Linux Metric Data Gatherer Version 2

Design and Implementation

Document Info

Owner	Viktor Mihajlovski (VM) <mihajlov@de.ibm.com>
Author	Viktor Mihajlovski
Version	2.0, Sep 9, 2004
Filename	/home/mihajlov/doc/gatherer/GathererDesignV2.sxw

Change Log

Date	***Author***	***Comment***
Aug 5, 2004	VM	Started based on Gatherer Version 1 Document
Sep 9, 2004	VM	Updated to reflect V2 content

Bugs and Todos

This document needs to be overhauled thoroughly

It's incomplete and inconsistent.

Add reference materials.

Fill glossary.

Table of Contents

1 Purpose of this document

This document is to document the architecture for the Linux Metric Data Gathering Infrastructure. The design points are:

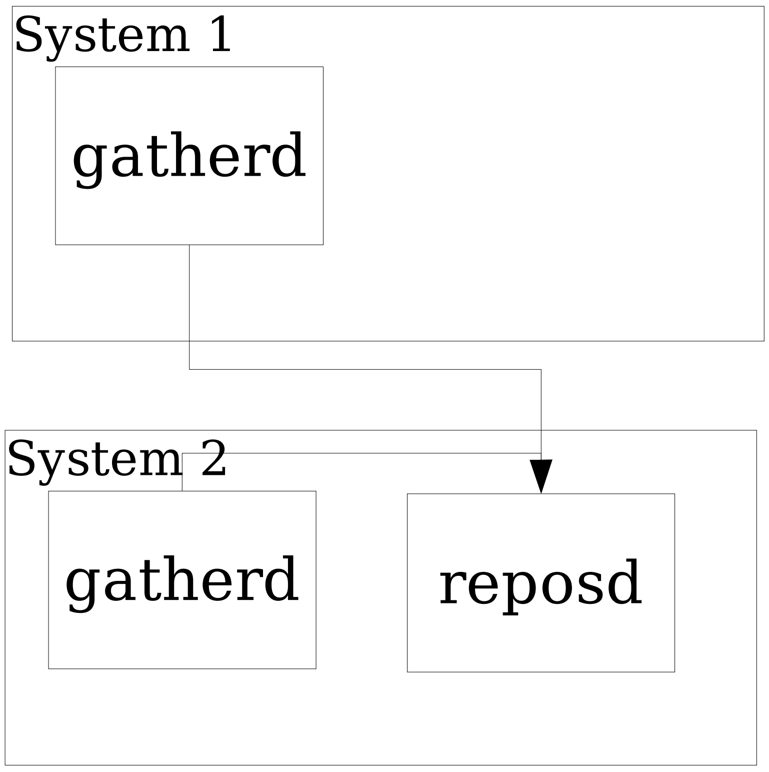
- Suitable to implement the CIM Monitoring model
- Interoperable/compatible with current RMF/Linux offering.
- CIMOM-friendly (efficient in conjunction with CIM architectures)
- Efficient and Scalable (especially in virtualized environments).

1.1 General Notes

The program package is called Gatherer throughout this document.

1.2 Distributed Gatherer

Starting with Version 2 the Gatherer can be used in a distributed fashion. For this purpose it's functionality is provided by two daemons : a Gathering Daemon (gatherd) and a Repository Daemon (reposd) for centralized data collection.



1.3 Gatherer Plugins

In order to be flexible regarding the usage of the Gatherer a highly modular structure is one goal. There are multiple plugin types allowing to configure the functionality at run time, specifically:

- Metric Retrieval Plugins for the Gathering Daemon.
- Metric Data Processing Plugins for the Repository Daemon.

The Gatherer defines the interfaces the plugins must adhere to and implements services that can be used by the plugins.

2 Gatherer Structure and Interfaces

2.1 Gathering Daemon

The central data structure used by the Gathering Daemon is the Metric List, which is a collection of Metric Blocks containing administrative information about the metrics to be gathered, like the id's, sample intervals, etc.

The Gathering Daemon uses an scheduling engine called Metric Scheduler to process the Metric List, respectively the Metric Blocks according to their scheduled sample intervals. Each Metric Block defines a callback function that will do the metric retrieval. This callback function will be called by the Metric Scheduler in order to retrieve metric data and forward the results to the Repository Daemon using the

Repository interface.

Alternatively (to interval sampling) the Scheduler can be woken up externally in order to process the metric retrieval.

The Metric Blocks do not directly interface to Metric Definitions as provided by a Plugin but use a metric id which is registered and maintained by the Metric Plugin Manager which is responsible for the loading and unloading of Metric Plugins.

Metric Retrieval Plugins are implemented by shared libraries which must export Metric Definitions and provide the metric retrieval function for each defined metric.

2.2 Repository Daemon

The Repository Daemon is the central entity for the data collection and processing. It is maintaining a repository of metric values that can be interrogated by clients. The Repository Daemon is receiving the Metric Values from one or more Gathering Daemons.

2.3 Metric Retrieval Plugin Interfaces

A Metric Retrieval Plugin must provide one or more Metric Definitions and associated Metric Retriever functions.

2.3.1 Plugin Module Entry Points

A plugin is packaged as a shared library that must define two entry points: _DefinedMetrics and _StartStopMetrics. The first one is called by the Plugin Manager to obtain a list of Metric Definitions. The latter is called to inform the plugin that sampling is about to begin (or end).

2.3.2 Metric Definitions for Retrieval

A Metric Definition is a C structure (type MetricDefinition) that is provided by the metric plugin and contains the following fields:

Name	***Type (C)***	***Comment***
mdVersion	short	Version of the Gatherer package that is required to run this plugin.
mdName	char*	Metric name defined by the plugin. Must be unique within the context of the plugin.
mdReposPluginName	char*	Name of the repository-side plugin that is needed to process the metric data.

Name	***Type (C)***	***Comment***
mdId	int	Metric id. Used to uniquely identify metric types within the various Gatherer components. Note: this identifier is assigned by the metric registry component.
mdSampleInterval	time_t	Sample interval for metric data retrieval.
mproc	MetricRetriever	Metric data retrieval callback, called by the scheduler to sample metric data.
mdeal	MetricDeallocator	Metric data deallocation callback, Called to free up storage allocated by the retriever.

2.3.3 Metric Retriever

A Metric Retriever is a plugin-provided callback function used to generate Metric Values for a given Metric Definition. It is invoked with a metric id and a repository callback (MetricReturner) used to store one or more metric values (one per resource). The C signature of MetricRetriever is

```
int (MetricRetriever) (int mid, MetricReturner mret);
```

2.3.4 Metric Deallocator

A Metric Deallocator is called when metric data is being disposed of, i.e. After transferring it to the repository. The plugin must provide the deallocator. The C signature of MetricDeallocator is

```
void (MetricDeallocator) (void *v);
```

2.4 Repository Plugin Interfaces

A Repository Plugin must provide one or more Metric Definitions and associated Metric Calculation functions.

2.4.1 Plugin Module Entry Points

A plugin is packaged as a shared library that must define one entry point: _DefinedRepositoryMetrics. It is called by the Plugin Manager to obtain a list of Metric Calculation Definitions.

2.4.2 Metric Definitions for Repository Plugin

A Metric Calculation Definition is the repository-side representation of a metric. It is required by the Repository Daemon in order to accept and process requests for a particular metric.

Name	***Type (C)***	***Comment***
mcVersion	short	Version of the Gatherer package that is required to run this plugin.
mcName	char*	Metric name defined by the plugin. Must be unique within the context of the plugin and identical to the corresponding.
mcId	int	Metric id. Used to uniquely identify metric types within the various repository components. Note: this identifier is assigned by the metric registry component.
mcMetricType	enum	Type of metric: currently supported are MD_RETRIEVED and MD_CALCULATED.
mcAliasId	int	For MD_CALCULATED metrics this is the id of the original MD_RETRIEVED metric this metric is based on.
mcDataType	unsigned	Data type of metric value.
mcalc	MetricCalculator	Metric computation callback. Computes the "user" metric value from raw data in the repository.

2.4.3 Metric Calculator

The Metric Calculator function is called whenever (a) metric value(s) is/are to be retrieved from the repository. This function is receiving a pointer to the original MD_RETRIEVED value as input parameter, allowing it to access the raw retrieved data needed for the computation. The C signature of MetricCalculator is

```
size_t (MetricCalculator) (MetricValue *mv, int mnum, void *v,
                           size_t vlen);
```

2.5 Common Interfaces

2.5.1 Metric Values

Metric Values are the C structures that are being filled by a Metric Retriever and transferred to a repository using the MetricReturner callback. The C type MetricValue is common between the Gathering Daemon and the Repository Daemon. A Metric Value structure contains the following fields:

Name	***Type (C)***	***Comment***
mvId	int	Metric id corresponding to the id of the metric definition.
mvTimeStamp	time_t	Timestamp of data retrieval
mvResource	char*	String identifying the resource this metric applies to.
mvDataType	unsigned	The data type of this metric value. **Note**: Although this is a duplication of information it is considered useful to have the data type attached to the value for efficient processing.
mvDataLength	size_t	Length of raw metric data.
mvData	opaque (char*)	Raw metric data. Needs to be processed before it can be used.

2.6 External Interfaces

The Gatherer can either be run as a standalone program or as a (number of) threads within another program. In both cases it is necessary to offer some control and data access facilities for the single modules. We will describe the interfaces coming with the standalone Gatherer daemon (gatherd). Note that these interfaces are not called directly but through the Remote Interfaces described later in this document.

2.6.1 Daemon Control Interfaces

gather_init

This function performs the initialization of the Gathering daemon. This must be performed before any other call is attempted.

gather_terminate

This function terminates the processing of the Gathering daemon. All the components of the Gatherer are stopped and the plugins are unloaded. Upon completion of this call no other function than gather_init can be invoked.

repos_init

This function performs the initialization of the Repository daemon. This must be performed before any other call is attempted.

repos_terminate

This function terminates the processing of the Repository daemon. All the components of the Gatherer are stopped, the plugins are unloaded and the local repository is cleared. Upon completion of this call no other function than repos_init can be invoked.

The C signatures of the daemon control functions are

```
int gather_init();
int gather_terminate();
int repos_init();
int repos_terminate();
```

2.6.2 Metric Scheduler External Interfaces

The Metric Scheduler is operating on the Metric List which is containing all the Metrics scheduled for retrieval. The functions are:

gather_start

This functions starts the scheduling process.

gather_stop

This functions stops the scheduling process.

Starting and Stopping the scheduling can be useful in virtualization environments where we don't want metric retrieval scheduling to take place while the virtual computer system is dormant, i.e., not performing any useful tasks.

Note: Metrics can be added and removed at any time without stopping the scheduler. The adding and removal of metrics is done implicitly through the Plugin Manager Load/Unload Functions.

gather_status

Returns status information about the Gatherer. Currently the daemon and scheduler state are reported as well as the number of loaded plugins and registered metrics. This information is conveyed in a structure called GatherStatus.

The C signatures of the scheduler functions are

```
int gather_start();
int gather_stop();
void gather_status(GatherStatus *gs);
```

2.6.3 Plugin Manager External Interfaces

The plugin manager is responsible for the loading and unloading of Metric Plugins. It is also maintaining a registry of the Metric Definitions provided by the plugins. Since they are similar for the two plugin types, they are listed together. The functions are:

metricplugin_add, reposplugin_add

This function loads a plugin module. The contained Metric Definitions are registered and corresponding Metric Blocks are added to the Scheduler.

metricplugin_remove, reposplugin_remove

This function unloads a plugin module *after* removing the associated Metric Blocks from the Scheduler and deregistering the Metric Definitions.

metricplugin_list, reposplugin_list

This function lists all metric definitions provided by the given loaded plugin and the resources they apply to. This information is stored in an array of PluginDefinition or RepositoryPluginDefinition structures.

The C structure PluginDefinition contains the following fields:

Name	***Type (C)***	***Comment***
pdId	int	Metric Definition Id
pdName	char *	Metric Name
pdResource	char **	Array of resource names this metric definition applies to. The array is delimited by a null pointer.

The C structure RepositoryPluginDefinition contains the following fields:

Name	***Type (C)***	***Comment***
rdId	int	Metric Definition Id
rdDataType	unsigned	Metric Data Type
rdName	char *	Metric Name
rdResource	char **	Array of resource names this metric definition applies to. The array is delimited by a null pointer.

The C signatures of the plugin manager functions are

```
int metricplugin_add(const char *pluginname);
int metricplugin_remove(const char *pluginname);
int metricplugin_list(const char *pluginname,
                           PluginDefinition **pdef,
                           COMMHEAP ch);


int reposplugin_add(const char *pluginname);
int reposplugin_remove(const char *pluginname);
int reposplugin_list(const char *pluginname,
                           PluginDefinition **pdef,
                           COMMHEAP ch);
```

2.6.4 Repository External Interfaces

The Repository can be queried for actual metric values. A metric value is identified by the metric name, the resource name, the timestamp and an interval. Currently (version 1) only one active repository is supported which must be accessed through the Gatherer module. For this reason there are no repository management interfaces defined yet.

reposvalue_get

This function receives as parameter a ValueRequest structure with a combination of the keys listed above. Depending on the keys specified in this structure metricvalue_get will return a list of metric values. The structure contains the following fields:

Name	***Type (C)***	***Comment***
vsId	int	Input: The metric definition id for the request. **Required.**
vsResource	char *	Input: The resource name for the request. If empty (NULL), metric values for all resources are wanted.
vsFrom	time_t	Input: The beginning of the interval. If <= 0 the most current values are wanted.
vsTo	time_t	Input: The end of the interval. If <= 0 the default interval length is used.
vsDataType	unsigned	Output: The returned value items data type.
vsNumValues	int	Output: The number of returned value items.
vsValues	ValueItem *	Output: Array of ValueItem structures.

If all keys are specified, the (one) matching value will be returned, with the correct data type processing (calculation) applied. If the *from* timestamp isn't specified, then the value with the most current timestamp will be returned.

If the resource name is omitted metric values for all resources are returned, with the correct data type processing (calculation) applied. If the timestamp/interval isn't specified, then the value with the most current timestamp will be returned.

Note on timestamps: for the integrated, history-less Repository there will be only a limited interval available.

The actual metric values are returned in C structures of the type ValueItem which represent a “cooked” form of the MetricValues originally sampled. The fields of ValueItem are as follows:

Name	***Type (C)***	***Comment***
viCaptureTime	time_t	Timestamp when data was captured for point metrics, beginning of interval for interval metrics.
viDuration	time_t	Length of interval for this metric value.
viValueLen	size_t	Length of value data.
viValue	Opaque (char *)	Calculated (cooked) but otherwise unformatted value data.
viResource	char *	Resource name this value applies to.

3 Communication Interface

In theory, the Gatherer can be configured to either run in-process (as pthreads), out-of-process on the same machine, or remotely. Depending on the way it is configured, the capabilities may vary.

3.1 The Core Interface

The core interface is always present regardless of the way the communication is configured. It is basically a collection of the external interfaces mentioned in chapter 2.6 on page 6. The core interface functions are all defined in the C header file “gather.h” and “repos.h”.

3.2 Out-of-Process Interface

The out-of-process interface is a remote interface that is to be used by “client applications”, meaning exploiters of the Gatherer. Although technically possible, this remote interface doesn't support remote access crossing computer system boundaries. Hence the name out-of-process.

The function names are similar to the ones in the core interface, only that the are prefixed by the letter “r”. For completeness reason here's a list of the C function declarations:

```
int rgather_init();
int rgather_start();
int rgather_stop();
int rgather_terminate();

int rmetricplugin_add(const char *pluginname);
int rmetricplugin_remove(const char *pluginname);
int rmetricplugin_list(const char *pluginname,
                    PluginDefinition **pdef,
                    COMMHEAP ch);
```

```
int rgather_status(GatherStatus *gs);

int rrepos_init();
int rrepos_terminate();

int rreposplugin_add(const char *pluginname);
int rreposplugin_remove(const char *pluginname);
int rreposplugin_list(const char *pluginname,
                     RepositoryPluginDefinition **rdef,
                     COMMHEAP ch);

int rreposvalue_get(ValueRequest *vr, COMMHEAP ch);
```

There are additional functions used to start and stop the Gathering and Repository module as standalone processes.

rgather_load

This function loads and executes the Gatherering daemon binary, preparing it for further commands.

rgather_unload

This function ends the Gatherering daemon process.

rrepos_load

This function loads and executes the Repository daemon binary, preparing it for further commands.

rrepos_unload

This function ends the Repository daemon process.

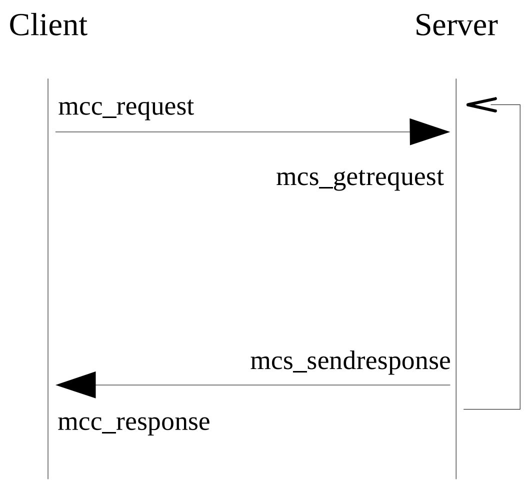
3.3 Implementation Notes

The information in this chapter is supplied for completeness purposes only. It is irrelevant for functionality of the Gatherer.

3.3.1 Communication Low Level Interface and Implementation

The Out-of-Process interface is implemented using a simple proprietary protocol over Unix Domain Stream Sockets (which has to change of course). The interface itself however doesn't imply the type of protocol that is underlying and could be implement using a variety of communication protocols including RPC, message queues, etc.

The communication interface is featuring a request response paradigm as illustrated below.



mcc_request

The client calls mcc_request in order to send an operation request to the server. It must supply a communication handle (obtained by a call to mcc_init()), apointer to a structure of type MC_REQHDR, a pointer to a contiguous block of data and the length of the data to transfer . The C signature is

```
int mcc_request(int commhandle, MC_REQHDR *hdr, void *reqdata,
size_t reqdatalen);
```

mcc_response

Upon completion of mcc_request the client calls mcc_response in order to wait for the server's response. The same MC_REQHDR structure instance as in the mcc_request call must be used and a buffer must be provided to receive the response data. The C signature is

```
int mcc_response(MC_REQHDR *hdr, void *respdata, size_t
*respdatalen);
```

mcs_getrequest

This call is issued by the server to wait for an incoming client request. It receives the request data in a buffer provided by the caller. Similarly to the client requests an MC_REQHDR has to be made available in which the function stores connection informations. The C signature is

```
int mcs_getrequest(MC_REQHDR *hdr, void *reqdata, size_t
*reqdatalen);
```

mcs_sendresponse

With this call the server sends back the response for a client's request. The MC_REQHDR of the client's request must be supplied together with a buffer containing the response data. The C signature is

```
int mcs_sendresponse(MC_REQHDR *hdr, void *respdata, size_t
respdatalen);
```

Note: A limitation of the current communication layer is that the receiver of data has to provide a buffer large enough to hold all received data. If the size is insufficient, the function will return an error indication. As the Repository interface is to be factored out of the Gatherer, the communication interface for data transfer between those has to be defined.

3.3.2 COMMHEAP Memory Management

Certain higher level communication functions must allocate non-contiguous memory areas during their execution that must be released after they have completed. Prominent examples are metricplugin_list and metricvalue_get. For convenience reasons the COMMHEAP functions can be used to allocate memory blocks that can be released in one single operation.

ch_init

This function creates a new COMMHEAP for use in subsequent calls.

ch_release

This function releases a COMMHEAP, meaning effectively that all memory blocks allocated for this COMMHEAP are freed.

ch_alloc

With this function a memory block can be allocated under COMMHEAP control.

4 Metric Plugins Currently Available

Note: The list of metrics is not complete right now.

The following metrics for Linux are currently included with the Gatherer.

The metric plugins will be grouped by resource type. We distinguish between regular and auxiliary metrics, where auxiliary metrics are not to be used by client applications. Instead, they serve as source for the regular metrics which are being computed from the auxiliary ones.

For each CIM resource class there is a BaseMetricDefinition class and a BaseMetricValue (see chapter 5 CIM Integration on page 15) derived. The class names and the name of the implementing libraries are specified below.

4.1 Operating System Metrics

The following metrics are implemented by the OS Plugin library:

Metric	**Type**
NumberOfUsers	Regular
NumberOfProcesses	Regular
MemorySize	Auxiliary
TotalVisibleMemorySize	Regular
SizeStoredInPagingFiles	Regular
FreeVirtualMemory	Regular
TotalVirtualMemorySize	Regular
FreePhysicalMemory	Regular
FreeSpaceInPagingFiles	Regular
PageInRate	Regular
LoadAverage	Regular
OperationalStatus	Regular
CPUTime	Auxiliary
KernelModeTime	Regular
UserModeTime	Regular
TotalCPUTime	Regular

Metric Definition CIM Class	Linux_OperatingSystemMetric
Metric Value CIM Class	Linux_OperatingSystemMetricValue
Plugin Library Name	libmetricOperatingSystem.so

4.2 Process Metrics

The following metrics are implemented by the Process Plugin library:

Metric	**Type**
CPUTime	Auxiliary
KernelModeTime	Regular
UserModeTime	Regular
TotalCPUTime	Regular
ResidentSetSize	Regular

Metric Definition CIM Class	Linux_UnixProcessMetric

Metric Value CIM Class	Linux_UnixProcessMetricValue
Plugin Library Name	libmetricUnixProcess.so

4.3 File System Metrics

The following metrics is implemented by the File System Plugin library:

Metric	**Type**
AvailableSpace	Regular
AvailableSpacePercentage	Regular

Metric Definition CIM Class	Linux_LocalFileSystemMetric
Metric Value CIM Class	Linux_LocalFileSystemMetricValue
Plugin Library Name	libmetricLocalFileSystem.so

4.4 NetworkPort Metrics

The following metrics are implemented by the Processor Plugin library:

Metric	**Type**
BytesSubmitted	Auxiliary
BytesTransmitted	Regular
BytesReceived	Regular
ErrorRate	Regular

Metric Definition CIM Class	Linux_NetworkPortMetric
Metric Value CIM Class	Linux_NetworkPortMetricValue
Plugin Library Name	libmetricNetworkPort.so

Note: NetworkPortUtilizationPercentage is a metric prescribed by the monitoring model which is NOT implemented.

4.5 Processor Metrics

The following metrics are implemented by the Processor Plugin library:

Metric	**Type**
TotalCPUTimePercentage	Regular

Metric Definition CIM Class	Linux_ProcessorMetric
Metric Value CIM Class	Linux_ProcessorMetricValue
Plugin Library Name	libmetricProcessor.so

5 CIM Integration

A very important aspect of the Gatherer design is that it will be used for implementation of the CIM Monitoring model. As the metrics in this model have the same structure, it is beneficial to have a set of generic, parameterizable CIM providers that access the Gatherer and the repository.

The generic CIM providers are configured to implement certain metrics via special classes that have to be instantiated by the instrumentation writer. This way the the Gatherer can be instructed to load the appropriate plugins.

5.1 Generic Metric Provider Description

The generic metric providers are routing CIM operation requests to the Gatherer or the Repository. For this purpose they must know the association between the CIM metrics (definitions and values) they are serving and the plugin libraries implementing them for the Gatherer/Repository. They are examining the instances of the Linux_MetricPlugin/Linux_RepositoryPlugin classes which contain the name of the metric class and the name of the shared library to load.

The generic providers implement CIM Operation requests for the BaseMetricDefinition and BaseMetricValue classes. BaseMetricDefinitions and BaseMetricValues are identified by the InstanceId and MetricDefinitionId properties these are mapped to the Gatherer specific ids when calling the Gatherer remote APIs. We will briefly describe the CIM class and the providers if available.

5.1.1 Linux_MetricGatherer

This CIM class derived from Service represents the Gatherer daemon. It has one instance (per computer system) and must be in the “started” state before any operations on Metric Definitions and Values can take place. The following methods are defined for this class:

StartService	Inherited, starts the Service (spawns gatherd)
StopService	Inherited, stops the Service (kills gatherd)
StartSampling	Begins Metric Sampling
StopSampling	Ends Metric Sampling

Further, the following properties are defined in addition to those inherited:

uint16 NumberOfPlugins	Number of loaded plugin libraries
uint16 NumberOfMetrics	Number of registered Metric Definitions

The provider for Linux_MetricGatherer is Linux_MetricGathererProvider and is contained in the shared library libOSBase_MetricGathererProvider.so.

The following provider operations are supported by this provider

enumerateInstances

This operation will return the single instance of MetricGatherer.

enumerateInstanceNames

Like enumerateInstances, but will return the CIM object path of the service.

getInstance

For completeness mainly: retrieves the instance of MetricGatherer according to the passed-in object path.

invokeMethod

This implements the model methods for the class Linux_MetricGatherer as defined above. The StartService operation loads all metric plugins represented by instances of the CIM class Linux_MetricPlugin.

5.1.2 Linux_MetricRepositoryService

This CIM class derived from Service represents the Repository Service daemon. It has one instance and must be in the “started” state before any operations on Metric Definitions and Values can take place. The following methods are defined for this class:

StartService	Inherited, starts the Service (spawns reposd)
StopService	Inherited, stops the Service (kills reposd)

Further, the following properties are defined in addition to those inherited:

uint16 NumberOfPlugins	Number of loaded plugin libraries
uint16 NumberOfMetrics	Number of registered Metric Definitions

The provider for Linux_MetricRepositoryService is Linux_MetricRepositoryServiceProvider and is contained in the shared library libOSBase_MetricRepositoryServiceProvider.so.

The following provider operations are supported by this provider

enumerateInstances

This operation will return the single instance of MetricRepositoryService.

enumerateInstanceNames

Like enumerateInstances, but will return the CIM object path of the service.

getInstance

For completeness mainly: retrieves the instance of MetricRepositoryService according to the passed-in object path.

invokeMethod

This implements the model methods for the class Linux_MetricRepositoryService as defined above. The StartService operation loads all metric plugins represented by instances of the CIM class Linux_RepositoryPlugin.

5.1.3 CIM_BaseMetricDefinition

All metric definitions served by the generic MetricDefinition provider are derived from the CIM class BaseMetricDefinition. The metric definition provider acts as instance provider for BaseMetricDefinition objects and as association provider[1] for MetricDefForME and MetricInstance. It interfaces to the Repository Service metric registry.

The generic MetricDefinition provider is called OSBase_MetricDefinitionProvider and is contained in the shared library named libOSBase_MetricDefinitionProvide.so.

This provider requires that the Repository Service is in the started state. The following provider operations are supported:

enumerateInstances

This operation requests a list of metrics for the specified Metric Definition class (implemented by a plugin library) and converts them to CIM instances.

enumerateInstanceNames

Like enumerateInstances, but will return CIM object paths.

getInstance

Retrieves a single metric definition from the repository.

associators[1]

Depending on the source object path returns CIM instances for either the metric definitions for a managed element (or metric value) or the measured elements (or metric values) for a metric definition.

Note that this call will need a resource-to-objectpath mapping function that has to be provided as plugin by the metric provider writer. Reason is that the gatherer plugins are not aware (and should not be) of CIM naming concepts.

associatorNames[1]

Like associators, but returns CIM object paths.

5.1.4 CIM_BaseMetricValue

All metric values will be implemented as instances of CIM classes derived from BaseMetricValue. The metric value provider acts as instance provider for BaseMetricValue objects and as association provider[2] for MetricForME. It interfaces to the Repository Service metric repository.

The generic MetricValue provider is called OSBase_MetricValueProvider and is contained in the shared library named libOSBase_MetricValueProvider.so.

This provider requires that the Repository Service is in the started state. The

1 Still to be designed.
2 Still to be designed.

following operations are supported:

enumerateInstances

This operation requests a list of metric values from the repository for the specified Metric Value class using the metric definition id and converts them to CIM instances.

enumerateInstanceNames

Like enumerateInstances, but returns CIM object paths.

getInstance

Retrieves a single metric value from the repository.

associators[2]

Depending on the source object path returns CIM instances for either the metric values for a managed element or the measured elements for a metric value.

Again, this call will need the resource-to-objectpath mapping function mentioned in the metric definition provider paragraph.

associatorNames[2]

Like associators, but returns CIM object paths.

6 Gatherer Usage

The Gatherer package consists of

- the Gatherer daemon (gatherd)
- the Gatherer control program (gatherctl)
- the Repository Service daemon (reposd)
- the Repository Service control program (reposctl)
- shared libraries
- the generic providers
- the plugins

The Gatherer and Repository daemons can be operated standalone or in the context of a CIMOM. Both usage scenarios are described now.

6.1 Installation

Before it can be used, the Gatherer package must be installed. As we have no binary package at the moment it is necessary to obtain the source package. The reader is referred to http://oss.software.ibm.com/sblim for details on how to do that.

Once the source code has been downloaded, it can be compiled by changing to the package's main directory and executing the make command. Refer to the accompanying README for further instructions, requisites, etc.

By executing “make install” the binaries are copied into the system directories. It will be necessary to do this as root on the majority of systems. It might also be necessary to adjust either the LD_LIBRARY_PATH or /etc/ld.so.conf to make sure that the shared libraries will be found by the shared object loader.

If the CIMOM part is to be used, it is necessary to change to the provider subdirectory and invoke “make” and “make install” there too.

6.1.1 Important Files

It is assumed that the gatherer components are installed to /usr/bin and /usr/lib.

Name	***Description***	***Needed by***
/usr/bin/gatherd	The Gatherer daemon.	Everybody
/usr/bin/gatherctl	The Gatherer control program.	Adminstrators and curious minds.
/usr/bin/reposd	The Repository daemon.	Everybody
/usr/bin/reposctl	The Repsoitory control program.	Adminstrators and curious minds.
/usr/lib/libmcserv.so	The communication layer library.	Everybody.
/usr/lib/libgather.so	The Gatherer core library.	The Gatherer.
/usr/lib/librgather.so	The Gatherer Remote Interface Library.	Clients: gatherctl, CIM Providers.
/usr/lib/librepos.so	The Repository core library.	The Gatherer.
/usr/lib/librrepos.so	The Repository Remote Interface Library.	Clients: reposctl, CIM Providers.

Further, there are files needed for CIM support, namely the provider modules for BaseMetricDefinition, BaseMetricValue, GathererService and MetricRepositoryService. The libraries are (in the same order):

- libOSBase_MetricDefinitionProvider.so
- libOSBase_MetricValueProvider.so
- libOSBase_MetricGathererProvider.so
- libOSBase_MetricRepositoryServiceProvider.so

These files get installed to the directory where the CIMOM expects to find provider modules. Please check the CIMOM's documentation for details.

Last but not least, there are the plugins for metric retrieval. See the list in chapter 4 starting on page 13.

6.2 Standalone Usage

The operation of the Gatherer and Repository can be controlled through the Gatherer control program gatherctl and the Repository control program reposctl. Using them l it is possible to start, stop and query the daemons. Further, metric plugins can be loaded and unloaded and metric values can be retrieved.

6.2.1 Gatherctl

Invocation Syntax

gatherctl

Description

Gatherctl is an interactive program that will support a number of commands when started. The commands are

h	Prints help
s	Display Status
i	Initialize Processing
t	Terminate Processing
b	Begin Sampling
e	End Sampling
l pluginname	Load a plugin library
u pluginname	Unload a plugin library
v pluginname	List metric definitions for loaded plugin library
q	Quit gatherctl, the daemon continues to run
k	Kill daemon process
d	Start daemon process

Example Session

The following lines show a usage sample for gatherctl, where the daemon will be started, a plugin library will be loaded and queried. User entered text is in bold typeface.

```
gatherctl
d
  PID TTY          TIME CMD
i
l libmetricOperatingSystem.so
```

```
v libmetricOperatingSystem.so
Plugin metric "NumberOfUsers" has id 111
Plugin metric "NumberOfProcesses" has id 112
Plugin metric "CPUTime" has id 113
Plugin metric "MemorySize" has id 114
Plugin metric "PageInCounter" has id 115
Plugin metric "LoadCounter" has id 116
b
```

As can be seen, the output of gatherctl is not really suitable for the end user. Instead, it is a tool for the writer of plugins to help in testing and debugging.

6.2.2 Gatherd

Syntax

gatherd

Description

Gatherd is the binary implementing the Gatherer daemon. Usually it is not started manually but through another program like gatherctl or a CIMOM provider. Important to notice is that gatherd is needing two files for internal operation:

```
/tmp/.gather-lockfile
/tmp/.gather-socket
```

Should problems occur, e.g., after running gatherd first as superuser and then as regular user, these file must be removed manually.

6.2.3 Reposctl

Invocation Syntax

reposctl

Description

reposctl is an interactive program that will support a number of commands when started. The commands are

h	Prints help
s	Display Status
i	Initialize Processing
t	Terminate Processing
b	Begin Sampling
e	End Sampling
l pluginname	Load a plugin library

u pluginname	Unload a plugin library
v pluginname	List metric definitions for loaded plugin library
q	Quit gatherctl, the daemon continues to run
k	Kill daemon process
d	Start daemon process
g id [resource [from [to]]]	Retrieves metric values for a given metric definition id. Additionally it is possible to filter by resource and specify an interval (seconds since now).

Example Session

The following lines show a usage sample for reposctl, where the daemon will be started, a plugin library will be loaded and queried. Further, metric values are being retrieved. User entered text is in bold typeface.

```
reposctl
d
  PID TTY          TIME CMD
i
l librepositoryOperatingSystem.so
v librepositoryOperatingSystem.so
Plugin metric "NumberOfUsers" has id 111 and data type 102
Plugin metric "NumberOfProcesses" has id 112 and data type 102
Plugin metric "CPUTime" has id 113 and data type 2000
Plugin metric "KernelModeTime" has id 114 and data type 104
Plugin metric "UserModeTime" has id 115 and data type 104
Plugin metric "TotalCPUTime" has id 116 and data type 104
Plugin metric "MemorySize" has id 117 and data type 2000
Plugin metric "TotalVisibleMemorySize" has id 118 and data type 104
Plugin metric "FreePhysicalMemory" has id 119 and data type 104
Plugin metric "SizeStoredInPagingFiles" has id 120 and data type 104
Plugin metric "FreeSpaceInPagingFiles" has id 121 and data type 104
Plugin metric "TotalVirtualMemorySize" has id 122 and data type 104
Plugin metric "FreeVirtualMemory" has id 123 and data type 104
Plugin metric "PageInCounter" has id 124 and data type 104
Plugin metric "PageInRate" has id 125 and data type 104
Plugin metric "LoadCounter" has id 126 and data type 402
Plugin metric "LoadAverage" has id 127 and data type 402
g 114
Value id 114 has value data
         for resource OperatingSystem 589070, sample time 1094743734
duration 0
g 123 * -60 -30
Value id 123 has value data
         for resource OperatingSystem 997484, sample time 1094743794
duration 0
```

6.2.4 reposd

Syntax

reposd

Description

reposd is the binary implementing the Repository Service daemon. Usually it is not started manually but through another program like reposctl or a CIMOM provider. Important to notice is that reposd is needing two files for internal operation:

```
/tmp/.reposd-lockfile
/tmp/.reposd-socket
```

Should problems occur, e.g., after running reposd first as superuser and then as regular user, these file must be removed manually.

6.3 CIMOM Usage

The CIM providers for the Gatherer must be registered with the CIMOM before they can be used. The sample makefile in the Gatherer package is written for the Pegasus CIMOM. Performing a “make install” in the provider subdirectory will install and register the providers. Other CIMOMs will have different registration procedures.

After the providers have been registered it is possible to start and stop the MetricGatherer Service (which is the CIM representation of the daemon), start and stop the gathering process and retrieve Metric Definitions and Values as well as starting and stopping the MetricRepositoryService.

With the MetricRepositoryService active it is now possible to enumerate instances and names of BaseMetricDefinitions and BaseMetricValues (and derived classes, of course).

6.4 Writing and Deploying Metric Plugins

A writer of performance metric instrumentation must perform the following steps:

– Write the plugin libraries for the Gatherer and the Repository Service
– Derive MetricDefinition and MetricValue classes for the set of metrics to be implemented by the plugin libraries (class MOFs)
– Write a resource to object path mapper plugin for the Generic Provider[3]
– Register the newly defined classes with the generic metric provider (registration MOF)
– Instantiate the Linux_MetricPlugin and Linux_MetricRepositoryPlugin class instances necessary to link the MetricDefinitions/MetricValues to the plugin libraries (instance MOF)

3 Still in need of definition

– Deploy the plugin libraries by copying them into a directory accessible by the Gatherer

7 Reference Materials

Add sample code, MOFs, URLs, etc,...

8 Glossary

Some terms need definition...