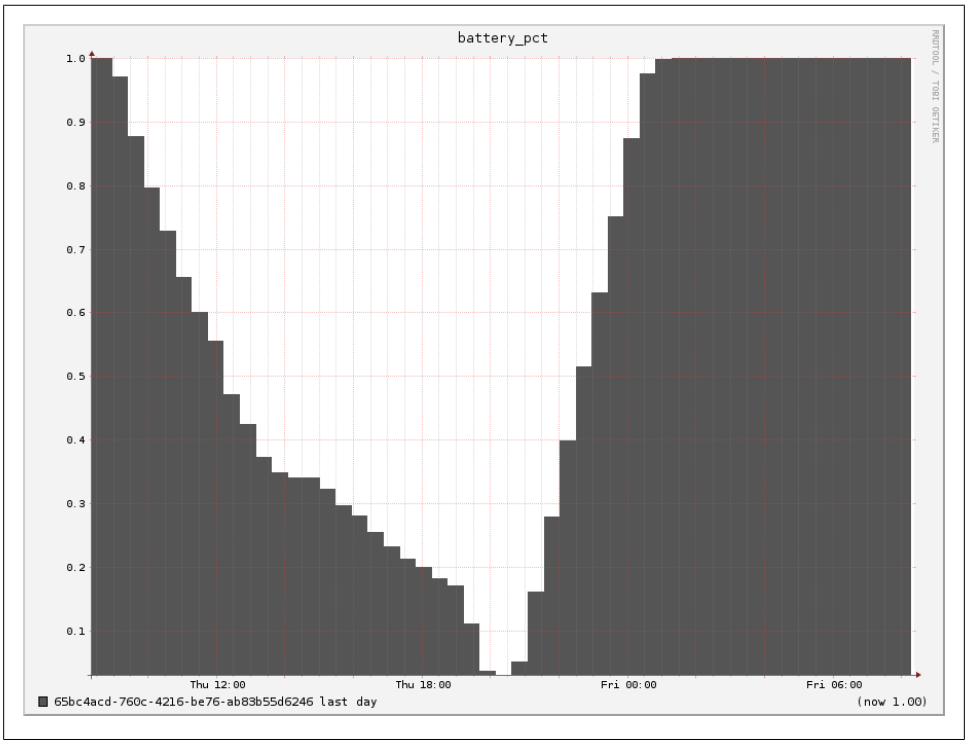


Figure 9-17. Lumicall battery metric

Lumicall: Conclusion

The data collected by gmetric4j in Lumicall and the advanced reporting provided by the new Ganglia web interface provide a unique opportunity for those deploying mobile VoIP to bring all of their quality metrics and useful support metrics within a single framework.

This powerful visualization method is likely to see increases in mobile VoIP service quality, particularly in campus environments where IT managers can tailor the Wi-Fi topology based on real-time data from Lumicall.

Wait, How Many Metrics? Monitoring at Quantcast

Jonah Horowitz, *Quantcast*
Adam Compton, *Quantcast*
Andrew Dibble, *Quantcast*

> Count what is countable, measure what is measurable, and what is not measurable, make measurable.
>
> —Galileo Galilei

Quantcast offers free direct audience measurement for hundreds of millions of web destinations and powers the ability to target any online audience across tens of billions of events per day. Operating at this scale requires Quantcast to have an expansive and reliable monitoring platform. In order to stay on top of its operations, Quantcast collects billions of monitoring metrics per day. The Ganglia infrastructure we've developed lets us collect these metrics from a wide variety of sources and deliver them to several different kinds of consumers; analyze, report, and alert on them in real time; and provide our product teams with a platform for performing their own analysis and reporting.

The monitoring infrastructure at Quantcast collects and stores about 2 million unique metrics from a number of data centers around the world, for a total of almost 12 billion metrics per day—all of which are made available to our monitoring and visualization tools within seconds of their collection (see Figure 9-18 as an example). This infrastructure rests on Ganglia's Atlas-like shoulders.

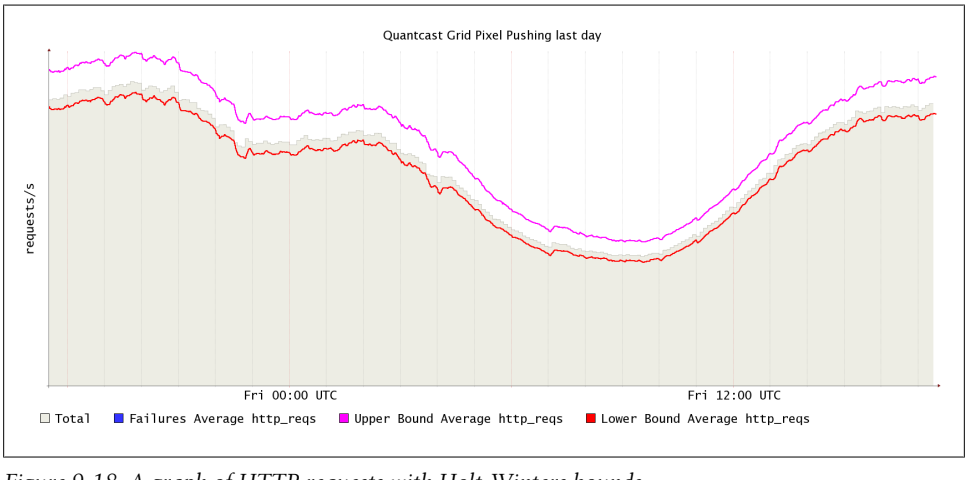


Figure 9-18. A graph of HTTP requests with Holt-Winters bounds

Some of the sources that generate these metrics are:

- Operating system metrics (CPU/memory/disk utilization)
- Application metrics (queries per second)
- Infrastructure metrics (network hardware throughput, UPS power utilization)
- Derived metrics (SLA compliance)
- Business metrics (spend and revenue)

We have a similarly broad spectrum of consumers of monitoring data:

- Alerting tools (Nagios)
- Business analysis of historical trends
- Performance analysis byproduct teams

Ganglia gives us a system that can listen to all of these sources, serve all of these consumers, and be performant, reliable, and quick to update.

Reporting, Analysis, and Alerting

Like many companies, we use Nagios to monitor our systems and alert us when something is not working correctly. Instead of running a Nagios agent on each of our hosts, we use Ganglia to report system and application metrics back to our central monitoring servers. Not only does this reduce system overhead, but it also reduces the proliferation of protocols and applications running over our WAN network. Nagios runs a Perl script called *check_qcganglia*, which has access to every metric in our RRD files stored on ramdisk. This allows us to have individual host checks as well as health checks for entire clusters and grids using the Ganglia summary data. Another great benefit of this approach is that we can configure checks and alerts on the application-level data that our developers put into Ganglia. It also allows us to alert on aggregated business metrics such as spend and revenue.

We have also implemented a custom Nagios alerting script that, through configuration of the alert type, determines which Ganglia graphs would be useful for an operator to see right away in the alert email and attaches those graphs as images to the outbound alert. Typically, these include CPU and memory utilization graphs for the individual host as well as application graphs for the host, cluster, and grid. These help the on-call operator immediately assess the impact of any given alert on the overall performance of the system.

Holt-Winters aberrance detection

Because Ganglia is built on RRDtool, we're able to leverage some of its most powerful (if intimidating) features as well. One of these is its ability to do Holt-Winters forecasting and aberrance detection. The Holt-Winters algorithm works on data that is periodic and generally consistent over time. It derives an upper and lower bound from historical data and uses that information to make predictions about current and future data.

We have several metrics that are well suited for Holt-Winters aberrance detection, such as HTTP requests per second. This metric varies significantly over the course of a single day and each day of the week, but from one Monday to the next, our traffic is pretty consistent. If we see a large swing of traffic from one minute to the next, it typically indicates a problem. We use Nagios and RRDtool to monitor the Holt-Winters forecast for our traffic and alert if the traffic varies outside of the expected range. This approach

allows us to see and respond to network and application problems very quickly, using dynamically derived thresholds that are always up to date.

Figure 9-19 shows an example of an aberrance. We took a datacenter offline for maintenance, and this triggered aberrance detection.

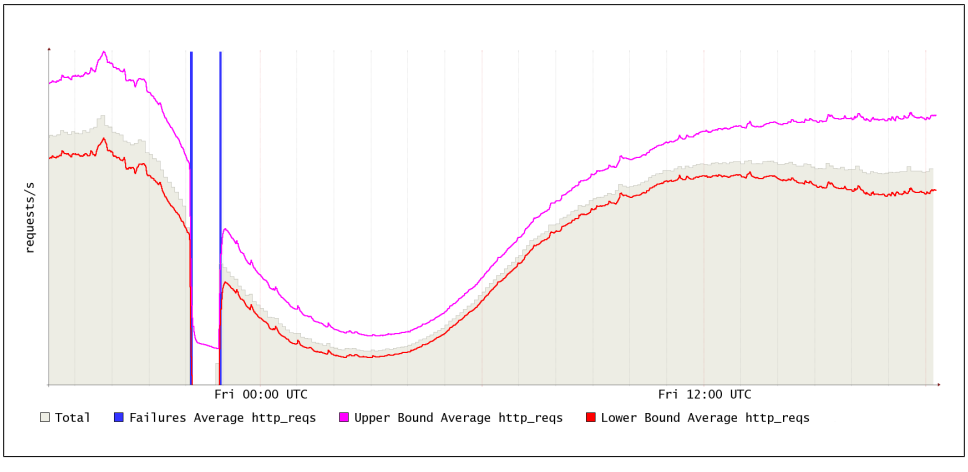


Figure 9-19. An expected aberrance is detected

Figure 9-20 shows an example of an unexpected aberrance alert.

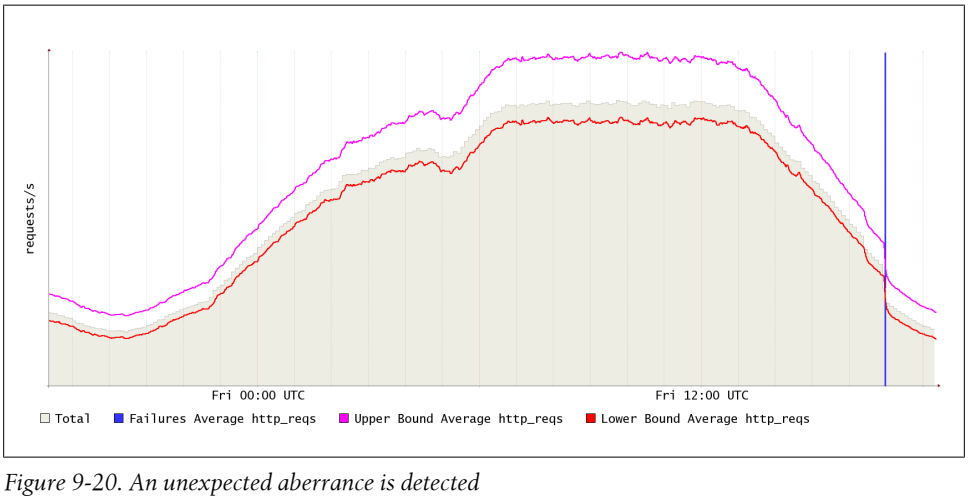


Figure 9-20. An unexpected aberrance is detected

Ganglia as an Application Platform

Because a gmond daemon is running on every machine in the company, we encourage our application teams to use Ganglia as the pathway for reporting performance data about their applications, which has several benefits:

Easy integration for performance monitoring
: The Ganglia infrastructure is already configured, so the application developer doesn't have to do any special work to make use of it. Especially in concert with the `json2gmetrics` tool described shortly, it's easy for an application to generate and report any metrics the developers think would be useful to see. Also, the developer doesn't have to worry about opening up firewall ports, parsing or rotating log files, or running other services devoted to monitoring, such as JMX.

Powerful analysis tools
: By submitting metrics to Ganglia in this way, application developers get easy access to very powerful analysis tools (namely, the Ganglia web interface). This set of tools is particularly useful when troubleshooting problems with an application that correlate with operating system metrics, such as attributing a drop in requests per second to a machine swapping the application out of memory.

Simple and flexible alerting
: Our Ganglia infrastructure is tied in with our alerting system. Once an alert is configured, an application can trigger an alert state by generating a specific metric with a sentinel value, which lets us centralize alert generation and event correlation; for instance, we can prevent an application from generating alerts when the system it runs on will be in a known maintenance window. This approach is much better than the alternative of each application having to reinvent the wheel of monitoring for an invalid state and sending its own emails.

Best Practices

Hard-won experience has given us some best practices for using Ganglia. Following these practices has helped us scale Ganglia up to a truly impressive level while retaining high reliability and real-time performance.

Using tmpfs to handle high IOPS

Collecting the sheer number of metrics we do and writing them to a spinning disk would be impossible. To solve the IOPS problem, we write all our metrics to a ramdisk (Linux tmpfs) and then back them up to a spinning disk with a cron job. Because we have multiple gmetad hosts in separate locations, we're able to protect against an outage at either location. Also, our cron job runs frequently enough that if the tmpfs is cleared (for instance, if the server reboots), our window of lost data is small.

Sharding and instancing

Another way we deal with our large number of metrics is by splitting our gmetad collectors up into several instances. We logically divided our infrastructure into several different groups such as webservers, internal systems, map-reduce cluster, and real-time bidding. Each system has its own gmetad instance, and the only place all the graphs come together are on web dashboards that embed the graphs from each instance. This design gives us the best of both worlds: a seamless interface to a large volume of data, with the added reliability and performance of multiple instances.

Tools

We've developed several tools to get even more use out of Ganglia.

snmp2ganglia

`snmp2ganglia` is a daemon we wrote that translates Simple Network Management Protocol (SNMP) Object Identifiers (OIDs) into Ganglia metrics. It takes a configuration file that maps a given OID (and whatever adjustments it might need) into a Ganglia metric, with a name, value, type, and units label. The configuration file also associates each OID with one or more hosts, such as network routers or UPS devices. On a configurable schedule, `snmp2ganglia` polls each host for the OIDs associated with it and delivers those values to Ganglia as metrics. We make use of the capability to "spoof" a metric's source IP and DNS name to create a pseudohost for each device.

json2gmetrics

`json2gmetrics` is a Perl script that wraps the `Ganglia::Gmetric` library to make it easy to import a lot of metrics from almost any source. The script takes a JSON string with a list of mappings of the form "name: <metric>, value: <value>." With this tool, it's trivial for any program or system to generate Ganglia metrics, and each import requires only one fork (as opposed to one fork per metric with the built-in `gmetric` tool).

gmond plug-ins

We make extensive use of gmond-style plug-ins, especially for operating system metrics. For instance, some of the plug-ins that we've written collect:

- SMART counters about disk health, temperature, etc.
- Network interface traffic (packets and bytes) and errors
- CPU and memory consumed by each user account
- Mail queue length and statistics
- Service uptimes

These metrics are a powerful tool for troubleshooting and analyzing system performance, especially when correlating systems that might seem unrelated through the Ganglia web UI.

RRD management scripts

We have a small army of scripts that work with RRD files:

- Adding new RRAs to existing files, such as a new MAX or MIN RRA
- Adjusting the parameters of existing RRAs
- Merging data from multiple files into one coherent RRD
- Smoothing transient spikes in RRD data that throw off average calculations (particularly common with SNMP OID counters)

Drawbacks

Although Ganglia is both powerful and flexible, it does have its weak points. Each of the following subsections describes a challenge we had to overcome to run Ganglia at our scale.

Necessity of sharding

As our infrastructure grew, we discovered that a single gmetad process just couldn't keep up with the number of metrics we were trying to collect. Updates would be skipped because the gmetad process couldn't write them to the RRD files in time, and the CPU and memory pressure on a single machine was overwhelming. We solved this problem by scaling horizontally: setting up a series of monitoring servers, each of which runs a gmetad process that collects some specific part of the whole set of metrics. This way, we can run as many monitoring servers as we need to reach the scale of our operations. The downside of this solution is significantly increased management cost; we have to manage and coordinate multiple monitoring servers and track which metrics are being collected where. We've mitigated this problem somewhat by setting up the Ganglia web UI on each server to redirect requests to the appropriate server for each category of metrics, so end users see a unified system that hides the sharding.

RRD data consolidation

RRD files offer the following quality: keeping an unbounded amount of time-series data in a constant amount of space. They do so by consolidating older data with a consolidation function (such as MIN, MAX, or AVERAGE) to make room for newer data at full resolution.

For example, an RRD file might be configured to keep a data point at 15-second intervals for an hour. This means keeping 240 unique values (60 minutes/hour × 4 data points/minute). A whole day of data at this resolution would require 5,760 data points!

However, instead of keeping all of those data points, this RRD file might instead be configured to keep just one averaged data point per hour for the whole day. As new updates are processed, a whole hour's worth of 15-second data points would be averaged to get a single data point that represents the entire hour. The file would contain high-resolution data for the last hour and a consolidated representation of that data for the rest of the day.

For Quantcast, we often need to analyze events in a very precise time window. If we saw a CPU load spike for just one minute, for instance, RRDtool's data consolidation would average that spike with the rest of the data for that hour into a single value that did not reflect the spike; the precise data about the event would be lost.

Coordination over a WAN

Quantcast is a global operation, so we needed to make some improvements to Ganglia to get it to a global scale. One of these improvements involved the process of collecting data from a gmond daemon. When a gmetad process collects metrics from a gmond, the data is emitted as an XML tree with a node for each metric. This winds up being a whole lot of highly compressible plain-text data. In order to save bandwidth over our WAN links, we patched gmond to emit a compressed version of that tree, and we also patched gmetad to work with compressed streams. Doing so substantially reduced the amount of monitoring data we were sending over our WAN links in each site, which left us with more room for business-critical data.

Excessive IOPS for RRD updates

Whenever a gmetad process writes a metric to an RRD file, the update process requires several accesses to the same file: one read in the header area, one write in the header area, and at least one update in the data area. RRDtool attempts to improve the speed of this process by issuing `fadvise`/`madvise` system calls to prevent the filesystem from doing normally clever things that interfere with this specific access pattern.

Because we write our RRD files into a tmpfs, the `fadvise`/`madvise` calls were just slowing things down. We wrote a `LD_PRELOAD` library to hook those calls and negate them (`return 0`). This approach allowed us to collect a much larger number of metrics with each gmetad shard and also significantly reduced CPU usage by the gmetad process.

Conclusions

Ganglia is an incredibly powerful and flexible tool, but at its heart, it's designed to collect performance monitoring data about the machines in a single high-performance computing cluster and display them as a series of graphs in a web interface. Quantcast has taken that basic tool and stretched it to its limit by growing it to an exceptionally large scale, extending it across a global infrastructure, and scouring the hidden corners of Ganglia's potential to find new tools and functionality. Most important, though, we've made Ganglia into a platform upon which other services rely. From Nagios alerts

and rich emails to integration with application development, it's become a cornerstone of our production operations.

Many Tools in the Toolbox: Monitoring at Etsy

John Allspaw, Etsy

Monitoring Is Mandatory

Having metrics on your infrastructure should be considered mandatory, not optional. We do wonder about how the underlying foundations of our applications are performing when we're deploying changes, not to mention the continual process of capacity planning.

We use Ganglia in the same vein that we use configuration management at Etsy: as the basic building blocks of our confidence. A common approach in fast-growing web operations is to gather clusters of nodes that do the same work and distribute workload amongst them in one way or another. A node knows what it will be when it's first born into the infrastructure; it's a "Web," or a "Shard," or a "Search" node, and so on. Very rarely (if at all) do we have fewer than two nodes in a single cluster.

Because of this, Ganglia's notion of the node/cluster/grid hierarchy makes it a good fit for tiered web applications. Getting a node into production in its specific role within Ganglia has one requirement: the placement of a single *gmond.conf* file. Because we need no *a priori* knowledge about the host's existence or even its hostname, it will come up in the cluster in Ganglia for virtually no cost, which makes it ideal in an environment cooked by automation.

A Spectrum of Tools

One of the challenges that all open source metrics collection and monitoring tools have is a breadth of scope problem. Successful projects (that is, projects that have solid communities and are reasonably popular, at least in the web operations field) all have something in common: they stick to a constrained scope and don't try to be everything all at once. I think this is what I love about Ganglia, and why we used it at Friendster, Flickr, and now Etsy.

Ganglia sits in a spectrum for us. I like to think of our metrics collection systems as layered. At the lower layers, we have detailed metrics for network devices. SNMP polled tools such as Cacti and our own FITB fit well here.

Above that layer sits Ganglia, picking up the metrics collection ball from network devices and giving us everything we need at a node level, situated in an intuitive arrangement. All host-level basics are covered: CPU, network, disk, memory, processes, and so on. We are then only one line of code (using gmetric) away from getting software-

specific metrics from a node into the aggregation hierarchy: Apache, memcached, Squid, MySQL, PostgreSQL, Lucene/Solr, Postfix, Hadoop... If you can produce a value or statistic from something, Ganglia will gladly and easily collect and display it for you in multiple contexts.

As we are a "dashboard-driven" engineering culture at Etsy (to put it mildly), we need tools that make it as free-as-possible to collect metrics. We want to focus on analysis, not collection. One thing is clear to me: sending ad hoc metrics via gmetric is at least 50 percent of the reason we use Ganglia.

The benefit here is that node-based context is critical in a number of areas, which can't be overstated. Troubleshooting a large install of nodes is largely an exercise in reducing the problem space as quickly as possible. Troubleshooting under time pressure means that I'm going to want to first establish that a single or group of nodes in a cluster is somehow behaving differently. In almost every case, we put nodes into a single cluster because we expect them to behave exactly the same.

With stacked graphs and a single metric view on every node at a cluster level, outliers will stick out easily. I can confirm (or not) the fact that all nodes are behaving identically. If they do, I can drill into them with one action. If they don't, I can eliminate node-level issues and move my process of elimination elsewhere.

Above that sits Graphite, which rounds out the rest of what we need. Some metrics simply don't have a node context. Application metrics such as "registrations," "purchases," and "logins" have rates and volumes that exist in the Etsy context, not the node context. They are actions that depend on multiple nodes in multiple clusters acting together to produce what is, in essence, a feature of our site.

Embrace Diversity

Trying to shoehorn these types of metrics into the node/cluster/grid context that Ganglia is so good at is akin to mowing a lawn with a snowplow. It would dilute the strengths of Ganglia. If I can gather "registrations," which node should I place them in? I can't point to one node (or even a cluster) that would be responsible for registering new members on Etsy, so I would have misplaced context if I put that metric there.

In the same vein, I don't believe that Graphite is appropriate at all for node and cluster-level metrics; the UI and collection methods simply don't encourage this usage and forcing it would be a tough sell for me on a cost/benefit basis. We use Graphite for application-level metrics such as those mentioned previously: statistics on high-resolution values such as page performance and error log line entries.

This is the boundary of Ganglia and Graphite. We have Logster (based on the original Ganglia logtailer), which can derive metrics from log files. We can send these metrics to either Ganglia (for things that make sense in a node context) or Graphite (for things that make sense in the application context), and we use both heavily. So we're leaning

on the strengths of both systems, via the same collection method, that are appropriate for the context.

We aim to layer tools together, in complementary fashion, as part of a palette of tools to gain situational awareness. To support this approach, we have a lightweight dashboard UI framework that can juxtapose graphs from Cacti, FITB, Ganglia, Graphite, and others.

Conclusion

Even in this world of disposable and fully automated infrastructures, node- and cluster-level metrics matter a great deal. This is why we use Ganglia. There are some who think that this "many tools" approach is wasteful, that there must be a singular "One Tool to Rule Them All." I don't believe that idea is true or even possible. I have yet to see an implementation of any metrics collection system that is the best at every one of these layers—only individual tools that are the best at one of the layers.

To those who aspire to create such an omniscient tool that can elegantly handle metrics collection for every device possible, along with flexible anomaly detection, alerting, and escalation—all at the same time remaining human-centered enough to be used efficiently: I salute you, and godspeed.

In the meantime, we have work to do. And Ganglia helps us do it.

APPENDIX A

Advanced Metric Configuration and Debugging

Module Metric Definitions

The following tables describe the metric modules that are part of the distribution of the Ganglia monitoring system. In addition to these metric modules, there are also a number of other modules that are available through the Ganglia module git repository. As new modules are developed, many developers share them with the Ganglia community through the Ganglia module repository. The Ganglia module git repository is open to the public, and the modules are free to download and use. Some of the additional modules that are available from this repository include modules for monitoring an Apache Web Server, MySQL database, and Xen virtual machine, as well as Tomcat and Jetty servlet monitoring through JMX.

Mod_MultiCPU

Prior to the introduction of Mod_MultiCPU, gmond was able to produce only a single CPU-related value for each of the various CPU metrics that it reported. If the hardware architecture supported multiple CPUs, gmond reported only the overall usage rather than the usage for each individual CPU. The Mod_MultiCPU module is capable of detecting how many CPUs exist on the system and constructs the series of metric definitions for each one (Table A-1). Through the configuration of Mod_MultiCPU, all CPU-related metrics can be reported for each CPU detected on the system.

Table A-1. Mod_MultiCPU: monitor individual CPU metrics

Metric Name	Description
`multicpu_user`	Percentage of CPU utilization that occurred while executing at the user level
`multicpu_nice`	Percentage of CPU utilization that occurred while executing at the nice level
`multicpu_system`	Percentage of CPU utilization that occurred while executing at the system level

Metric Name	Description
multicpu_idle	Percentage of CPU utilization that occurred while executing at the idle level
multicpu_wio	Percentage of CPU utilization that occurred while executing at the wio level
multicpu_intr	Percentage of CPU utilization that occurred while executing at the intr level
multicpu_sintr	Percentage of CPU utilization that occurred while executing at the sintr level

Mod_GStatus

The Mod_GStatus module (Table A-2) started out as a metric module debugging tool when the modular interface was first introduced into gmond. The purpose of this module was to detect and report all of the metric gathering, data packet sends, and receives as gmond was running. In other words, as gmond is capable of monitoring every aspect of the system, why shouldn't gmond also monitor itself? Some of the metrics that Mod_GStatus reports are the number of metadata packets sent and received and the number of value packets sent and received, as well as the overall totals. It also tracks any failures in the system. If gmond is incapable of sending or receiving a packet of any kind, Mod_GStatus will report these failures as well.

Table A-2. Mod_GStatus: monitor gmond metrics

Metric Name	Description
gmond_pkts_recvd_value	Number of metric value packets received
gmond_pkts_recvd_metadata	Number of metric metadata packets received
gmond_pkts_sent_value	Number of metric value packets sent
gmond_pkts_recvd_failed	Number of metric packets failed to receive
gmond_pkts_sent_metadata	Number of metric metadata packets sent
gmond_pkts_recvd_request	Number of metric metadata packet requests received (multicast only)
gmond_pkts_sent_request	Number of metric metadata packet requests sent (multicast only)
gmond_version	gmond version
gmond_pkts_recvd_all	Total number of metric packets received
gmond_pkts_sent_all	Total number of metric packets sent
gmond_version_full	gmond full version
gmond_pkts_recvd_ignored	Number of metric packets received that were ignored

Multidisk

The Multidisk module (Table A-3) was introduced as one of the new metric gathering modules for many of the same reasons as Mod_MultiCPU. In previous versions of gmond, the metrics that reported disk space added up the totals for all disks and reported this value for total disk space and used space. The Multidisk module provided

a way to report disk usage metrics for each individual disk rather than a total of all disks on the system.

Table A-3. Multidisk (Python module): report disk available and disk used space for each individual disk device

Metric Name	Description
`<device name>_disk_total`	Available disk space for each disk device
`<device name>_disk_used`	Amount of disk space used for each disk device

memcached

The memcached module (Table A-4) introduced a way to take a closer look at what is actually happening under the hood of the memory management system. The standard memory metrics only report overall memory usage and totals for the system and do not provide any further details about memory management. This module dives a little deeper into how the memory is being used and can help to point out memory inefficiencies.

Table A-4. memcached (Python module)

Metric Name	Description
`<metric prefix>_curr_items`	Current number of items stored
`<metric prefix>_cmd_get`	Cumulative number of retrieval reqs
`<metric prefix>_cmd_set`	Cumulative number of storage reqs
`<metric prefix>_bytes_read`	Total number of bytes read by this server from network
`<metric prefix>_bytes_written`	Total number of bytes sent by this server to network
`<metric prefix>_bytes`	Current number of bytes used to store items
`<metric prefix>_limit_maxbytes`	Number of bytes this server is allowed to use for storage
`<metric prefix>_curr_connections`	Number of open connections
`<metric prefix>_evictions`	Number of valid items removed from cache to free memory for new items
`<metric prefix>_get_hits`	Number of keys that have been requested and found present
`<metric prefix>_get_misses`	Number of items that have been requested and not found
`<metric prefix>_get_hits_rate`	Hits per second
`<metric prefix>_get_misses_rate`	Misses per second
`<metric prefix>_cmd_get_rate`	Gets per second

Metric Name	Description
<metric prefix>_cmd_set_rate	Sets per second
<metric prefix>_cmd_set_hits	Number of keys that have been stored and found present
<metric prefix>_cmd_set_misses	Number of items that have been stored and not found
<metric prefix>_cmd_delete	Cumulative number of delete reqs
<metric prefix>_cmd_delete_hits	Number of keys that have been deleted and found present
<metric prefix>_cmd_delete_misses	Number of items that have been deleted and not found

TcpConn

The TcpConn metric module (Table A-5) provides a way to look at TCP network connections in an effort to detect problems or misconfiguration. By monitoring the TCP connection activity on the system, this module can help point out issues that may affect network latency or the inability to send or receive data in an efficient manner. This module also introduced a new pattern for how to write Python metric modules that include threading and caching. Because the TcpConn module relies heavily on the netstat Linux utility to acquire TCP metric data and the fact that gmond is not a multithreaded daemon, the module doesn't want to cause any delays in the gmond gathering process due to latency in calling an external process utility. In order to avoid any kind of latency issues, the TcpConn module starts up its own gathering thread, which is then free to invoke the netstat utility as required. By invoking the netstat utility within a thread, the module is able to gather the TCP connection related values without having to worry about delaying the gmond gathering process. As the metrics are being gathered from within the thread, these values are stored in a shared cache that can be accessed quickly whenever gmond asks the module for its metric values. Introducing threads through Python is actually a very convenient way to make a single-threaded gmond daemon act as if it were multithreaded.

Table A-5. TcpConn (Python module): monitor TCP connection states

Metric Name	Description
tcp_established	Total number of established TCP connections
tcp_listen	Total number of listening TCP connections
tcp_timewait	Total number of time_wait TCP connections
tcp_closewait	Total number of close_wait TCP connections
tcp_synsent	Total number of syn_sent TCP connections
tcp_synrecv	Total number of syn_recv TCP connections

Metric Name	Description
tcp_synwait	Total number of syn_wait TCP connections
tcp_finwait1	Total number of fin_wait1 TCP connections
tcp_finwait2	Total number of fin_wait2 TCP connections
tcp_closed	Total number of closed TCP connections
tcp_lastack	Total number of last_ack TCP connections
tcp_closing	Total number of closing TCP connections
tcp_unknown	Total number of unknown TCP connections

Advanced Metrics Aggregation and You

There are certain types of metrics aggregation that can't easily be accomplished by using only gmond and gmetric submission. The most notable of these instances are "derivative" values and "counters." These require collection and aggregation over time, in the case of derivative values, and are less than optimally collected using counters.

It is worth pointing out, at this point in this text, that Ganglia does have *a* way of dealing with some derivative values. Metrics submitted using a "positive" slope generate RRDs that are created as `COUNTER`s; however, this mechanism is not ideal for situations involving incrementing values that submit on each iteration (i.e., Apache httpd page serving counts without log-scraping).

One of the solutions for dealing with counter values is statsd. It was created by the nice folks at Etsy. It is written in Node.js, though there are quite a few ports and clones available. Table A-6 lists some of these that are available at the time of writing.

Table A-6. statsd implementations

Software	Language	Description
statsd	Node.js	*https://github.com/etsy/statsd*. It should be noted that the original statsd implementation does not have Ganglia/gmetric submission support without an additional module, which is available here.
statsd-go	Go	*https://github.com/jbuchbinder/statsd-go*. This implementation is a fork of the gographite port of statsd, which did not have Ganglia/gmetric submission support at the time of writing.
py-statsd	Python	*https://github.com/sivy/py-statsd*
Ruby statsd	Ruby	*https://github.com/fetep/ruby-statsdserver*
	C	*https://github.com/jbuchbinder/statsd-c*

The protocol for statsd is relatively simple, and most of the statsd servers come with an example client for submitting statsd.

In addition, there is another piece of software called VDED, which can be used to track ever-increasing values.

Configuring statsd

Most statsd instances are fairly similar to configure for submitting metrics to Ganglia. The important considerations should be how that data is represented in your Ganglia instance. For example, statsd and its clones don't have any particular notion of "host," so each statsd instance is tied to submitting metrics that will be associated with a specific Ganglia host.

statsd

The original statsd instance requires the additional Ganglia NPM module to be installed, using `npm install statsd-ganglia-backend`. It can then be configured by adding `statsd-ganglia-backend` to the array of backends and configuring the Ganglia config key in your statsd configuration file:

```
{
  ganglia: {
      host: "127.0.0.1"             // hostname/IP of gmond instance
    , port: 8649                    // port of gmond instance
    , spoof: "10.0.0.1:myhost.mynet" // ganglia spoof string
    , useHost: 'myhost.mynet"       // hostname to present to ganglia
  }
}
```

statsd-c

statsd-c can be configured to submit values to Ganglia by specifying:

```
-G (ganglia host) -g (ganglia port) -S (spoof string)
```

as part of the starting command line for statsd-c.

py-statsd

py-statsd can be configured to submit values to Ganglia by specifying:

```
--transport="ganglia" --ganglia_host="localhost" --ganglia_port=8649
--ganglia_spoof_host="statd:statd"
```

as part of the starting command line for py-statsd.

Configuring VDED

VDED has some of the same constraints as statsd, except that it is not constrained to submit all values as if they belonged to a single host. The optional "spoof" parameter for submitting metrics to VDED allows different spoof arguments to be associated with

different tracked metrics. It is a good idea to remember that VDED aggregation is limited to the cluster of which the receiving gmond instance is a member.

The command-line arguments for VDED, which are managed in */etc/vded/config* on RHEL installations, is configured using the following switches:

```
--ghost=(ganglia host) --gport=(ganglia port) --gspoof=(default spoof)
```

rrdcached

rrdcached is a high-performance RRD caching daemon, which allows a larger number of RRD files to be maintained by a gmetad instance without the higher IO load associated with reading/writing those files to and from disk. It can be controlled via a command socket and is distributed with the standard RRDtool packages for most Linux distributions.

Note that rrdcached may be unnecessary if you're using a ramdisk to store your RRD files.

Installing

The rrdcached package can be installed on Debian-based distributions (Debian, Ubuntu, Linux, Mint, etc.) by using `apt`:

```
$ sudo apt-get -y install rrdcached
```

For Red Hat/RHEL-based distributions (Red Hat/RHEL, Fedora, CentOS, etc.), rrdcached can be installed via the `rrdtool` package, which was probably installed already for gmetad to function properly:

```
$ sudo yum install -y rrdtool
```

Configuring gmetad for rrdcached

gmetad can be configured to use rrdcached by setting the `RRDCACHED_ADDRESS` variable in the configuration file included by gmetad's init script. For Red Hat distributions, this is */etc/sysconfig/gmetad*, and for Debian distributions, it is */etc/default/gmetad*. For local sockets, the format `unix:/PATH/TO/SOCKET` should be used to specify the address parameter.

Along with the gmetad configuration change (which will require a restart of any running gmetad processes), it is also recommended that a change be made to the Ganglia web frontend, which will force the frontend to also use rrdcached for forming graphs. Enabling rrdcached support in the web frontend is done by setting the configuration variable `$conf['rrdcached_socket']` to the value of gmetad's `RRDCACHED_ADDRESS`.

Controlling rrdcached

There are a number of useful operations that can be performed by using telnet, netcat, or socat (depending on whether you have a network or Unix socket set up as the control socket). For example, a `FLUSHALL` command forces the rrdcached daemon to flush all RRD data to disk as soon as it can:

```
$ echo "FLUSHALL" | sudo socat - unix:/var/rrdtool/rrdcached/rrdcached.sock
$ echo "FLUSHALL" | nc -v -w 3 localhost 42217
```

Troubleshooting

There are several things that can go awry with an rrdcached Ganglia installation, primarily because an extra layer of complexity has been added.

Permissions

Make sure that the permissions on the rrdcached socket file are permissive enough to allow both the gmetad service user and the web server user to be able to read and write. Failures to communicate via the socket will be visible in gmetad's log.

Delays in metrics

rrdcached uses a series of event logs to cache changes to RRD files before it writes them to disk. Heavy load on the server hosting the rrdcached instance may result in a backlog of metrics that have not been written properly to disk. (Note that this does not indicate that the metrics have been dropped, but rather that the rrdcached file still has not written them to their final location.)

Individual metrics can be flushed to disk by using the rrdcached socket and issuing a `FLUSH` command followed by the full pathname to the target RRD file. This will bring the specified RRD file to the top of the rrdcached job queue. Alternatively, a full flush to disk of all queued RRD updates can be initiated by sending a `FLUSHALL` instead.

Debugging with gmond-debug

gmond-debug is a useful tool for debugging inbound gmetric formatted traffic. It can be used to debug any of the third-party gmetric libraries or to track down most unusual gmetric behaviors.

To install gmond-debug, run the following commands:

```
$ git clone git://github.com/ganglia/ganglia_contrib.git
Cloning into 'ganglia_contrib'...
remote: Counting objects: 479, done.
remote: Compressing objects: 100% (302/302), done.
remote: Total 479 (delta 200), reused 434 (delta 156)
Receiving objects: 100% (479/479), 1.10 MiB | 908 KiB/s, done.
Resolving deltas: 100% (200/200), done.
```

```
$ cd ganglia_contrib/gmond-debug
$ . source.env
$ for i in gems/cache/*.gem; do gem install $i; done
Successfully installed dante-0.1.3
1 gem installed
Installing ri documentation for dante-0.1.3...
Installing RDoc documentation for dante-0.1.3...
Successfully installed diff-lcs-1.1.3
...
Successfully installed uuid-2.3.5
1 gem installed
Installing ri documentation for uuid-2.3.5...
Installing RDoc documentation for uuid-2.3.5...
$
```

Now that you have gmond-debug installed, starting the service is straightforward.

```
$ . source.env
$ ./bin/gmond-debug Starting gmond-zmq service...
With the following options:
{:zmq_port=>7777,
:host=>"127.0.0.1",
:verbose=>false,
:pid_path=>"/var/run/gmond-zmq.pid",
:gmond_host=>"127.0.0.1",
:test_zmq=>false,
:log_path=>false,
:gmond_port=>8649,
:debug=>true,
:gmond_interval=>0,
:zmq_host=>"127.0.0.1",
:port=>1234}
Now accepting gmond udp connections on address 127.0.0.1, port 1234...
```

To test using gmond-debug, point your gmetric submission software at this machine, port 1234. As soon as UDP packets on port 1234 are received, gmond-debug will attempt to decode it and print a serialized version of the information contained therein.

APPENDIX B
Ganglia and Hadoop/HBase

You've got data—lots and lots of data that's just too valuable to delete or take offline for even a minute. Your data is likely made up of a number of different formats, and you know that your data will only grow larger and more complex over time. Don't fret. The growing pains you're facing have been faced by other people and there are systems to handle it: Hadoop and HBase.

If you want to use Ganglia to monitor a Hadoop or HBase cluster, I have good news—Ganglia support is built in.

Introducing Hadoop and HBase

Hadoop is an Apache-licensed open source system modeled after Google's MapReduce and Google File System (GFS) systems. Hadoop was created by Doug Cutting, who now works as an architect at Cloudera and serves as chair of the Apache Software Foundation. He named Hadoop after his son's yellow stuffed toy elephant.

With Hadoop, you can grow the size of your filesystem by adding more machines to your cluster. This feature allows you to grow storage incrementally, regardless of whether you need terabytes or petabytes of space. Hadoop also ensures your data is safe by automatically replicating your data to multiple machines. You could remove a machine from your cluster and take it out to a grassy field with a baseball bat to reenact the printer scene from *Office Space*—and not lose a single byte of data.

The Hadoop MapReduce engine breaks data processing up into smaller units of work and intelligently distributes them across your cluster. The MapReduce APIs allow developers to focus on the question they're trying to answer instead of worrying about how to handle machine failures—you're regretting what you did with that bat now, aren't you? Because data in the Hadoop filesystem is replicated, Hadoop can automatically handle failures by rerunning the computation on a replica, often without the user even noticing.

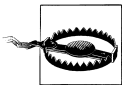
HBase is an Apache-licensed open source system modeled after Google's Bigtable. HBase sits on top of the Hadoop filesystem and provides users random, real-time read/write access to their data. You can think of HBase, at a high level, as a flexible and extremely scalable database.

Hadoop and HBase are dynamic systems that are easier to manage when you have the metric visibility that Ganglia can provide. If you're interested in learning more about Hadoop and HBase, I highly recommend the following books:

- White, Tom. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.
- Sammer, Eric. *Hadoop Operations*. O'Reilly Media, 2012.
- George, Lars. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

Configuring Hadoop and HBase to Publish Metrics to Ganglia

Ganglia's monitoring daemon (gmond) publishes metrics in a well-defined format. You can configure the Hadoop metric subsystem to publish metrics directly to Ganglia in the format it understands.



The Ganglia wire format changed incompatibly at version 3.1.0. All Ganglia releases from 3.1.0 onward use a new message format; agents prior to 3.1.0 use the original format. Old agents can't communicate with new agents, and vice versa.

In order to turn on Ganglia monitoring, you must update the *hadoop-metrics.properties* file in your Hadoop configuration directory. This file is organized into different contexts: `jvm`, `rpc`, `hdfs`, `mapred`, and `hbase`. You can turn on Ganglia monitoring for one or all contexts. It's up to you.

Example B-1 through Example B-5 are example configuration snippets from each metric context. Your *hadoop-metrics.properties* file will likely already have these Ganglia configuration snippets in it, although they are commented out by default.

Example B-1. Hadoop Java Virtual Machine (jvm) context

```
# Configuration of the "jvm" context for ganglia
jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.period=10
jvm.servers=localhost:8649
```

Example B-2. Hadoop Remote Procedure Call (rpc) context

```
# Configuration of the "rpc" context for ganglia
rpc.class=org.apache.hadoop.metrics.ganglia.GangliaContext
rpc.period=10
rpc.servers=localhost:8649
```

Example B-3. Hadoop Distributed File System (dfs) context

```
# Configuration of the "dfs" context for ganglia
dfs.class=org.apache.hadoop.metrics.ganglia.GangliaContext
dfs.period=10
dfs.servers=localhost:8649
```

Example B-4. Hadoop MapReduce (mapred) context

```
# Configuration of the "mapred" context for ganglia
mapred.class=org.apache.hadoop.metrics.ganglia.GangliaContext
mapred.period=10
mapred.servers=localhost:8649
```

Example B-5. HBase (hbase) context

```
# Configuration of the "hbase" context for ganglia
hbase.class=org.apache.hadoop.metrics.ganglia.GangliaContext
hbase.period=10
hbase.servers=localhost:8649
```

I'm sure you noticed some obvious patterns from these snippets. The prefix of the configuration keys is the name of the context (e.g., `mapred`), and each context has `class`, `period`, and `servers` properties.

class
: The `class` specifies what format that metric should be published in. If you are running Ganglia 3.1.0 or newer, this class should be set to `org.apache.hadoop.metrics.ganglia.GangliaContext31`; otherwise, set the class to `org.apache.hadoop.metrics.ganglia.GangliaContext`. Some people find the name `GangliaContext31` to be a bit confusing as it seems to imply that it works *only* with Ganglia 3.1. Now you know that isn't the case.

period
: The `period` defines the number of seconds between metric updates. Ten seconds is a good value here.

servers
: The `servers` is a comma-separated list of unicast or multicast addresses to publish metrics to. If you don't explicitly provide a port number, Hadoop will assume you want the default gmond port: 8649.

If you bind the multicast address using the `bind` option in *gmond.conf*, you *cannot* also send a metric message to the unicast address of the host running gmond. This is a common source of confusion. If you are not receiving Hadoop metrics after setting the `servers` property, double-check your gmond `udp_recv_channel` setting in *gmond.conf*.

Let's work through a few samples.

If you wanted to set up Hadoop to publish `jvm` metrics to three Ganglia 3.0.7 gmond instances running on hosts 10.10.10.1, 10.10.10.2, and 10.10.10.3 with the gmond on 10.10.10.3 running on a nondefault port, 9999, you would drop the following snippet into your *hadoop-metrics.properties* file:

```
# Configuration of the "jvm" context for ganglia
jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.period=10
jvm.servers=10.10.10.1,10.10.10.2,10.10.10.3:9999
```

If you want to set up Hadoop to publish `mapred` metrics to the default multicast channel for Ganglia 3.4.0 gmond, drop the following snippet into your *hadoop-metrics.properties* file:

```
# Configuration of the "mapred" context for ganglia
mapred.class=org.apache.hadoop.metrics.ganglia.GangliaContext31
mapred.period=10
mapred.servers=239.2.11.71
```

If you want to set up HBase to publish `hbase` metrics to a single host with the hostname `bigdata.dev.oreilly.com` running Ganglia 3.3.7 gmond, use the following snippet:

```
# Configuration of the "mapred" context for ganglia
mapred.class=org.apache.hadoop.metrics.ganglia.GangliaContext31
mapred.period=10
mapred.servers=bigdata.dev.oreilly.com
```

For your metric configuration changes to take effect, you must restart your Hadoop and HBase services.

Once you have Hadoop/HBase properly configured to publish Ganglia metrics (Table B-1), you will see the metrics in your gmond XML and graphs will appear in your Ganglia web console for each metric:

```
$ telnet 10.10.10.1 8649
<METRIC NAME="jvm.DataNode.metrics.gcCount" VAL="4" ...
<METRIC NAME="jvm.NameNode.metrics.logError" VAL="0" ...
<METRIC NAME="jvm.NameNode.metrics.maxMemoryM" VAL="888.9375" ...
<METRIC NAME="jvm.DataNode.metrics.logWarn" VAL="9" ...
<METRIC NAME="jvm.DataNode.metrics.memHeapUsedM" VAL="2.4761734" ...
<METRIC NAME="jvm.DataNode.metrics.threadsWaiting" VAL="8" ...
...
```

At this point, you know how to turn on monitoring for any Hadoop context.

The Hadoop JVM metrics cover garbage collection, memory use, thread states, and the number of logging events for each service: DataNode, NameNode, SecondaryNameNode, JobTracker, and TaskTracker.

The Hadoop RPC metrics will give you information about the number of open connections, processing times, number of operations, and authentication successes and failures.

The Hadoop DFS metrics provide information about data block operations (read, removed, replicated, verify, written), verification failures, bytes read and written, volume failures, and local/remote client reads and writes.

The Hadoop MapReduce metrics provide information about the number of map and reduce slots used, shuffle failures, and tasks completed.

Table B-1. List of HBase metrics

Metric Name	Explanation of value
`hbase.regionserver.blockCacheCount`	Block cache item count in memory. This is the number of blocks of StoreFiles (HFiles) in the cache.
`hbase.regionserver.blockCacheEvictedCount`	Number of blocks that had to be evicted from the block cache due to heap size constraints.
`hbase.regionserver.blockCacheFree`	Block cache memory available (bytes).
`hbase.regionserver.blockCacheHitCachingRatio`	Block cache hit caching ratio (0 to 100). The cache-hit ratio for reads configured to look in the cache (i.e., `cacheBlocks=true`).
`hbase.regionserver.blockCacheHitCount`	Number of blocks of StoreFiles (HFiles) read from the cache.
`hbase.regionserver.blockCacheHitRatio`	Block cache hit ratio (0 to 100). Includes all read requests, although those with `cacheBlocks=false` will always read from disk and be counted as a "cache miss."
`hbase.regionserver.blockCacheMissCount`	Number of blocks of StoreFiles (HFiles) requested but not read from the cache.
`hbase.regionserver.blockCacheSize`	Block cache size in memory (bytes), that is, memory in use by the BlockCache.
`hbase.regionserver.compactionQueueSize`	Size of the compaction queue. This is the number of Stores in the RegionServer that have been targeted for compaction.
`hbase.regionserver.flushQueueSize`	Number of enqueued regions in the MemStore awaiting flush.
`hbase.regionserver.fsReadLatency_avg_time`	Filesystem read latency (ms). This is the average time to read from HDFS.
`hbase.regionserver.fsReadLatency_num_ops`	Filesystem read operations.
`hbase.regionserver.memstoreSizeMB`	Sum of all the memstore sizes in this RegionServer (MB).
`hbase.regionserver.regions`	Number of regions served by the RegionServer.
`hbase.regionserver.requests`	Total number of read and write requests. Requests correspond to RegionServer RPC calls; thus, a single Get will result in 1 request, but a Scan with caching set to 1,000

Metric Name	Explanation of value
	will result in 1 request for each "next" call (i.e., not each row). A bulk-load request will constitute 1 request per HFile.
`hbase.regionserver.storeFileIndexSizeMB`	Sum of all the StoreFile index sizes in this RegionServer (MB).
`hbase.regionserver.stores`	Number of Stores open on the RegionServer. A Store corresponds to a ColumnFamily. For example, if a table (which contains the column family) has three regions on a RegionServer, there will be three stores open for that column family.
`hbase.regionserver.storeFiles`	Number of StoreFiles open on the RegionServer. A store may have more than one StoreFile (HFile).

Index

Symbols

A

B

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Authors

Matt Massie open sourced Ganglia in 2000 while working as a Staff Researcher at the University of California, Berkeley. He designed Ganglia to monitor a shared computational grid of clusters distributed across the United States for scientific research. In 2010, he contributed a chapter on cluster monitoring for the O'Reilly book *Web Operations: Keeping the Data On Time* by John Allspaw and Jesse Robbins. Matt is currently a software engineer at Cloudera and is focused on Apache Hadoop enterprise management and monitoring.

Bernard Li is a High-Performance Computing (HPC) Systems Engineer at Lawrence Berkeley National Laboratory. He is currently one of the maintainers of the Ganglia project. He has been involved with HPC since 2003 and has worked on Open Source projects such as OSCAR, SystemImager, and Warewulf.

Brad Nicholes is a member of the Apache Software Foundation and is currently working as a Consultant Software Engineer for Novell. In addition to being a committer on the Apache HTTPD and APR projects, Brad is also a developer as well as one of the administrators of the Ganglia project. As a developer on the Ganglia project, Brad developed and introduced the C/C++ and Python metric module interface into Gangla 3.1.x. He also developed and contributed several of the initial metric modules that currently ship with Ganglia. Brad attended school at the University of Utah and Brigham Young University and holds a degree in computer science.

Vladimir Vuksan (Broadcom) has worked in technical operations, systems engineering, and software development for over 15 years. Prior to Broadcom he has worked at Mocospace, Rave Mobile Safety, Demandware, and the University of New Mexico implementing high-availability solutions and building tools to make managing and running infrastructure easier.

Colophon

The animal on the cover of *Monitoring with Ganglia* is a *Porpita pacifica*, which is found in the tropical Pacific. *P. pacifica*, commonly called the sea money or blue button, is a blue-fringed disc about 1.5 inches in diameter. Its delicate tentacles are sticky and extend from chambers in the gas-filled disc; the tentacles are usually damaged in the surf and reportedly deliver a sting that is not powerful but may cause irritation to human skin.

The blue button lives on the surface of the sea and consists of two main parts: the float and the hydroid colony. The hard golden-brown float is round, almost flat, and about 1 inch wide. The hydroid colony, which can range from bright blue turquoise to yellow, resembles tentacles like those of the jellyfish. Each strand has numerous branchlets, each of which ends in knobs of stinging cells called nematocysts.

In the food web, its size makes it easy prey for several organisms. The blue button itself is a passive drifter, meaning that it feeds on both living and dead organisms that come in contact with it. It competes with other drifters for food and mainly feeds on small fish, eggs, and zooplankton. The blue button has a single mouth located beneath the float, which is used for both the intake of nutrients and the expulsion of wastes. This species reproduces by releasing tiny medusa, which go on to develop new colonies.

The cover image is from *Beauties and Wonders of Land and Sea*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.