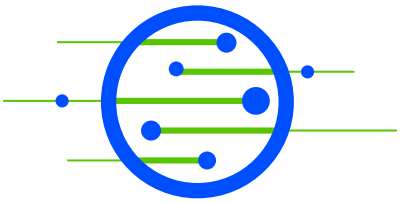

A Simple Common Sensor Interface for PlanetLab

Timothy Roscoe
Intel Research – Berkeley

Larry Peterson
Princeton University

Scott Karlin
Princeton University

Mike Wawrzoniak
Princeton University

PDN–03–010
March 2003

Status: Ongoing Draft, Last Updated May 2003

A Simple Common Sensor Interface for PlanetLab

Timothy Roscoe, Larry Peterson, Scott Karlin, Mike Wawrzoniak

May 13, 2003

1 Introduction

This document goes into some detail in justifying its own existence. If you're sold on the idea and want to cut to the chase, read section 5, followed by 7.

Sensors are our name for the abstraction used by many distributed query processors or management systems for the sources of their data. Sensors encapsulate raw observations that already exist in many different forms, from kernel parameter settings exported through the `/proc` file system, to status information reported through various commands and system calls (e.g., `uptime`, `ping`, and `traceroute`). A sensor might also generate new information from a combination of existing sources.

This document came about because of the number of distributed query processors the authors were aware of which were about to be deployed on PlanetLab: at the very least PIER [4], Sophia [9], IrisNet [7], and TAG [5], and probably Astrolabe [8]. In addition, systems like InfoSpect are already deployed and in daily use.

It is clearly a Good Thing if the data generated on a PlanetLab node can be shared between query processors. Some of this data is generated so trivially (for example, the node's current IP address) that it isn't worth coming up with an abstraction for providing it: clients can simply read it from the kernel. However, many sources of data (such as a service which measures ping times to other nodes, or one that logs connection requests to the node) need to be abstracted behind an interface, for at least one of the following reasons:

- privilege is required to generate the data, but not all the potential clients have (or should have) the required privilege,
- it is expensive to replicate the process by which the data is generated - it is better to have one process generate it and share the result,
- the data cannot be generated multiple times, and so some process is required to duplicate it for each client

Some kind of (possibly trivial) server for a given sensor is therefore needed. We briefly considered using the filing system for this purpose: sensors simply write their data into files which clients can read. We rejected this idea because in the future evolution of the PlanetLab kernel filing systems might vary between slices on a node, which might not even be running the same operating system. Consequently, we opted for a networking-based model where clients connect to sensors via the local IP loopback interface.

The scheme we propose here is intended to be flexible, extensible, and as much as possible avoid getting in the way of what people might want to do, while providing enough structure to make interoperability work.

It's also motivated from an ease-of-programming point of view, bearing in mind also that the data returned by most sensors is expected to be very simple and unprocessed at this level.

2 Definitions

We start by defining a few terms. These should not come as too much of a surprise:

Sensor: A sensor provides a particular kind of information. Sensors are local to nodes: they provide information derived locally.

Client: A client of a sensor is some application which requests the sensor data and hopefully makes use of it. An example might be a particular capsule of a distributed query processor running on a node. This document is about the interface between clients and sensors.

Sensor server: A sensor server aggregates several sensors at a single access point. To obtain a sensor reading, a client makes a request to a sensor server. Sensor servers exist for a number of reasons: they multiplex lots of sensors onto a single access point, they provide controlled sharing of sensors among many clients, and they can, if need be, act as a security monitor to mediate access to sensor data.

Tuple: Each sensor outputs one or more tuples of untyped data values. Every tuple from a sensor conforms to the same schema. Thus a sensor can be thought of as providing access to a (potentially infinite) database table.

3 Data Model

Tuples.

4 Snapshot and Streaming Sensors

Sensor data can arrive in different ways. It may be

Push, pull, history, snapshot. Parameterisation (query).

We divide sensor semantics into two types, *snapshot* and *streaming*.

4.1 Snapshot sensors

Snapshot sensors maintain a finite size table of tuples, and immediately return the table (or, conceivably, some subset of it) when queried for it. This can range from a single tuple which rarely varies (e.g. "number of processors on this machine") to a circular buffer which is constantly updated, of which a snapshot is available to clients (for instance, "the times of 100 most recent connect system calls, together with the associated slices").

Using a notation we just made up, we can write:

```
time ← uptime()
```

or, at the risk of too much notation:

$$\texttt{time} \leftarrow_1 \texttt{uptime()}$$

- to indicate that the `uptime` sensor returns 1 tuple, which is a single value. Similarly,

$$\texttt{min, max, median} \leftarrow_n \texttt{node\_pings()}$$

- indicates that `node_pings` returns an indeterminate number of tuples, each of which consists of three values.

4.2 Streaming sensors

Streaming sensors follow more of an event (or "push") model, and deliver their data asynchronously, a tuple at a time, as it becomes available. A client connects to a streaming sensor and receives tuples until either it or the sensor server closes the connection. Hence:

$$\texttt{src, dest, sport, dport, proto} \twoheadleftarrow \texttt{rx\_packets()}$$

- denotes a streaming sensor which delivers information about packets received at a node, formatted as a five-tuple.

5 Syntax and Protocol

A sensor is accessed via a *sensor server* using HTTP [3].

> **Motivation:** *HTTP is a simple, well-understood protocol. For our purposes, most of the complexity of HTTP can be ignored. HTTP libraries and utilities exist for practically all modern programming languages. HTTP is easy to debug, particularly on a single machine.*

5.1 Sensor URIs

A sensor is addressed using a *uniform resource identifier* (URI) of the form:

`http://127.0.0.1:` *port* `/` *sensorname* [*extension*] [`?` *arguments*]

The *port* identifies the sensor server and the *sensorname* identifies the sensor. A sensor may be optionally parameterized by either an *extension* or a list of *arguments*. An extension is simply an ordered list of elements separated by slashes. Arguments are unordered assignments of values to names and are separated by either ampersands or semicolons. Here are some example sensor URIs:

```
http://127.0.0.1:54321/uptime
http://127.0.0.1:54321/uptime?UNITS=DAYS
http://127.0.0.1:54321/interrupts/bus1/device3
```

Additional points about sensor URIs:

- The sensorname, extension, and arguments are case sensitive.
- Consecutive slashes are not allowed in the URI. The exception is that a double slash is allowed following the HTTP protocol identifier (i.e., `http://`).

- The sensorname and extension are encoded per RFC2396 [2]. (The reserved characters, "`%;/?:@=&+$,`" are replaced by a percent sign followed by two hexadecimal digits corresponding to the US–ASCII code of the character.)
- The field names and values within the arguments are encoded as `x-www-form-urlencoded` [1]. This means spaces are replaced with "`+`" and reserved characters are escaped in the same manner as the sensorname and extension.
- Consecutive ampersands are not allowed in the URI.

5.1.1 Reserved URIs

The following URIs are reserved:

`http://127.0.0.1:` *port* **`/`**
: This form (i.e., no *sensorname*) generates a `text/plain` response containing a list of available sensors (one per line).

`http://127.0.0.1:` *port* **`/`** *sensorname* **`/README`**
: This form generates a `text/plain` response containing documentation about the sensor. It could be as simple as an e-mail contact of the author, a URL for a web page, or a man page.

`http://127.0.0.1:` *port* **`/README`**
: This form generates a `text/plain` response containing documentation about the sensor server.

5.2 HTTP/1.1 Subset

Within the actual protocol of HTTP we are not implementing most of the optional features. To support debugging with browsers, servers should respond with appropriate status codes per RFC2616.

- Servers only implement the `GET` and `HEAD` methods.
- Clients requests need only contain the following fields: `Host`, `Connection`, and `User-Agent`.
- Ranges are not supported.
- Client requests should not contain an entity-body.
- Persistent connections are not supported.
- A normal server response will include the following headers: `Connection`, `Content-Length`, `Content-Type`, `Date`, `Last-Modified`, and `Server`.
- Streaming sensors use *chunked transfer coding* and must include a `Transfer-Encoding: chunked` header line and must *not* include a `Content-Length` header. Trailing headers are not supported.

Appendix A contains additional details.

5.3 Cache Control

For now, we do not take advantage of the caching model in HTTP/1.1. Because the distributed query processors are likely to cache information obtained from sensors, it may be advantageous (in the future) to use server-specified expiration to inform the distributed query processors when they should request fresh data from the sensors.

5.4 Example Request / Response Pair

The request:

```
GET /uptime/ HTTP/1.0
Connection: close
User-Agent: Sophia/1.02
Host: 127.0.0.1:33080
```

The response:

```
HTTP/1.1 200 OK
Date: Mon, 17 Mar 2003 21:56:10 GMT
Server: Kernel_Sensors/0.03
Last-Modified: Mon, 17 Mar 2003 21:56:10 GMT
Accept-Ranges: none
Content-Length: 8
Connection: close
Content-Type: text/plain; charset=iso-8859-1

539220
```

Note that the `Content-Length` includes the trailing `CRLF`. Also note that HTTP *requires* that each line of the header be terminated by `CRLF`.

5.5 Response Body

When feasible, sensor servers should be designed such that the response body is a (possibly empty) list of tuples in comma-separated-value (CSV) format using the ISO-8859-1 character set (that is, `text/plain; charset=iso-8859-1`). See Appendix B for a description of the CSV format.

If it is not feasible to use CSV format (e.g., for binary data or highly structured data), then XML [6] (`text/xml; charset=utf-8`) or binary (`application/octet-stream`) is acceptable.

> **Motivation:** *CSV values are untyped and much to parse than more baroque formats such as ASN.1, XDR, and XML. The lack of structure beyond a simple tuple list is not much of an issue in sensors, since the idea is to keep the data as raw as possible and push as much semantic processing as possible out of the sensor and into the client query processor.*

6 Naming and Discovery

Sensors are effectively addressed by a combination of name and port; a port number on 127.0.0.1 addresses a sensor server. The discovery of sensors (how a client gets hold of a port and/or name) is out of scope of this document, as is the discovery of what parameters a sensor understands, and whether its output is in the form of a stream or finite table.

Instead, we offer some suggestions here for how parts of this problem may be addressed:

- The *documentation* for a sensor implementation should describe which parameters are understood, how they are interpreted, what defaults exist, and the intended interpretation of the output. In addition, the documentation should state whether the results are a stream or finite table (or single tuple).
- Some sensors or sensor servers may exist on well-known ports and/or with well-known names.
- A name service (service discovery service, trader, portmapper, whatever) can be used to lookup locally available sensors on the basis of some search specification or constraint set. Such a service would need sensors to register with it. The interface to such a service might be a sensor itself.
- A sensor server might offer a "meta-sensor" with a well-known name, which gives details of all sensors offered by the server on that port.

7 Examples

Sensors encapsulate raw observations that already exist in many different forms, from kernel parameter settings exported through the `/proc` file system, to status information reported through various commands and system calls (e.g., `uptime`, `ping`, and `traceroute`). A sensor might also generate new information from a combination of existing sources.

This section gives several example sensors that one might implement to the interface defined in the previous section. When implementing sensors, the key design issues are (1) how to aggregate a set of related sensors into a single sensor server, and (2) at what level should a single sensor expose (or synthesize) raw information.

7.1 Kernel Sensors

One obvious sensor server reports various information about kernel activities. The various sensors exported by this server are essentially wrappers around the `/proc` file system. For example, we have already implemented the following set of sensors, expressed using notation I just made up:

`memtotal, memfree, memused` $\leftarrow$ `meminfo()`: returns information about current memory usage; implemented as a wrapper around `/proc/meminfo`.

`load` $\leftarrow_1$ `load()`: returns 1-minute load average; implemented as a wrapper around `/proc/loadavg`, returning the first value.

`load` $\leftarrow_1$ `load5()`: returns 5-minute load average; implemented as a wrapper around `/proc/loadavg`, returning the second value.

`load` $\leftarrow_1$ `load()`: returns 15-minute load average; implemented as a wrapper around `/proc/loadavg`, returning the third value.

`time` $\leftarrow_1$ `uptime()`: returns uptime of the node in seconds; implemented as a wrapper around `/proc/uptime`.

`rate` $\leftarrow_1$ `bandwidth(slice)`: returns the bandwidth consumed by a `slice` (given by a slice id); implemented as a wrapper around `/proc/scout`.

These examples are simple in at least two respects. First, they require virtually no processing; they simply parse and filter values already available in `/proc`. Second, they neither stream information nor do they maintain any history. One could easily imagine a variant of `bandwidth`, for example, that both streams the bandwidth consumed by the slice over that last 5 minute period, updated once every five minutes, or returns a table of the last *n* readings it had made.

7.2 Registry Sensors

Our second example sensor server reports registry information about the PlanetLab nodes. This information is periodically updated (perhaps once per day) from the PlanetLab Network Operations Center (NOC).

`siteID` $\leftarrow_n$ `sitelist()`: returns the list of PlanetLab sites.

`nodeID` $\leftarrow_n$ `nodelist(siteID)`: returns the list of PlanetLab nodes at a given site.

`siteName,latitude,longitude,country` $\leftarrow$ `siteinfo(siteID)`: returns information about a given site.

`ipAddr,dnsName,nodeType,revision` $\leftarrow$ `nodeinfo(nodeID)`: returns information about a given node. The `nodeType` may indicate if this is a normal, alpha, or beta node; the `revision` may indicate the revision of the PlanetLab kernel running on the node.

7.3 Topology Sensors

Our third example sensor server reports information about how the local host is connected to the Internet. The example illustrates sensors that require more complex and expensive implementations; some send and receive messages over the Internet before they can respond, and some cache the results of earlier invocations. The example also illustrates how the same raw information might be exposed through multiple sensors.

`graph` $\leftarrow$ `getgraph(resolution, scope)`: returns a map of the Internet (an adjacency list) at some resolution (possible values are `AS-Level`, `Router-Level`, and `Physical-Level`), over some scope of the Internet (possible values are `Root`, `AS`, and `Network`). Currently, only resolution `AS-Level` and scope `Root` are supported, and it is implemented using feeds from a collection of BGP routers.

`path` $\leftarrow$ `getpath(node1, node2, resolution)`: returns a path between a pair of nodes (given by IP addresses) at some resolution (same possible values as for `getgraph`). When `node1` is not the local node, the sensor forwards the query to `node1`, passes whatever values that node returns to the caller, and caches the result. When the resolution is `AS-Level`, the implementation uses a local BGP feed. When the resolution is `Router-Level`, the implementation is a wrapper for `traceroute` (with results of previous invocations cached). An implementation for resolution equal to `Physical-Level` is not currently supported.

`distance ← getdistance(node, resolution)`: returns the distance from the local machine to the specified `node` (given by IP addresses) at some resolution (possible values are the same as for `getgraph`). When the resolution is `AS-Level`, the implementation uses a local BGP feed, and returns the number of AS hops between the local machine and the specified `node`. When the resolution is `Router-Level`, the implementation is a wrapper for `traceroute` (with results of previous invocations cached for one hour), and the sensor returns the number of router hops between the local machine and the specified `node`. When the resolution is `Physical-Level`, the implementation is a wrapper for `ping` (with results of previous invocations cached for one hour), and the sensor returns the round-trip time between the local machine and the specified `node`.

`path ← iproute(node)`: returns the sequence of routers between the local machine and the specified `node` (given by an IP address). Implemented as a wrapper for `traceroute`.

`time ← iplatency(node)`: returns the round-trip time between the local machine and the specified `node` (given by an IP address). Implemented as a wrapper for `ping`.

The first three sensors export an abstract interface that overlays can use to learn about the topology of the Internet at multiple levels of resolution. These three sensors are designed to support a particular class of applications—service overlays that implement their own routing strategy—that must pay attention to the cost of probing the Internet []. Notice that sometimes these sensors depend on the underlying commands `ping` and `traceroute`, the results of which may be cached from a previous reading. In contrast, the last two sensors provide a low-level interface to these same to commands. We expect them to be used by other applications that require the most recent data possible, but probe the Internet so rarely that cost is not a concern.

8 Design Guidelines for Sensors

Gradually, we'll get some more experience writing and using sensors. At the moment, a few guidelines come to mind:

Avoid data abstraction. When building a sensor, you don't know in general what someone on the side of the planet will want to do with your data in 12 months time. Consequently, avoid "cleaning" the data in any way or imposing some abstraction or aggregation on it as much as possible - that's something clients will want to do themselves and in ways you can't predict. It also makes it easier to write the sensor if you avoid transforming the data any more than strictly necessary.

Don't implement a streaming sensor if a snapshot will do. It's harder to build a streaming sensor: it needs to handle multiple connections, and consequently has to have some way to bound this and get out of trouble when it has too many clients. While there are data sources which can only usefully be provided as a stream, many can given in snapshot form, sometimes as a circular buffer of recent history.

Don't hammer snapshot sensors. When building a client to a sensor, don't repeatedly poll it in a tight loop, and show some restraint in when to request data. Since snapshot sensors return data immediately, your query processor can wait until a query comes it before asking the sensor. If the query rate is very high, consider caching the sensor results in the client. A truly robust sensor will have some notions of rate limiting and fairness builtin, but well-written clients should not assume this.

A HTTP Header Fields for Sensor Servers and Clients

A.1 Client Request Header Fields

Table 1 lists the required and recommended client request header fields. Next, Table 2 lists fields which may appear in a client request but that are ignored by sensor servers. The use of these fields is discouraged; however, these fields may be generated by browsers (during debugging of servers) or by third-party libraries and not be under the direct control of client developer. Finally, Table 3 lists fields that should never occur in client requests. Requests that include these header fields will receive an error response.

Header Field	Use	Notes
Request-Line	required	Servers only implement the `GET` and `HEAD` methods. Servers receiving other methods will return status 405 (Method Not Allowed).
`Host`	required	Must be of the form: "`Host: 127.0.0.1:`*port*". If omitted or incorrect, servers will return status 400 (Bad Request).
`Connection`	recommended	Sensor servers do not implement persistent connections; clients should send "`Connection: close`"
`User-Agent`	recommended	Indicates client name and version.

Table 1: Required and Recommended Client Request Header Fields

`Accept`	`Content-MD5`	`Last-Modified`
`Accept-Charset`	`Content-Range`	`Max-Forwards`
`Accept-Encoding`	`Content-Type`	`Pragma`
`Accept-Language`	`Date`	`Proxy-Authorization`
`Allow`	`Expires`	`Referer`
`Authorization`	`From`	`TE`
`Cache-Control`	`If-Match`	`Trailer`
`Content-Encoding`	`If-Modified-Since`	`Transfer-Encoding`
`Content-Language`	`If-None-Match`	`Upgrade`
`Content-Length`	`If-Range`	`Via`
`Content-Location`	`If-Unmodified-Since`	`Warning`

Table 2: Client request header fields that are ignored by servers.

Header Field	Use	Notes
`Expect`	no	Servers respond with status status 417 (Expectation Failed).
`Range`	no	Servers respond with status status 416 (Requested range not satisfiable).

Table 3: Forbidden Client Request Header Fields

A.2 Server Response Header Fields

Table 4 lists the required and recommended server response header fields. Next, Table 5 lists fields that should not be sent by sensor servers and should be ignored by clients. Finally, Table 6 lists fields that should never occur in server responses. Responses that include these header fields should be rejected by the client.

Header Field	Use	Notes
Status-Line	required	
`Content-Length`	required	Only used when there is a non-chunked body.

`Content-Type`	required	Only used when there is a body. The only types a client may support are:
		`text/plain; charset=iso-8859-1`
		`text/xml; charset=utf-8`
		`application/octet-stream`

`Date`	required	
`Last-Modified`	recommended	
`Accept-Ranges`	recommended	Servers need not support ranges and should include "`Accept-Ranges: none`" in their responses.
`Connection`	recommended	Sensor servers do not implement persistent connections and should include "`Connection: close`" in their responses.
`Server`	recommended	Indicates name/version of server
`Transfer-Encoding`	*see notes*	Use "`Transfer-Encoding: chunked`" for streaming sensors.
`Content-Range`	*see notes*	Ranges are not supported. This field is only sent as "`Content-Range: *`" in response to a client request containing a `Range` field.
`Allow`	*see notes*	Required in a status 405 (Method Not Allowed) response "`Allow: GET, HEAD`"; otherwise optional

Table 4: Required and Recommended Server Response Header Fields

`Age`	`Content-MD5`	`Upgrade`
`Cache-Control`	`ETag`	`Vary`
`Content-Encoding`	`Expires`	`Via`
`Content-Language`	`Pragma`	`Warning`
`Content-Location`	`Retry-After`	

Table 5: Server response header fields that are ignored by clients.

B CSV Format for Sensor Servers and Clients

Because there seems to be no official standard for CSV (comma separated value) files, we describe the recommended format for sensor servers and clients here.

Header Field	Use	Notes
`Location`	no	Clients should ignore response
`Proxy-Authenticate`	no	Clients should ignore response
`Trailer`	no	Clients should ignore response
`WWW-Authenticate`	no	Clients should ignore response

Table 6: Forbidden Server Response Header Fields

A CSV-formatted file is a text file where each line is a record consisting of one or more fields separated by commas. Blanks lines and comments lines (indicated by a # character in the first column) are ignored.

Fields that contain commas, quotes, spaces, or control characters, must be quoted using the same format as a string in the C programming language.

Here is an example CSV file:

```
# Comment line
field1,23.45,"Field 3",42
"A field with commas, \"quotes,\" and control characters\t!",2,x,43
```

References

[1] T. Berners-Lee and D. Connolly. Hypertext Markup Language – 2.0; RFC 1866. *Internet Request for Comments*, November 1995.

[2] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax; RFC 2396. *Internet Request for Comments*, August 1998.

[3] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1; RFC 2616. *Internet Request for Comments*, June 1999.

[4] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[5] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation (OSDI '02), Boston, MA, USA.*, December 2002.

[6] M. Murata, S. S. Laurent, and D. Kohn. XML Media Types; RFC 3023. *Internet Request for Comments*, January 2001.

[7] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. Irisnet: An architecture for compute-intensive wide-area sensor network services. Technical Report IRP-TR-02-10, Intel Research Pittsburgh, December 2002.

[8] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. `http://www.cs.cornell.edu/Info/Projects/Spinglass/public_pdfs/Astrolabe.pdf`, September 2002.

[9] M. Wawrzoniak. Sophia. http://www.cs.princeton.edu/%3Emhw/sop-hia/index.php, March 2003.

Version Control Information

CVS version information:

```
$Id:  sensors.tex,v 1.2 2003/05/13 13:11:48 scott Exp $
$Id:  http.tex,v 1.2 2003/05/13 13:11:48 scott Exp $
$Id:  examples.tex,v 1.1.1.1 2003/03/28 21:09:32 scott Exp $
$Id:  fields.tex,v 1.1.1.1 2003/03/28 21:09:32 scott Exp $
$Id:  csv.tex,v 1.1 2003/05/13 13:11:48 scott Exp $
```

This paper was run off May 13, 2003.