Design and Implementation of a Scalable Network Monitoring System

Vincent Geddes
vincent.geddes@gmail.com

Supervisors:

Greg Kempe
Amazon.com
Amazon Web Services
gkempe@amazon.com

Dr. Michelle Kuttel
University of Cape Town
Department of Computer Science
mkuttel@cs.uct.ac.za

Dr. Patrick Marais
University of Cape Town
Department of Computer Science
patrick@cs.uct.ac.za

November 4, 2008
Abstract

Monitoring systems give network administrators a better view and understanding of their networks. Amongst their many uses, they can be used to audit computing assets, profile resource usage, and pinpoint security problems.

Current monitoring systems have not really explored the limits of monitoring scalability, preferring to focus on other important issues such as reliability and node discovery.

We present a monitoring system that scale to over 100000 nodes. It has minimal local and global overhead, and maintains integrity in the face of transient network failure. Through a hierarchal organisation, our monitoring system can operate in multiple administrative zones.

Since we did not have a large fleet of machines at our disposal, we simulated a large fleet of machines, upon which a smaller network of real machines was overlayed. This system was the testbed for our scalability evaluations.

In addition we include a web service interface, which allows access to our system via HTTP. This frees consumers from the need to implement special clients for interfacing with our system. In

This work is part of a larger project, Panopticon, which is a complete monitoring solution, including a database tier and visualisation client.
## Contents

List of Figures 3
List of Tables 3

1 Introduction 4

2 Background 4
   2.1 Network Management ................................. 5
   2.2 Architecture .......................................... 5
   2.3 Data Transmission Formats ............................ 6
   2.4 Scalability ........................................... 6

3 Design 7
   3.1 Design Methodology ...................................... 7
   3.2 Monitoring Agent ...................................... 7
   3.3 Logical Zones .......................................... 8
   3.4 Metrics ................................................ 8
   3.5 Protocol ................................................ 9
      3.5.1 Syntax ............................................ 10
      3.5.2 Extensibility ..................................... 10
   3.6 Web Service Interface ................................ 10
   3.7 Command-line Utility .................................. 11
   3.8 Design Limitations .................................... 11
   3.9 Summary of Deliverables .............................. 12

4 Implementation 12
   4.1 Monitoring Agent ...................................... 12
      4.1.1 The Zone Tree .................................... 12
      4.1.2 Server Component ................................ 12
      4.1.3 Client Component ................................ 13
      4.1.4 Time Synchronization .............................. 14
      4.1.5 Network Error Tolerance ......................... 14
      4.1.6 Metric Collection ................................ 14
   4.2 SCGI Server ........................................... 15
      4.2.1 Query Support .................................... 15
      4.2.2 Resource Representations ....................... 16
      4.2.3 Concurrent Requests .............................. 16
   4.3 Daemon Processes .................................... 16
   4.4 Deployment ............................................ 17
   4.5 Configuration ......................................... 17
   4.6 Logging ............................................... 17
   4.7 Testing ............................................... 17
   4.8 Software License .................................... 17

5 Evaluation 17
   5.1 Systems Evaluated .................................... 18
   5.2 Local Overhead ....................................... 18
   5.3 Global Overhead ...................................... 19
   5.4 Scalability ........................................... 19
5.4.1 Analysis ................................................. 20
5.5 Uptime .................................................. 21

6 Future Research ........................................ 23

7 Conclusion ................................................ 23

A Sample configuration file ............................... 26
B Sample network information ........................... 26

List of Figures

1 Agent Hierarchy in Astrolabe ......................... 5
2 Zone Hierarchy ........................................... 8
3 Tree Serialization ........................................ 13
4 Organisation of scalability evaluation ................ 21
5 Scalability as a function of network size ............ 22
6 Scalability as a function of physical memory usage . 22
7 Scalability as a function of CPU usage ............... 22

List of Tables

1 Metric Descriptions ..................................... 9
2 Systems Evaluated ...................................... 18
3 Local overhead for terminal agents ................ 19
4 Local overhead for non-terminal agents ........... 19
5 Global overhead ......................................... 19
6 Results for scalability evaluation .................. 20
1 Introduction

Computer networks are complex systems, consisting of many heterogeneous hardware and software components. Monitoring systems are often employed to make computer networks more manageable. There are several benefits to monitoring, including being able to find the sources of failure and inefficiency, as well as being able to plan for network growth.

Monitoring systems provide a subset of the features provided by network management systems. They do not concern themselves with any aspect of management or control other than monitoring. Usually, their behavior is passive, and are unable to influence network devices in any way. Most monitoring systems perform some subset of the following tasks:

- Monitoring resource usage and hardware performance.
- Detecting operating system and application errors.
- Identifying computing assets by their name, location, or hardware characteristics.
- Uncovering security vulnerabilities.

Large companies often have massive data centers which provide the computational and storage resources used to power their enterprise. These data centers are often composed of thousands of nodes built from commodity hardware and free Linux-based operating systems. This has been considered more cost effective than buying mainframes and specialized server hardware. However, the increased multiplicity of nodes requires that monitoring systems be in place to monitor each and every node. Many current monitoring systems are capable of dealing with networks with sizes that range from hundreds to thousands of nodes. This is no longer adequate. Data centers need to grow fast in order cope with the expansion of the Internet.

Our aim is thus to design and implement a monitoring system that is capable of monitoring a very large number of nodes with minimal overhead. For our purposes, we want to be able to monitor a local area network containing at least 100000 nodes. This goal has been achieved.

However, designing a scalable and efficient monitoring system is not a trivial task. Such a system will be highly distributed, and gauging global interactions and effects can be a difficult task. A naive implementation may impose a heavy burden on a network, using up costly computational resources. We perform a systematic evaluation of the our system, evaluating its footprint on the network.

2 Background

It is useful to look at prior work in this field in order to gain understanding of the problem. In this section we review existing approaches to system monitoring and compare and contrast their architectures and features.
2.1 Network Management

Provisions for network monitoring have traditionally been incorporated into network management systems. This is certainly a natural organisation, since in order to manage a network, network administrators would need to be able to monitor it. Management systems would typically provide more features which enable network administrators to be more efficient.

The SNMP (Simple Network Management Protocol) [7] is an example of a network management protocol. Astrolabe [1] is another management system, but with a focuses on monitoring rather than management. On the other hand, systems like Supermon [8] and Ganglia [3] focus exclusively on monitoring. For the latter systems, The main user interaction would be to view and analyse the network information, whilst network administrators would be responsible for installation and configuration.

2.2 Architecture

There are many different kinds of network systems, all designed for different purposes. We can therefore imperative that one chooses the right design for the right kind of network.

Most monitoring systems have a two-level architecture. At the bottom level, monitoring daemons are installed on network nodes, and report useful information. In the terminology of Van Renesse, R et al [1], Case, J et al [7], these are referred to as Agents. At the second level in the architecture, a hierarchical tree of aggregation nodes pull data up from the agents, and make it available to in aggregated form to clients. This second level is the main enabler of scalability.

Ganglia [3, 4], is a prime example of a two-level architecture. It has traditionally been used to monitor federations of clusters. At the cluster level, it uses replication to increase robustness and failure tolerance. Within clusters, each node can act as a source of information for all nodes in the cluster. This is implemented using a multicast protocol in which each node broadcasts its current state to all other nodes. At this level, machine state is replicated us-
ing a binary format. One node in the cluster acts as a representative, and provides collected data to an aggregation daemon. In Ganglia, an aggregation daemon repeatedly polls its child sources, each of which then pushes up data. The aggregation daemon then collates this data and makes it available for further use.

Astrolabe [1, 2] is a network monitoring system with a more generic architecture than Ganglia. A distributed system being monitored by Astrolabe can be recursively divided up into a set of mutually-exclusive zones. The leaf zones can be individual files or operating system metrics, whilst zones higher up in the hierarchy would usually represent some administrative grouping of computers, such as a cluster or even a wide area network.

The authors state that Astrolabe is a fault-tolerant and robust system. As in Ganglia, redundancy is used to achieve these goals. Agents will occasionally communicate with each other with the purpose of replicating state. This communication can happen within zones and between sibling zones. However, unlike Ganglia, agents communicate with each other using a unicast (point-to-point), in a random non-uniform fashion.

2.3 Data Transmission Formats

As a monitoring system is usually distributed around the network, captured metrics must be encoded and transferred between components of the system. There are several data formats which can be used for this purpose, each of which have different effects on network performance and artefact usability.

In SNMP [7], state is communicated using the binary ASN.1 format, which is suitable for encoding structured data. In Ganglia [3] two different formats are used. At the cluster level, agents send metrics to each other using the binary XDR format. Compared to ASN.1, XDR is simpler and more suited to sending scalar values. It can still encode structured data though. At the higher level, when aggregating metrics from a federation of clusters, Ganglia agents transfer metrics using XML. This format is text-encoded and human-readable, but is less efficient than XDR and ASN.1 in terms of memory usage and processing time.

Another system, Supermon [8] takes an approach which combines the good qualities of text-encoding and binary-encoding. It encodes data using S-Expressions, a compact text-encoding traditionally used in the LISP programming language. This format is human-readable, yet arguably takes up less memory and is faster to parse than XML. The author further claims that S-Expressions provides good support for composability, that is, the ability for aggregation services to compose data from multiple sources in a flexible manner.

2.4 Scalability

According to the Ganglia paper [3], the system has been tested on a cluster with over 2,000 nodes. However the paper goes on to say that the system is not designed for dealing with clusters of that large size, leading to an unsuitable level of network traffic.

In the Supermon paper [8], the author states that his system is designed to monitor metrics at a very high frequency refresh rate, and that there is no
suitable scaling mechanism which would allow his system to monitor very large networks while keeping the refresh rate high. In Supermon, aggregation daemons did not help scaling, as they increased network traffic to a point that had a negative effect on refresh rate.

3 Design

The principle aim of this work is the design of a scalable monitoring system. There are several challenges involved in designing such a system. Monitoring a relatively large fleet of computers using only one or two collection nodes is not feasible, as they would soon be overwhelmed by task of processing that many computers. To provide scalability, we build on the work of Ganglia [1] and Astrolabe [2] and introduce a hierarchal tree of aggregation nodes.

In a large network with many nodes, a naive monitoring system may itself impose a burden on network and processing resources. This can also effect the quality of statistics gathered. Our system uses an efficient binary protocol to transmit state, minimising network load and processor usage (parsing textual data is relatively processor intensive). For third-party consumers, the system does provide an interface for accessing collected state in a convenient textual format.

Our system has a generic design which makes it widely applicable to a wide assortment of network configurations. It can be used to monitor a small cluster or it can be used to monitor a large scale grid. It is only up to the network administrator to configure the monitoring system as appropriate.

3.1 Design Methodology

We chose an iterative approach to the design and implementation of our system. Solutions with more and more functionality were progressively developed. An emphasis was made on the system being buildable and runnable at all times. Large design changes were first evaluated as separate prototypes.

3.2 Monitoring Agent

An agent is the basic component from which the monitoring system is assembled. An agent is installed on each network node that requires monitoring. The agent runs in the background as a daemon process and performs its activities as required. Our agent has three key responsibilities:

1. Monitor the host system and record interesting metrics
2. Consume state from a number of other agents
3. Provide state to other agents on request

Scalability is achieved by having a hierarchal tree of agents. Each agent aggregates state from child agents and makes it available for other interested agents. More and more nodes can be monitoring by increasing the depth of the zone tree.
3.3 Logical Zones

In the same spirit as Astrolabe [1], the network being monitored will be recursively divided up into mutually exclusive zones. Each zone has a human readable name. For example, a single host can be identified as a zone. Its immediately enclosing local area network can likewise be identified as a zone.

The purpose of having zones is to clearly mark administrative boundaries, as well as provide a means of identifying networks and the relationships between them. These requirements are not fully expressable through traditional network identification schemes (such as hostnames and IP addresses). It also happens that host names are not unique across networks. Zones therefore act as a useful namespaces which allows the system to clearly discriminate between hosts with non-unique names.

Each agent will store a Zone Tree. The root node will represent the zone which the agent’s zone. Children nodes will represent sub-zones. Each node also stores the metrics collected for that zone. If the agent does not have an zone name explicitly set, then the hostname of the host is used implicitly.

3.4 Metrics

Up to 256 different metrics can be measured by the monitoring system. For our scope, the supported metrics and their collection routines will be hard-coded in the agent. Each metric will be identified by a unique integer value in the range \([0, 255]\). This scheme is an enabler of efficient network communication, as only a single byte is required to identify a certain metric. Human readable names are also mapped to metric identifiers. A variety of value types will be supported, corresponding the native data types available in the C programming language.

We have decided upon an initial set of metrics to monitor. We list the most import ones in Table 1.

\[
\begin{array}{|c|c|}
\hline
\text{Metric} & \text{Description} \\
\hline
\text{CPU usage} & \text{Percentage of CPU utilization} \\
\text{Memory usage} & \text{Amount of memory used} \\
\text{Network throughput} & \text{Throughput of network traffic} \\
\text{Disk usage} & \text{Percentage of disk space used} \\
\hline
\end{array}
\]
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostname</td>
<td>Name of host (/etc/hostname)</td>
</tr>
<tr>
<td>address</td>
<td>IPv4 Network Address</td>
</tr>
<tr>
<td>uptime</td>
<td>Number of seconds since last boot</td>
</tr>
<tr>
<td>usage</td>
<td>20 second sample of cpu usage</td>
</tr>
<tr>
<td>load1</td>
<td>1 minute load average</td>
</tr>
<tr>
<td>load5</td>
<td>5 minute load average</td>
</tr>
<tr>
<td>load15</td>
<td>15 minute load average</td>
</tr>
<tr>
<td>transmit</td>
<td>total bytes per second transmitted via network interfaces</td>
</tr>
<tr>
<td>receive</td>
<td>total bytes per second received via network interfaces</td>
</tr>
<tr>
<td>totalram</td>
<td>Total physical ram in bytes</td>
</tr>
<tr>
<td>freeram</td>
<td>Free physical ram in bytes</td>
</tr>
<tr>
<td>processors</td>
<td>Number of cpu cores</td>
</tr>
</tbody>
</table>

Table 1: Metric Descriptions. A selection of the most important metrics which we monitor.

3.5 Protocol

Our protocol is modeled in terms of messages. Agents send a request message and receive a response message in turn. These messages are atomic in the sense that an agent must fully read a request message before formulating a response message. This follows a design principle in distributed systems that reads should be separated from writes.

The agent-to-agent communication protocol should have a low transport and bandwidth costs. To this end, we have chosen to use a binary protocol over a text protocol for several reasons. Creating and parsing a text message takes more time and memory accesses than the equivalent operations for binary messages. Text messages are also generally larger than binary messages and thus require more bandwidth.

Each metric will be encoded as a 2-tuple. The first tuple member is a byte value which identifies the metric and its type. The second member will be the actual metric value. Our design will only support int, long, short, double, and null-terminated string value types.

A request message must specify a method, which is a symbolic identifier which tells the receiver what action to carry out upon receiving the message. The following methods are supported:

- **GET**
  Requests that the receiver return all aggregated zone state.

- **STATUS**
  Requests that the receiver return useful operational information and metadata.

- **VERSION**
  Requests that the receiver return the protocol version.
3.5.1 Syntax

The message grammar is described below. It is described in a simple variant of BackusNaur Form (BNF). The terminals byte, int, short, long are 8, 16, 32 and 64 bits in size respectively.

message: request-message | response-message

request-message: version method
response-message: version status payload?

version: byte
method: byte
status: byte

payload: zone | byte*

zone: zone-name host-name metric-array zone-array
zone-name: byte-string
host-name: byte-string
metric-array: length metric*
zone-array: length zone*
byte-string: length byte*
length: int

metric: metric-identifier metric-value
metric-identifier: byte
metric-value: byte | int | short | long | byte-string

Note that the zone fragment can include other zone fragments. This is due to the fact that each agent serializes its zone tree in a depth-first recursive manner.

3.5.2 Extensibility

The design of the protocol allows for future extension. Additional method and response codes can be supported without changing the grammar. Since messages include version numbers, certain incompatibilities can be resolved gracefully. An agent will return an error response code if it does not understand the method supplied in the request.

3.6 Web Service Interface

Binary protocols are not human-readable and thus suffer from accessibility problems. To ameliorate this situation, we include a web service interface. This interface will be provided by another type of agent, which is gathers state from monitoring agents and makes it available for consumption using an HTTP interface.

HTTP is a text protocol and is thus human-readable. There are several benefits to using HTTP. It enjoys widespread support as it forms part of the
backbone of the world-wide-web. There are a large number of computer professionals who are familiar with the protocol, and many libraries which are able to work with the protocol. HTTP is also a standardized protocol, which ensures good interoperability with a wide variety of web service agents.

Our HTTP interface is designed using the web architecture principles outlined by Roy Fielding [9]. The core idea is model services as resources (which have representations). Complex RPC idioms are also replaced with simpler verbs such as GET resource, PUT resource. Zone information is currently made available from the interface in the textual JSON format\textsuperscript{1}. A semi-formal description of our supported HTTP interfaces are described below.

<table>
<thead>
<tr>
<th>Method</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td><a href="http://server:port/zone">http://server:port/zone</a></td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves state for the specified zone</td>
</tr>
<tr>
<td>Representations</td>
<td>application/json</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>HEAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL</td>
<td><a href="http://server:port/zone">http://server:port/zone</a></td>
</tr>
<tr>
<td>Description</td>
<td>Retrieves metadata about state for specified zone</td>
</tr>
<tr>
<td>Representations</td>
<td>application/json</td>
</tr>
</tbody>
</table>

3.7 Command-line Utility

In addition to the web interface, the design will include a command-line tool which can be used to query an agent. Unlike the web service, this tool has minimal dependencies, and does not require a web server. It can thus be installed on any GNU/Linux machine. We have named this tool \texttt{pget}. The tool will accept a command-line parameter which will be the network address of an monitoring agent. If no such parameter is given, the tool will attempt to contact an agent residing on \texttt{localhost}.

When zone state is requested, the tool will output such state in the JSON format to standard output. This allows the tool to be integrated with many other UNIX commands through input redirection. For example, one can use \texttt{grep}\textsuperscript{2}, to perform a wide range of filtering operations on the state.

3.8 Design Limitations

We have not made allowance for certain types of network failure. If an agent fails permanently, all the zones it has information on will become invisible to any agents higher up in the zone tree. Our main goal in this project was to research scalability and throughput rather than highly reliable distributed systems. However, Our system will be designed to maintain integrity when faced with transient network failure. We will elaborate on this in the implementation section.

\textsuperscript{1}A markup language derived from Javascript: http://www.json.org
\textsuperscript{2}A text matching tool implemented using regular expressions
3.9 Summary of Deliverables

- `pagent`
  The monitoring agent. These monitor various metrics on their host systems and talk to each other using a binary protocol.

- `pwebd`
  Web service interface. Acts as a proxy between a HTTP server and an agent.

- `pget`
  Command line utility for interfacing with agents.

4 Implementation

Our system was written in the D programming language. It is a systems programming language with similar syntax and semantics to C and C++. It has several features, including automatic memory management, which make it a good choice for rapid prototyping and evaluation of ideas.

D programs are statically compiled rather than interpreted, freeing us from the significant overhead of that a virtual machine runtime introduces.

4.1 Monitoring Agent

4.1.1 The Zone Tree

The central data structure in the monitoring agent is the Zone Tree. Each node in this tree represents a zone. There are several attributes which are attached to each node. These include the actual collected metrics for that zone, a zone name, and a dictionary containing child zones. The keys for this dictionary are the names of the child zones.

Several important methods are defined on this tree. The most important are serialization and deserialization methods. The serialization method traverses through the tree and writes each node to a binary representation in a data buffer. The deserialization method does exactly the opposite. It reads bytes from a data buffer and reconstructs the data structures that make up the tree. The serialization method is used to prepare the zone tree for transmission to another agent. Figure 3 depicts these operations.

Several threads compete for concurrent access to the zone tree. As such, several access patterns need to be synchronized in order to prevent race conditions and data corruption. There is one writer thread and several reader threads. No reader thread can read from the tree whilst a writer thread is mutating the tree, and vice versa. Multiple reader threads are allowed to read from the tree at the same time.

4.1.2 Server Component

The server component of the agent is composed of several threads. When a request is sent to an agent, it is processed by one of these threads. Special objects named Servlets do the actual processing of requests.
If the method field of the request is GET, the servlet’s thread must read data from the zone tree. Since other threads may also need to read from the zone tree, we use synchronization primitives to ensure mutual exclusion for access to the zone tree. Only one thread is allowed to read data from the zone tree at any point in time.

Server threads are not created on demand. Rather, a fixed number of threads are launched at startup. When server threads are idle, they are kept in a thread pool. The reason for this implementation decision is that launching a thread on demand is a costly operation, requiring thousands of cpu cycles.

4.1.3 Client Component

The first iteration of our agent talked to each child agent in a separate thread. This was to ensure that multiple child agents could be polled concurrently. This multi-threaded approach resulted in non-deterministic behavior and deadlocks. After much fruitless debugging, we resolved to use an event-based approach to concurrent IO. The event-based approach allows a single thread to perform IO on multiple file descriptors in a concurrent fashion.

The main idea is to use the blocking `epoll_wait()` function which watches for read, write and error events on a set of file descriptors. The system call blocks until events are available or a specified timeout is reached. After the system call returns, we then process each file descriptor for which events occurred. When a read or write event is received for a file descriptor, we can safely perform a non-blocking read from or a write to that file descriptor. We call `epoll_wait()` in a loop.

However, this approach forces us to use a state machine to structure the IO operations, as we can only perform one IO operation for a single event. We

---

3 Network socket connections are also file descriptors
kept counters which indicated how many bytes still needed to be read from
or written to a file descriptor. There are four states, READY, CONNECTING,
WRITING, READING, and ERROR.

Consider a child agent. In the READY state, we are able to commence a
connection, and then move into the CONNECTING state. Once the connec-
tion is made, we move into the WRITING state. We then perform as many
writes as are needed to send the request message. Once the message has been
fully written, we move into the READING state and then perform as many
reads as are necessary until the response message is full read. If any error
events are received we move into the ERROR state.

4.1.4 Time Synchronization

It is useful to include timestamps for collected state. However we do not do
this for several reasons. The central issue is that time may not be synchro-
nized across all nodes participating in the system. This makes timestamps
unreliable, and a possible source of confusion.

There are solutions to the time synchronization problem. The most no-
table of these is the NTP time server which progressively synchronizes time
across all nodes. The major drawback with NTP time servers is that not all
hosts may be configured to use them. This is out of the control of the moni-
toring agent.

4.1.5 Network Error Tolerance

It is a fallacy to assume that network connections are always reliable. There is
always the possibility of connections being abruptly terminated, or network
topologies being temporarily altered due to awry DHCP servers (amongst
other delights). After suffering through such problems, we choose to care-
fully design our system to cope in the face of such problems. Our agent
understands that child agents may be temporary unavailable (for instance,
a failed connect). It will simply ignore a non-responding agent for a short
while after which it will try and poll it again. Similarly, if a connection is
aborted right in the middle of reads or writes, error-handling code takes con-
trol and shifts the state machine for the child agent handler to ERROR.

When a previously responding child agent becomes unavailable, all state
collected from the child agent is purged from the zone tree. This enforces the
invariant that if a zone appears in the tree, it should be alive.

4.1.6 Metric Collection

Metrics were gathered from a variety of sources. We preferred programmatic
interfaces for metric collection. Many of our metrics were collected using the
sysinfo() function available in Linux systems. This function returned val-
ues relating mostly to physical resources such as physical ram, swap space,
etc.

An interesting problem was for an agent to determine the network-visible
IP address of its host. Linux machines can have multiple network interfaces,
so there is no concept of a primary network IP address. Our solution was to
leave address retrieval until a GET request was sent to an agent. The sender
of the request (another agent) can then easily retrieve the IP address of the remote side by using current TCP connection information.

There is no convenient interface for collecting processor usage and network in/out rates. We had to calculate data rates ourselves using known related information. On Linux a unit of cpu usage is known as a jiffy, and on most configurations, is equal to a 0.01 seconds. The number of jiffies since boot is stored in a file called /proc/stat. This file resides on the Linux proc filesystem which is a file-based interface for system information.

The abstract equation below describes how cpu usage was calculated by an agent. The functions user and system return the number of jiffies in user and system mode since boot respectively. The variables $t$, $k$, and $h$ represent the time, the sample period, and jiffy frequency respectively.

$$
usage(t, k, h) = \frac{k \cdot \text{user}(t + k) - \text{user}(t) + \text{system}(t + k) - \text{system}(t)}{h}
$$

(1)

Our agent has a thread which is dedicated to sampling cpu usage and network in/out rates. The sample period is by default 20 seconds, but this can be changed in the configuration file.

4.2 SCGI Server

The web service interface is implemented using the SCGI protocol. SCGI is a protocol for talking to HTTP servers. The HTTP server forwards the request from the client to the SCGI server, which then processes the request and formulates a response. The response is then sent back to the client via the HTTP server. The SCGI and HTTP server communicate via TCP/IP sockets. Cheroke was our choice of HTTP server, mainly because it advertised its strong support for SCGI.

We choose SCGI as it is more scalable than previous HTTP server interfaces such as CGI. CGI is similar to SCGI except that a new process is launched for each HTTP request. The overhead of instantiating a process for each request is fairly large and inhibits the scalability of the web service. SCGI only requires one process which serves multiple requests from the HTTP server.

Another reason for choosing SCGI is that the web service server lives in a separate address space from the HTTP server. If the web service crashes, it does not bring down the HTTP server as well. This is a problem one has to face when using in-process HTTP server extensions (such as that for the PHP and Java programming languages).

4.2.1 Query Support

The web service supports minimal querying capabilities on Zone data. It allows one to select a subtree of the entire zone hierarchy. The query language simply consists of an ordered list of names. Each name corresponds to a zone name. One can think of this list as a path similar to that of a URI or filesystem path, except that zones are addressed rather than web pages or files.
4.2.2 Resource Representations

The web service has interfaces which allows network state to be rendered in multiple representations. The client can inform the SCGI server of the available representations via the HTTP request header `Accept`. This header specifies a list of mime-types of formats that are required. On the other hand, the SCGI server informs clients of available representations via the `Content-type` response header. A client can use the HTTP HEAD method to query the value of this header. Currently, the supported formats are JSON and S-Expressions. These markup formats are derived from the Javascript and LISP programming languages respectively.

The zone tree is serialized into these formats using a recursive procedure which processes each node and writes the serialized data to a buffer.

4.2.3 Concurrent Requests

Multiple web agents may want to request data from the web service concurrently. To support such concurrency, the web interface services each request using a separate thread of control. The service threads are spawned on startup and are kept dormant in a thread pool. We avoid spawning a thread on demand as native threads take thousands of cycles to spawn, which increases contention for the processor and thus limits the amount of concurrent requests the service would be able to handle.

A special thread is responsible for polling a monitoring agent for network state. It currently polls the agent every 20 seconds. The state is stored as a zone tree using the same type of data structures as the monitoring agents.

4.3 Daemon Processes

Our agent is required to run in the background, even when all users are logged off. Processes that run in such a way are known as daemon processes. There is a rather complicated process one has to follow in order to create daemon processes. After executing the agent, the following actions are performed:

1. Fork the running process and terminate the parent. The kernel then moves the child process to be under the main `init` process.

2. Ensure that the process runs in its own session group. This is done using the `setsid()` function. UNIX allows one to send signals to all members of a process group. By making the agent run in its own session group we prevent unwanted signals being sent to the agent.

3. Set the current directory to `/` using `chdir()`. This ensures that the daemon does not prevent mounted drives from being unmounted.

4. Close all open file descriptors. This is necessary as the child process inherits all open file descriptors from its parent.

In addition we also implemented a script which tells GNU/Linux to launch the daemon when the system boots up. This is an important requirement, as nodes may often be temporarily shut down for maintenance. When these nodes are powered on again, their agents processes will start automatically.
4.4 Deployment

Deploying our system was more of a tedious than difficult task. We were faced with the problem of installing agents more than 100 physical machines. These machines were all heterogenous, so our agents had to be compiled for different CPU architectures. To simplify matters, we choose only to support machines running Debian GNU/Linux. We created installer packages with pre-compiled binaries. These packages were then uploaded to a central package repository. Each of the debian systems was then instructed to download the packages and install them. This was done by using a script which used ssh to login into each machine and run the package manager apt-get.

4.5 Configuration

The behavior of an agent can be customized using a configuration file that is installed along with the agent. It is named /etc/pagentd.conf. This is where the zone name and the network addresses of child agents are set. Various other parameters are exposed via this interface.

4.6 Logging

In order to test the implementation, we often needed feedback from the agent daemon. We implemented a logging facility which sent debug messages from an agent to its host’s syslog daemon. This entity is responsible for logging messages on behalf of other daemons. Log messages also get placed in the /var/log/daemon.log file. The level of logging can be specified in the agents configuration file.

4.7 Testing

During implementation, testing was performed on the networks at our disposal. These networks are described in the Evaluation section. Testing was a difficult, as rolling out new versions of the agent was a time-consuming task. Our main testing approach was to inspect the log statements produced by the agents.

4.8 Software License

Our implementation is licensed under the terms of the MIT/X11 license. We feel that the permissive terms of this license encourages the free exchange of ideas.

5 Evaluation

We evaluated the monitoring system on a variety of network systems. We closely follow the methodology used in the Ganglia evaluation [3], and evaluate local and global overhead seperately. The evaluation of local overhead is more concerned with a single agent’s footprint on its host system, taking into account its resource usage. The evaluation of global overhead considers how much bandwidth the entire monitoring system imposes on a network.
To evaluate scalability, we needed to be consider networks with a 1000 and more nodes. Systems this large were not at our disposal. For this reason, we developed a distributed simulation which allowed us to map the topology of a very large virtual network onto a smaller real one.

Our objective is thus to test the monitoring system on real network systems, and then move forward and investigate performance on virtual network systems where we have free reign to explore the upper limits of scalability.

5.1 Systems Evaluated

Our main evaluation system is a cluster on Amazon.com’s EC2 cloud compute service. Each node is a virtualized GNU/Linux system running on top of Xen virtual machine monitors. The cluster consists of 20 SMP nodes, each of which have 4 virtual processors and 15 GB of physical memory. Each node runs a Linux 2.6.18 kernel. For comparison with real-world hardware, the computing capacity of each of these nodes is equivalent to two 1.0 GHz Intel Opteron processors. We choose this system due to its homogenous nature. All nodes run the same operating system and have the same hardware configuration. An agent was installed on each node. One of these agents aggregated state from all the others.

Another network we tested the system on was a general-purpose computing laboratory at the University of Cape Town. It consists of 160 nodes, although only about half of them were ever online at any point in time. Each node had 512 MB of physical memory and one 3.00 GHz Intel Pentium processor. Each node ran Ubuntu GNU/Linux with kernel version 2.6.20. The same monitoring system configuration as in the Amazon cluster was employed.

The final system was a wide area network composed from the systems mentioned above. It consists of a single node on which an aggregating agent is installed. The node ran Ubuntu GNU/Linux 8.04 with kernel version 2.6.24. The node had 1024 MB and a Intel Pentium 4 processor, running at 2800 GHz. The installed agent aggregates state from representatives of both clusters. These clusters reside on different continents, and thus the system constitutes a wide area network.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of nodes</th>
<th>Number of clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Cluster</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>TSL Lab</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>Chen</td>
<td>56</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Systems Evaluated

5.2 Local Overhead

We measured the local footprint of the agents on their host systems. We take CPU and memory usage into account. The top command was used to access resource usage statistics for the agent processes. The evaluation ran for 5 minutes. We executed top in non-interactive mode every minute to obtain a
series of statistics over the evaluation period. Aggregator agents polled child agents every 15 seconds.

<table>
<thead>
<tr>
<th>System</th>
<th>CPU (%)</th>
<th>Physical Memory (MB)</th>
<th>Virtual Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Cluster</td>
<td>&lt;0.1</td>
<td>1.316</td>
<td>68.01</td>
</tr>
<tr>
<td>TSL Lab</td>
<td>&lt;0.1</td>
<td>1.276</td>
<td>61.18</td>
</tr>
</tbody>
</table>

**Table 3:** Footprint for terminal agents. These are agents which are at the bottom of the zone tree. We take the average measurement over a sample group of terminal agents in each cluster.

<table>
<thead>
<tr>
<th>System</th>
<th>CPU (%)</th>
<th>Physical Memory (MB)</th>
<th>Virtual Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Cluster</td>
<td>&lt;0.1</td>
<td>1.380</td>
<td>70.128</td>
</tr>
<tr>
<td>TSL Lab</td>
<td>&lt;0.1</td>
<td>1.970</td>
<td>73.98</td>
</tr>
<tr>
<td>Chen</td>
<td>&lt;0.1</td>
<td>2.520</td>
<td>77.60</td>
</tr>
</tbody>
</table>

**Table 4:** Footprint for aggregator agents. One node in each cluster plays the role of the aggregator.

Tables 3 and 3 show the results from our local overhead evaluations. They show that for small-to-medium sized networks, the local footprint for agents is very low, with CPU usage not really being a factor at all. The reason for this is that the agents have a highly IO-bound nature, with very short CPU bursts.

### 5.3 Global Overhead

We now consider global effects of the monitoring system. Our main consideration here is bandwidth usage. The `iptables` tool is used to monitor incoming and outgoing traffic. It can be configured to monitor certain IP packets based on the source or destination port specified in the packet.

We only need to monitor these metrics on the system the aggregator agent is installed on. We can do this since `iptables` is able to monitor and analyze incoming traffic from terminal agents. This relieves us from the need to monitor anything on the operating systems of the terminal agents.

<table>
<thead>
<tr>
<th>System</th>
<th>Bandwidth (Kbits/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Cluster</td>
<td>12.96</td>
</tr>
<tr>
<td>TSL Lab</td>
<td>23.9</td>
</tr>
<tr>
<td>Chen</td>
<td>36.75</td>
</tr>
</tbody>
</table>

**Table 5:** Global overhead measurements. The aggregator agent on the Chen node aggregates metrics from the EC2 Cluster and the TSL Lab.

### 5.4 Scalability

For testing the scalability of our system, we were faced with a practical problem. We only had less than a 100 machines at our disposable, certainly not
enough to test how scalable our system was. We needed to see how the system would cope when 10000 or more machines were involved. Our solution was to randomly generate fake zones. Each fake zone had randomly generated metric values. Since zones are trees (in that they can have child zones), we essentially created random tree structures.

There is a setting in the configuration file which allows one to set the number of fake zones for an agent. When the agent starts up, it creates the fake zone tree and inserts it into the main zone tree. Zone data read from child agents can still be inserted into the main tree alongside the fake zones.

Seven nodes in the EC2 cluster participated in the evaluation. The benchmark was carried out as follows. One node, named box0, was to be the main focus of evaluation. The agent on it was set up to poll state from five child agents on nodes box1, box2, box3, box4, box5. These five child agents were setup to supply box0 with fake zone trees. Additionally, an agent on box6 was setup to poll state from the agent on box0. The organisation is depicted in Figure 4. The reason for this organisation is that it is mimics how the monitoring system may be used for real networks.

The benchmark consisted of 9 steps. In each successive step, the five child nodes for box0 were instructed to supply larger amounts of fake data. In each step, we recorded the bandwidth, memory, and cpu usage of the agent on box0. The duration of each step was five minutes. Table 6 shows the results obtained in each step of the benchmark.

<table>
<thead>
<tr>
<th>Node Count</th>
<th>Bandwidth (Mbits/s)</th>
<th>Physical Memory (MB)</th>
<th>CPU (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.03</td>
<td>2.44</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0.13</td>
<td>4.72</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0.25</td>
<td>8.72</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>1.22</td>
<td>16.76</td>
<td>0.01</td>
</tr>
<tr>
<td>10000</td>
<td>2.59</td>
<td>31.76</td>
<td>0.02</td>
</tr>
<tr>
<td>25000</td>
<td>6.2</td>
<td>74.76</td>
<td>0.08</td>
</tr>
<tr>
<td>50000</td>
<td>12.28</td>
<td>126.76</td>
<td>0.17</td>
</tr>
<tr>
<td>75000</td>
<td>18.32</td>
<td>188.76</td>
<td>0.29</td>
</tr>
<tr>
<td>100000</td>
<td>21.41</td>
<td>310.76</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Table 6: Results for scalability evaluation

5.4.1 Analysis

From the results in Table 6, we plotted several graphs, showing scalability as a function of bandwidth, memory, and cpu usage. Note that we achieved our goal of trying to monitor at least 100000 nodes in a simulated local area network.

Bandwidth usage increased mostly in a linear fashion, as shown in Figure 5. A likely explanation is that there is a linear relationship between the number of nodes in the zone tree and the size in bytes of the serialized zone tree. Serialized zone tree’s are by far the main constituent of the agent-to-agent messages. Message headers are only a few bytes long. At about $N = 75000$ there is an decrease in the rate of bandwidth usage. Further work needs to be done to investigate this decrease. When $N = 100000$, 21.4 Mbits/s of band-
width was required. This is well within the capabilities of 100Mbits/s and 1Gbits/s local area networks (such as Ethernet). Less positive things can be said about such bandwidth usage on wide area networks. There is simply too much data which needs to be sent, and agents on wide area networks would need to drastically enlarge the polling interval in order to reduce bandwidth usage.

Memory usage was in line with our expectations. We think that the 310 MB of physical memory required to store data for 100000 nodes is not excessive. For instance, consider that the box0 node has 15 GB of physical memory. Thus 310 MB represents only 0.02% of total physical memory.

We still feel that memory usage can be reduced. We think the data structures used to form the zone tree are a bit fat. Through some investigations we found that a buffer containing a serialized zone tree with 100000 nodes requires only 24 MB. The actual Zone tree structure requires space that is an order of magnitude larger than that (around 240 MB). A likely explanation is that each Zone node contains an associative array containing pointers to child nodes. The keys for this associative array are the names of child zones. We need to consider whether a list would be more space-efficient, keeping in mind space-time tradeoffs.

Even if we had unlimited bandwidth and memory, the CPU usage showed in 7 shows that there is a theoretical limit to how many nodes we can process at any instance in time. The cpu usage increases at a exponential rate, and will eventually reach the maximum value of 1.05. Only one thread of control is used to read and deserialize data from child agents. If the CPU spends all its time deserializing data, it cannot poll other child agents every 15 seconds as required. What we can infer from this is that polling frequency decreases indefinitely as CPU usage approaches 100%.

5.5 Uptime

We are able to report very high uptimes for our agent and web service. During testing, we noted that several agent and web service instances had been

---

4 An associate array is a data structure used to map keys to values. A red-black tree is used to implement $O(log(n))$ lookup operations.

5 On an SMP machine, 100% CPU usage is quite feasible for one thread.
Figure 5: Scalability as a function of network size.

Figure 6: Scalability as a function of physical memory usage.

Figure 7: Scalability as a function of CPU usage.
running for more than a week, even while processing requests every minute of the day.

6 Future Research

Many existing monitoring systems have improved failure-tolerance by using peer-to-peer approaches. One nice property of having a peer-to-peer monitoring system is that it allows for a degree of redundancy. In the context of a cluster, each and every node can act as a source of information for the entire cluster. The problem with our hierarchal approach is that if an agent in the zone tree fails, the entire subtree represented by that agent is then invisible to the monitoring system.

We believe that with some modification, our system can support peer-to-peer node discovery and communication. The main idea is let the agents interface with existing peer-to-peer systems. Distributed hashtables are systems which are used to map keys to nodes. They can be used to implement peer-to-peer systems on top of normal IP networks. The agent protocol will need to support two additional methods. One to instruct an agent to collect state from a specified agent, the other to instruct an agent to stop collecting state from a specified agent. A special proxy daemon will be used to interface with the distributed hashtable and the agents.

One research question is how our hierarchal organisation can be complemented with the peer-to-peer organisation. Both models have their benefits and drawbacks. For instance, peer-to-peer communication between agents does not scale when it comes to bandwidth usage and UDP packet rates. The complexity order for peer-to-peer communication is $O(n^2)$, since each node has to communicate with every other node.

7 Conclusion

In this project, we have developed a monitoring system which is capable of monitoring large networks. Our monitoring system is composed from a heirarchal tree of agents. Each agent defines a logical namespace known as a zone. An agent reports a variety of metrics from resource usage to operating system metadata. Recursively dividing the system into zones allows the system to managed in multiple administrative domains.

We have shown that our system has minimal overhead on small to medium sized networks, requiring very little resources. Its resource usage on larger networks is satisfactory, but there is room for improvement. Bandwidth usage can reduced by decreasing the frequency of information updates. This is one method of making our system suitable for monitoring thousands of nodes across a wide area network.

The main thrust of this project was scalability. As such we did not consider certain modes of network failure. Peer-to-peer systems hold great promise in this regard. Data compression is a further avenue of research, and may make our system suitable for wide-area networks where bandwidth is limited.
Our system has some capacity for dealing with unreliable networks. An agent understands that its child agents may be temporary unavailable. It will try and poll dead children every so often in case they come online.

Careful consideration was given to protocol design. The protocol is extensible, supports versioning, and uses an efficient binary encoding for transportation.

A web service interface was developed, which allows the monitoring system to interface with HTTP clients on the internet. This gives consumers access to our system without needing to implement the low-level agent-to-agent protocol. Network state can be delivered in multiple textual representations, including the JSON format. Retrieving network information can be as simple as visiting an URL using a web browser. A by-product of the web service implementation is a high performance SCGI server, which is able to handle concurrent requests.

We have developed installer packages which allow our system to installed painlessly on Debian GUN/Linux systems. A special script ensures that our agent is started up when its host is booted up.

Our work has been used as part of a larger project. It forms the lowest level of the complete Panopticon monitoring system. The storage and visualization components depend on our monitoring system for reliable and high-quality network information.

One key success metric was whether our monitored system could scale to tens of thousands of nodes. Through simulations of extremely large virtual networks, we have shown that our system can handle networks with over 100000 nodes.
References


A  Sample configuration file

This snippet shows a sample configuration file. It sets up an agent to poll other agents on four other nodes. The interval for polling state from child agents is 15 seconds. The logical zone defined by the agent is named `MyCluster`.

```
{
    "Zone" : "MyCluster",
    "VerboseLogging" : true,
    "PollInterval" : 15,
    "Sources" : [ "box0", "box1", "box2", "box3" ]
}
```

B  Sample network information

This snippet shows sample network information produced by the web service and command-line utility (pget). Only one node is being monitored.

```
[
{
    "zone" : [ "chen" ],
    "hostname" : "chen",
    "address" : "127.0.0.1",
    "uptime" : 68668,
    "usage" : 0.00,
    "load1" : 0.57,
    "load5" : 0.43,
    "load15" : 0.29,
    "transmit" : 0,
    "receive" : 0,
    "totalram" : 2112712704,
    "freeram" : 282529792,
    "sharedram" : 0,
    "totalswap" : 3084439552,
    "freeswap" : 3084439552,
    "procs" : 256,
    "processors" : 2,
    "sysname" : "Linux",
    "release" : "2.6.24-18-generic",
    "version" : "#1 SMP Wed May 28 19:28:38 UTC 2008",
    "machine" : "x86_64"
}
]
```