

IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

D11.2 DETAILED ARCHITECTURE SPECIFICATION

Author(s):	Zoltán Balaton, Gábor Gombás
Document Filename:	GridLab-11-D11.2-01-Architecture
Work package:	WP11
Partner(s):	GridWare, MU, SZTAKI, VU
Lead Partner:	SZTAKI
Config ID:	GridLab-11-D11.2-01-v1.2
Document classification:	IST

Abstract: In this document the architecture of the monitoring system in the GridLab project is presented. The document gives a detailed description of the monitoring system architecture and some implementation details of the composite components.



Contents

1	Introduction	2
2	Abstract Structure of the Monitoring System	2
3	Metrics	2
3.1	Metric Definition	3
3.2	Metric Instance	4
4	Components of the Monitoring System	4
4.1	Local Monitor	5
4.2	Main Monitor	6
4.3	Monitoring Service	6
4.4	Other Related Components	6
5	Consumer–Producer protocol	7
5.1	Channels	7
5.2	Commands	8
5.2.1	AUTH	8
5.2.2	COLLECT	8
5.2.3	STOP	9
5.2.4	SUBSCRIBE	9
5.2.5	BUFFER	9
5.2.6	GET	9
5.2.7	DEF_CHANNEL	10
5.2.8	SET_CHANNEL	10
5.2.9	GET_SPECIAL	10
5.3	Data encoding	10
	References	11
A	Example Metrics	12

1 Introduction

The Grid is a complex system and therefore monitoring is essential for understanding its operation, debugging, failure detection and for performance optimisation. In the Requirements Analysis Report [1] the usage scenarios and requirements of a monitoring system for the Grid were discussed. In this document we present the detailed architecture of the monitoring system planned for the GridLab project. It is based on the general Grid Monitoring Architecture [5] proposed by the Global Grid Forum. This document gives a detailed description of the monitoring system and some implementation details of the composite components.

2 Abstract Structure of the Monitoring System

At the lowest level it is the task of Sensors to measure properties of physical entities (see Figure 1). Sensors output representations of measured physical quantities that are input to the monitoring system. The output of the monitoring system (at its user interface) are representations of physical quantities transformed to the appropriate form according to the user's requirements. This transformation also includes deriving higher level representations from the low-level data measured by sensors. Thus, the monitoring system can be regarded as a transformation layer between the physical and the logical (user or application) layers.

Physical resources are collected into groups: they form so called grid resources. For the user, the conglomeration of resources is a single entity: the Grid. The monitoring system however has to handle each resource individually. Consequently, the monitoring system consists of several monitors and the union of which provides information about the whole system. Thus, apart from the vertical layering discussed above, the monitoring system is also distributed horizontally across the available resources. This layering and distribution of the monitoring system is shown in Figure 1.

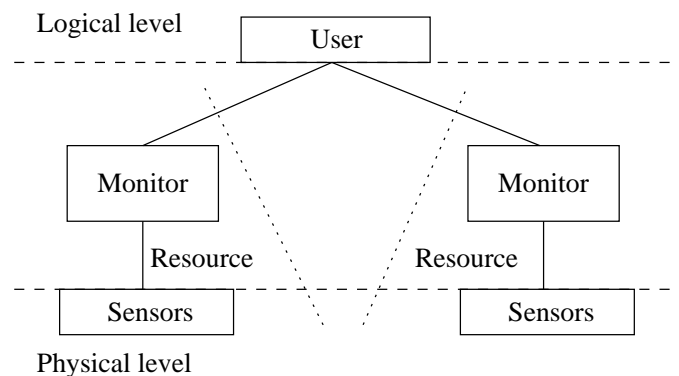


Figure 1: Abstract Structure of the Monitoring System

3 Metrics

Abstract representations of measurable quantities in the monitoring system are called metrics. There are two classes of metrics:

- **Local metrics** are those that are directly measured on a resource. These can be highly dependent on the physical parameters of the resource. Thus, local metrics originating from two different resources are not necessarily comparable. (E.g. 1 hour CPU time on a 1 GHz Intel processor is different than 1 hour CPU time on an 800 MHz PPC processor although the numeric values are

equal.) However resource administrators who know the configuration of the resource need local metrics for detailed monitoring of the status and operation of the resource.

- **Grid metrics** on the other hand have predefined semantics, therefore they are resource independent. Grid metrics are derived from one or more local metrics by applying a specific, well defined algorithm (such as unit conversion, aggregation or averaging). Because of this transformation, grid metrics may have less precision or they could be less specific but are guaranteed to be comparable between different resources. Unlike local metrics which a local resource is free to change, grid metrics must be agreed upon, standardised and introduced by community consensus.

Metrics have a definition (e.g. CPU load) and instances (e.g. CPU load measurement on a specific machine). In the Grid, there are a huge number of metrics that are possible to measure but a user is only interested in a subset of these at a time. The monitoring system allows the selection between possible metrics by using a unique name that identifies a metric definition and metric identifiers that identify metric instances.

3.1 Metric Definition

A metric definition contains the following properties:

- Metric name
- Parameters
- Measurement type
- Data type
- Unit

The **Metric name** is used to identify the metric definition (e.g. CPU usage). It consists of dot separated words, e.g. `host.cpu.user`. The last component of a metric name is the actual name of the metric, the preceding components are called scope. The scope can be used to group metrics as well as to differentiate between similar metrics defined at different levels (for example, CPU utilisation can be measured on a per-job or per-host level).

The **Parameters** field in the metric definition contains the formal definition of the metric parameters. Many metrics can be measured at different places simultaneously. For example, CPU utilisation can be measured on several hosts or grid resources. The metric parameters can be used to distinguish between these different metric instances.

The **Measurement type** can be *continuous*, meaning data is always available, or *event-like*, meaning data only becomes available when some external event happens (e.g. a sensor embedded in an application can send events at any time). Continuous metrics are only available using *pull model delivery* unless the user specifies a measurement frequency i.e. without a specified measurement frequency, there is no event triggering the measurement of a continuous metric. If the user asks for periodic measurements by specifying a measurement frequency the monitoring system will generate periodic events automatically. This avoids polling and allows the monitoring system to use the measurement frequency information to optimise measurements.

The **Data type** is the definition of structure used for representing measurement data. The monitoring system supports the following basic types:

- INT32: 32 bit, signed integer
- INT64: 64 bit, signed integer

- UIN32: 32 bit, unsigned integer
- UIN64: 64 bit, unsigned integer
- DOUBLE: Double precision floating point number
- STRING: Text in Unicode UTF-8 encoding, allowed length must be at least 1024 bytes
- OPAQUE: Arbitrary binary data uninterpreted by the monitoring system, allowed length must be at least 1024 bytes

Note: The main difference between the *STRING* and *OPAQUE* types is that a general purpose client (such as a visualisation tool) could display *STRING* type data but can ignore *OPAQUE* type data.

For more complex metrics the monitoring system also supports the following complex types:

- ARRAY: Array of identical types with fixed or variable size
- RECORD: A composite of other types with a predefined layout

The above types allow the definition of arbitrarily complex data types, however monitoring data is typically represented with simple types. Moreover writing general clients and visualisation tools is only possible if these can parse and handle measurements, which can be difficult if very complex data types are used. Consequently, the use of simple types is preferred.

The **Unit** specifies the physical unit in which the metric is measured and is represented as a text string. It is only valid for simple numeric types and arrays of these types. In the case of arrays, it means the unit of each element in the array. For *STRING*, *OPAQUE* and *RECORD* types the meaning of the *Unit* field is undefined.

3.2 Metric Instance

Substituting concrete values in the formal parameters of a metric definition yields a metric instance. The notion of metric instances is necessary because the same metric could be measured at different places within one resource at the same time (e.g. on different hosts within a cluster). Metric instances are used to differentiate between these measurements. For example, instances of a metric describing the available memory on a host are distinguished by a parameter containing the hostname (`host.mem.avail(host.gridlab.org)`). Another example is if an application is running on more than one host and the hostname is not included in the parameters of a metric describing some properties of the application then consumers cannot differentiate between measurements arriving from different hosts.

A concrete measurement corresponding to a metric instance is called metric value. Metric values contain a timestamp and the measured data according to the data type of the metric definition.

4 Components of the Monitoring System

The architecture of the monitoring system planned for the GridLab project follows the Consumer-Producer model of the general Grid Monitoring Architecture [5] proposed by the Global Grid Forum. Figure 2 shows the components of the monitoring system.

The figure depicts the situation where an application started as a grid job is running on a grid resource. The running application consists of processes (labelled P in the figure) running on hosts constituting the grid resource. Processes are identified by process identifiers (PIDs). The monitoring system is indicated by components drawn with solid lines. These components (LM, MM and MS) will be provided by the Monitoring Work Package. Components drawn with dashed lines show other parts of the system the

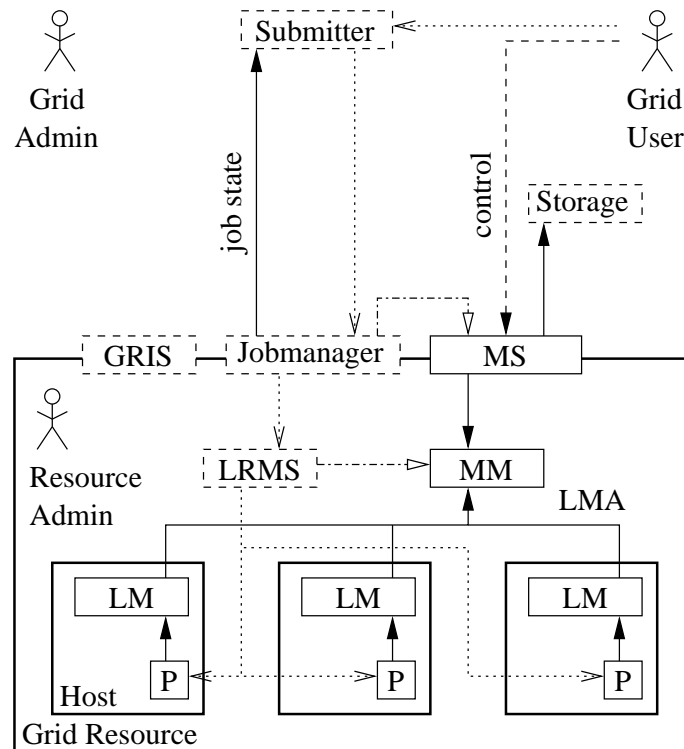


Figure 2: Components of the Monitoring System

monitoring system is connected with. As cooperation is required from these components for correct operation of the monitoring system we plan to work together with and expect help from work packages providing these components. Each component is described in detail below.

4.1 Local Monitor

Every host being monitored must run an instance of a local monitor (LM). The task of the local monitor is to collect monitoring information originating at the particular host. This includes information about the host itself as well as information about processes running on the host. The LM sends all data up to a main monitor.

The LM also manages sensors on the host. It can start, stop and control sensors and initiate measurements on a user's request. Most sensors are loadable modules that are dynamically linked into the LM depending on configuration. This modularity makes the monitoring system flexible and makes it easy to add new sensors. Some very simple sensors could also be built into the LM.

Sensors can get measurement data from any sources they like. They do not have to take all measurements themselves, they could also contact external processes to get data. For example, a sensor can query the local accounting system if available or it is also possible to implement persistent sensors (such as network monitors) that are running independently of the monitoring system and gather statistics over time.

Applications can also provide application specific metrics to the monitoring system. For this purpose a special sensor exists which can accept monitoring data from processes. The application can link to an instrumentation library which can connect to this special sensor. The instrumentation library provides the following functionality:

- **Process identification:** The application registers itself with the monitoring system so it can be identified which application generated the metric instances emitted by this process.

- Event generation: Generate a measurement corresponding to a metric either predefined by the instrumentation library or defined by the programmer.

The functionality of the instrumentation library could also be provided via the GAT by implementing it as a GAT adaptor.

4.2 Main Monitor

The main monitor (MM) collects monitoring information from local monitors running on hosts belonging to a grid resource. It provides a central location to access the monitoring system on a grid resource. It is used by local resource administrators to get measurements of local metrics to monitor the resource and by the Monitoring Service to gather raw measurements.

The main monitor also controls local monitors running on hosts constituting the monitored resource. When a user requests monitoring of a metric, the main monitor notifies the appropriate local monitors to take measurements and then gathers the results. In large sized clusters there may be more than one main monitor to balance network load. In this case each main monitor controls and collects data from local monitors that are close to it in the network sense.

The local monitors and the main monitor together constitute the Local Monitoring Architecture (LMA) which may augment or be substituted by other monitoring solutions already existing on the resource.

4.3 Monitoring Service

The task of the monitoring service (MS) is to make monitoring information available to grid users and administrators outside of the resource. The MS allows outside users to access monitoring information of the grid resource in a similar fashion to the way the main monitor provides information about the resource for local users. The MS gathers raw measurements of local metrics from the LMA (and possibly other sources such as the information system) and converts these local metrics to grid metrics. Furthermore, the MS also handles the authentication and authorisation of grid users. The exact security features implemented are dependent on the recommendations of the Security Work Package. The MS is the GridLab service implementing the monitoring API in the GAT and therefore a corresponding GAT adaptor will be provided for this purpose. Depending on the recommendations of the Grid Application Toolkit Work Package, an OGSA service description (WSDL document) defining the interface of the monitoring service can also be provided.

It is also the task of the MS to coordinate and control the monitoring of grid jobs. Specifically, the MS sets up initial monitoring at job startup according to the requirements of the user. Thereafter, the user can control monitoring while the job is running. For this, the MS needs cooperation from both local and grid jobmanager facilities (the jobmanager and LRMS components). The MS can also talk to a user specified grid storage service to store data by a user request instead of delivering measurements directly to the user. The implementation of this feature is dependent on the API provided by the Data Handling Work Package.

4.4 Other Related Components

The monitoring system is connected to and needs cooperation from the following components:

- The local resource management system (LRMS) controls jobs running on the resource. It allocates hosts to jobs, starts and stops jobs on user request and possibly restarts jobs in case of an error. It may also checkpoint and migrate jobs between hosts inside the resource. The jobs managed by the LRMS are identified by a local job identifier (LJID). Only the LRMS knows which processes belong to a job, thus it must provide an interface for the LMA to provide the mapping between PIDs

and LJIDs. The LRMS also must cooperate with the monitoring system when starting, checkpointing and migrating jobs in order to ensure correct monitoring of these jobs.

- The submitter represents a grid entity which starts a grid job and keeps track of the job status. The submitter always gets notifications about changes in the state of the job and so it always knows which grid resources the application is running on. The user can use this information to find the monitoring service contact points at the appropriate resources to contact in order to control the monitoring of the job and get measurements.
- The jobmanager represents the grid service, which allows users to start jobs on a grid resource. The user hands to the jobmanager a document called *job manifest* that contains the specification of the requested local resources and the description of the job. After successful authentication and authorization, the jobmanager translates the job manifest into a form understood by the local resource management system and then starts the job. The jobmanager also allows the user to control the execution of the job. A grid job identifier (GJID) is used to reference jobs started by the jobmanager. The jobmanager maintains the mapping between GJIDs and LJIDs and must provide an interface for the monitoring service to get this information. The user may specify initial monitoring requirements in the job manifest and therefore the jobmanager must also pass it to the MS.
- The grid resource information service (GRIS, part of the Grid Information System) can provide information about the monitoring service (such as available metrics, metric definitions, contact point of the monitoring system) although this information is also available from the monitoring system directly. Moreover, the monitoring and information systems can use the same sensors in some cases. This can be achieved either by the monitoring system obtaining information from the information system or vice versa. The Monitoring Work Package is open to work with the Information System Work Package in this area.

5 Consumer–Producer protocol

In the GGF GMA terminology, the monitoring service is a producer, the user accessing monitoring information is a consumer. The protocol described in this section is used for communication between these parties.

There are 3 main protocol data units: commands, command responses and metric values. Every command has an identifier (e.g. COLLECT or GET) and a sequence number. Responses have an error code, the same sequence number as the original command and an optional result identifier. It is possible to issue multiple commands without waiting for a response for the previous command to arrive. In case of multiple parallel commands the protocol does not guarantee that the order of responses matches the order of the original commands.

The producer is allowed to send metric values at any time. The only restriction is that if a metric value is being sent as a result of a command, the response for the command must precede the metric value (i.e. when using the GET command to request data the response for the GET command must be sent before the requested data corresponding to this GET command).

5.1 Channels

A channel is a logical connection between a producer and a consumer defined over a physical network connection. Each channel has a channel identifier which is assigned by, and specific to, the producer. There are two special channel identifiers a consumer can use. One references all channels defined between it and the producer it is talking to, the other references the current channel. When a network

connection is opened between a producer and a consumer, the producer automatically defines a channel and makes it the current channel. The consumer can later define new channels and change the current channel.

There are two kinds of channels:

- Consumer initiated: where the actual network connection corresponding to the channel is initiated by the consumer.
- Producer initiated: where the actual network connection corresponding to the channel is initiated by the producer. The parameters of the network connection and appropriate authorization credentials have to be defined by the client in advance.

Both kinds of channels are necessary to both support the delivery of monitoring data to passive services (such as a storage service) and to traverse firewalls.

Other (possibly later implemented) properties of channels are:

- Persistent: the channel will survive if the network connection is broken or the consumer and/or the producer is restarted.
- Delayed open (producer initiated only): the network connection is opened only if there are data to be sent.
- Close if idle (producer initiated only): when there are no more data to send the network connection is closed and will be reopened when necessary.
- High throughput: data will be sent in large packets.

5.2 Commands

When a consumer contacts a producer, the producer first sends a string identifying the protocol version. The consumer then can use the following commands:

5.2.1 AUTH

- Arguments: *To be defined later*
- Response: error code, channel identifier

The AUTH command authenticates the user to the monitoring system. The AUTH command should be used once and it should be the first command issued by the consumer after a channel is opened. Any other commands issued before the AUTH command will result in an error. Issuing the AUTH command on an already authenticated channel will also result in an error. If the authentication was successful an identifier for the current channel will be returned.

5.2.2 COLLECT

- Arguments: metric name, parameter list
- Response: error code, metric identifier

The COLLECT command instructs the monitoring system to create a metric instance with the given parameters. If this is successful, the response contains the metric identifier of the metric instance. The parameter list is a list of name/value pairs.

5.2.3 STOP

- Arguments: metric identifier, channel identifier
- Response: error code

The `STOP` command tells the monitoring system that no more metric values for this identifier should be sent to the specified channel. Metric values for this identifier that have been queued in the monitoring system for the specified channel will be lost.

If the metric identifier is still subscribed to other channels, both the `GET` and `SUBSCRIBE` commands can be used to receive further metric values for this metric id. If there are no other channels where this metric identifier is subscribed, the `STOP` command will destroy the metric instance and all further references to this metric instance will result in an error.

5.2.4 SUBSCRIBE

- Arguments: metric identifier, channel identifier
- Response: error code

The `SUBSCRIBE` command instructs the monitoring system that all metric values for the given metric identifier should be automatically sent to the specified channel. The same metric identifier can be subscribed to more than one channel by using the `SUBSCRIBE` command multiple times. If there were metric values queued for the given metric identifier on the current channel they will be copied to the destination channel.

5.2.5 BUFFER

- Arguments: metric identifier, channel identifier
- Response: error code

The `BUFFER` command instructs the monitoring system that all metric values for the given metric identifier should be buffered on the specified channel. It is possible to use the `SUBSCRIBE` and `BUFFER` commands for the same metric identifier on different channels.

5.2.6 GET

- Arguments: metric identifier, channel identifier
- Response: error code

The `GET` command performs the following tasks:

- If there are metric values queued for the given metric identifier on the given channel, the monitoring system will send them to the consumer.
- For continuous metrics, a new measurement is requested from the appropriate sensor. When the measurement is ready, the measured metric value will be sent to the consumer on the given channel. The monitoring system makes no guarantee when (if ever) this metric value will be sent.

5.2.7 DEF_CHANNEL

- Arguments: target URL, parameter list
- Response: error code, channel identifier

The DEF_CHANNEL command defines a producer initiated channel that the monitoring system should open to the specified external location. The parameter list is a list of name/value pairs the actual meaning of which is dependent on the protocol of the target URL.

5.2.8 SET_CHANNEL

- Arguments: channel identifier
- Response: error code

The SET_CHANNEL command changes the current channel to the one identified by the given channel identifier while keeping the network connection. If the destination channel had an open network connection it will be closed.

5.2.9 GET_SPECIAL

- Arguments: metric identifier, parameter
- Response: error code

The GET_SPECIAL command is similar to the GET command and is used to request specific internal data from the monitoring system (e.g. metric definitions). The metric identifier should be a special pre-defined metric identifier, it cannot be an identifier returned by the COLLECT command. The parameter is either a metric or a channel identifier depending on the first argument. This command is meant for internal communication between the consumer API and the producer and may change in later protocol versions.

5.3 Data encoding

Measurement data transmitted by the protocol must be encoded in a way that is platform independent and preserves its structure described in the data type field of the metric definition. There are several standard data encodings which satisfy these requirements, the most well known are:

- **XML:** The Extensible Markup Language [2] is a World Wide Web Consortium (W3C) standard that allows the representation of structured data in a verbose textual form.
- **ASN.1:** The Abstract Syntax Notation One [3] is an international standard (ISO/IEC 8824-1:1998) that allows the definition of data types and values. A related standard (ISO/IEC 8825-1:1998) defines the Basic Encoding Rules (BER) which can be applied to values of data types defined using ASN.1.
- **XDR:** The External Data Representation [4] is an Internet standard for the description and encoding of the most commonly used data types in a concise way.

An important requirement for the monitoring system is that it should influence the system being monitored as little as possible and be able to handle a large volume of monitoring data. Thus, the data encoding chosen should be efficient and the resource requirement of its interpretation should be low. Both XML and ASN.1 are very general, extensible languages that are able to describe and represent very complex data types. Because of this generality and extensibility interpretation of these languages have a relatively high resource requirement. Monitoring data however consists of simple datatypes in most cases which can be described by XDR well. The resource requirement of XDR is also a fraction of what is needed for more general languages. Because of this, the data encoding used in the monitoring system is similar to XDR, but only a subset of XDR is needed to represent metrics. Only this subset is implemented.

References

- [1] Zoltán Balaton, Gábor Gombás: Requirements Analysis Report
Technical Report, GridLab Project, March 2002.
<http://www.gridlab.org/Project/Deliverables.html>
- [2] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen and Eve Maler (editors):
Extensible Markup Language (XML) 1.0 (Second Edition)
W3C Recommendation, World Wide Web Consortium, October 2000.
<http://www.w3c.org/TR/2000/REC-xml-20001006>
- [3] ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998:
Information Technology – Abstract Syntax Notation One (ASN.1): Specification Of Basic Notation
- [4] Raj Srinivasan: XDR: External Data Representation Standard
IETF RFC 1832, August 1995.
<http://www.ietf.org/rfc/rfc1832.txt>
- [5] Brian Tierney et al.: A Grid Monitoring Architecture
Technical Report, Global Grid Forum, GMA-WG, March 2000.
<http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf>

A Example Metrics

This is the list of metrics provided by the currently implemented example sensors. This list should not be considered final (or even as a recommended set of metrics). It is only useful for testing the monitoring system and to demonstrate possible types of data sources and metrics. Other metrics corresponding to the GridLab requirements and to the recommendations of the Global Grid Forum DAMED and NMWG working groups can be implemented when their specification is available.

First an example of a simple metric:

- `host.mem.avail`: Available physical memory
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: kilobyte

The following metric is static configuration information not measured by a sensor but such configuration information could also be provided by the monitoring system at the convenience of consumers.

- `host.mem.size`: Size of physical memory
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: kilobyte

The following metric also looks like configuration information but, with the current development of Linux CPU hotplug support, it will change soon. When combined with Intel Hyperthreading technology this metric might be quite useful (there are workloads where hyperthreading improves performance and there are some workloads where it decreases performance so dynamically turning it on and off – and thereby changing the number of visible CPUs – might be useful).

- `host.cpu.avail`: Number of available CPUs
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: number of items

Some other simple metrics:

- `host.swap.avail`: Available swap space
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: kilobyte
- `host.swap.size`: Size of swap space
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: kilobyte

- `host.processes.all`: Number of all existing processes
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: number of items
- `host.processes.running`: Number of currently running processes
Parameters: hostname
Measurement type: continuous
Data type: `uint32`;
Unit: number of items

The classic example metric (despite it is not very meaningful without other knowledge about the system where it is measured) demonstrates a metric with a fixed size array data type:

- `host.loadavg`: Current load average
Parameters: hostname
Measurement type: continuous
Data type: `double [3]`;

The following two metrics demonstrate how information about multiple objects can be combined into a single metric value using an array with an undefined size:

- `host.disk.size`: Total size of mounted file systems
Parameters: hostname
Measurement type: continuous
Data type: `rec { string device; uint64 size; } []`;
- `host.disk.free`: Free space on mounted file systems
Parameters: hostname
Measurement type: continuous
Data type: `rec { string device; uint64 size; } []`;

The next one is an example of a local metric directly exposing raw data provided by the operating system. This allows the sensor code to be very simple but in practice consumers might prefer data in some more “cooked” form.

- `host.disk.io`: Disk I/O since system boot
Parameters: hostname
Measurement type: continuous
Data type: `rec { string device; uint64 rio; uint64 rblk; uint64 wio; uint64 wblk; } []`;

The following metrics do the contrary to `host.disk.io`. Instead of exposing the raw data (seconds of cpu time since system boot spent in user/nice/system/idle mode) they expose preprocessed values:

- `host.cpu.user`: CPU time used by user processes in the last 5 seconds
Parameters: hostname, CPU ID
Measurement type: continuous
Data type: `uint32`;
Unit: per thousand

- `host.cpu.nice`: CPU time used by niced processes in the last 5 seconds
Parameters: hostname, CPU ID
Measurement type: continuous
Data type: `uint32`;
Unit: per thousand
- `host.cpu.system`: CPU time used by the OS in the last 5 seconds
Parameters: hostname, CPU ID
Measurement type: continuous
Data type: `uint32`;
Unit: per thousand
- `host.cpu.idle`: Idle CPU time in the last 5 seconds
Parameters: hostname, CPU ID
Measurement type: continuous
Data type: `uint32`;
Unit: per thousand

An example of a composite metric that unifies the previous four:

- `host.cpu.all`: Information about CPU usage in the last 5 seconds
Parameters: hostname, CPU ID
Measurement type: continuous
Data type: `rec { uint32 user; uint32 nice; uint32 system; uint32 idle; }`