Metrics Service

Metrics Service Overview

The Cougaar Metrics Service records and integrates metrics data captured by local sensors, such as CPU load and message traffic sensors, and advertises these real-time metrics to local components and remote user interfaces. For example servlet view snapshots, see the Operation Using Servlets page. The remainder of this manual covers the design, configuration, and developer use of the Metrics Service.

See here for Design and Architecture. The Metrics Service constructs and maintains a data model which integrates raw sensor values from sources throughout any given Node's containment hierarchy, reconciling conflicting sensor view as needed. The resulting metrics are accessible to any client Component in that Node. The data-model perspective is always local to the given Node, though the information may also include data about, and in some cases from, remote Nodes and Agents.

The Metrics Service data model is updated in real-time in a demand-based way. As a result, a given data model becomes a more comprehensive view of the society as a whole not only as available data increases but also as demand increases. The services themselves do not do any measurements. Their function is only to collect and integrate raw values into a coherent and reliable data model, and to provide integrated best-guess metrics to clients that need them.

Note that the Metric Service role is limited to capturing the current *state* of the society and its underlying resources. The Metrics Service gives the best guess at what the society is doing *now* from the local Nodes perspective. The Metrics Service does *not* support other kinds of statistics gathering, such as event/traps, time history, resource control, or policy management. Other Cougaar services handle these kinds of access mechanisms, even though the underlying statistics may be similar.

A Node's Metrics Service data model can be accessed by several ways, each tailored for a different type of consumer. When reading, Metrics are named based on looking of a *formula* in the data model. Since the lookup can follow relative relationship between *resource contexts* in the model, the lookup syntax is called a Path.

- Components can read metrics using direct calls to the MetricsService interface. The interface supports both query or subscribe to specific metrics.
- Operators and debuggers can access tables of metrics using Servlets
- Management/Monitoring Agents can subscribe to metrics which are not collected locally, but which are collected on remote nodes. The Gossip Service will automatically transport these metrics to the local node.
- The Adaptivity Engine has a standard interface and naming convention for accessing metrics in rule-books.

- ACME and CSMART will be able to access metrics via a Servlet which returns queries in XML or Cougaar Properites format. (Not available in 10.4.0).
- The QuO framework for generating QoS adaptive Aspects and Comonents has access to metrics, by using QuO SysConds.

The Metrics Services has several ways for sensors to write metric values into the Node's local data model. When writing, Metrics are named using on a hieractical naming convention called Keys

- Components can write metrics using direct calls to the MetricsUpdateService
- Network/System Management Systems can import external information about the network and host configurations. Special DataFeeds can be created to import information in different formats. For example, default configuation information can be imported from a web page or file using a PropertiesDataFeed, see Configuring The Metrics Service.

Sources of the Metrics Service can be explained as differing flows, with data from sensor code, netowrk management configuration files, and even host-level probes.

- Sensor Design & Creation describes each data flow, and the process in Cougaar in which to create new sensors.

Engineering Assessment

Null Metrics Service

For engineering assessment, it is possible to configure the metric service to add or remove functionality. The assessment idea is to remove as much functionality as possible to obtain a base line. The functionality is then added back piece by piece and the system is re-evalutate to determine the overhead.

The first level of testing is to remove the metrics service sensors and clients. To remove the clients of the service, the metrics rules should not be loaded into the society XML. Past measurements show that the metric service clients and less than 10% load on a Node.

Another level of base-lining is to remove the metric-service implementation itself. The Metric Service is really a wrapper around the QuO Resource Status Service. If the RSS class is not found, the wrapper will act as a blackhole. The MetricsService will accept subscriptions, queries, but the wrapper will always return a NO-VALUE Metric. Also, MetricsUpdaterService can input with key-value pairs, but the wrapper will just drop them.
The empty implementation of MetricsService returns a 0-credibility metric with the provenance set to "undefined" (this constant UndefinedMetric) for all getValue calls and does nothing for all subscribeToValue and unsubscribeToValue calls. The empty implementation of MetricsUpdateServicedoes nothing for the updateValue call.
Of course none of the Metrics clients will work, and should not be configured in the society XML.

In 11.0 you can disable the QuO -RSS implementation, getting empty implementations of the MetricsService and the MetricsUpdateService, if qos.jar isn't accessible. Simply remove the quoSumo.jar from the Class path $CIP/sys/qouSumo.jar.
In subsequent releases you get empty implementations of the MetricsService and the MetricsUpdateServiceby loading

the component org.cougaar.core.qos.metrics.MetricsServiceProvider. Full Component-loading isn't functional in 11.0, so provided is a workaround to do this with a -D flag:

```
-Dorg.cougaar.society.xsl.param.metrics=trivial.
```

Cougaar

Cougaar Agent Architecture Open-Source site

Agent Load Servlet

Agent Load for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:48:15 GMT 2004

NODE	CPUloadAvg	CPUloadMJIPS	MsgIn	MsgOut	BytesIn	BytesOut	Size
NODE1	3.32	1868.608	291.86	292.26	265861	375069	52101120
AGENTS							
NODE1	0.00	0.207	0.30	0.30	247	190	0
Source2	0.19	79.054	58.31	58.41	53123	75001	0
Source5	0.19	80.059	59.71	59.81	54396	76796	0
Source3	0.20	81.134	58.21	58.31	53032	74873	0
Source1	0.19	80.488	56.81	56.91	51758	73078	0
Source4	0.19	79.586	58.51	58.51	53305	75130	0
SERVICES							
MTS	2.00	856.542					
Metrics	0.34	144.180					
NodeRoot	0.01	4.412					

— Agent Load Servlet

CPULoadAvg

Average number of thread servicing the agent. For example, a load average of 2, means that two threads were servicing the agent for the whole averaging interval. This is unlikely and is a indicator that one of the Agent's plugins is inappropriately holding onto a thread during a sleep, or wait for I/O

CPULoadMJIPS

Average rate of CPU used servicing the agent. This is the integration load average * the effective cpu.

MsgIn

Message Per Second from all agent to this agent

MsgOut

Message Per Second from this agent to all other agents

BytesIn

Bytes Per Second from all agent to this agent

BytesOut

Bytes Per Second from this agent to all agent

PersistSize

Size of the last Agent Persist

Cougaar

Cougaar Agent Architecture Open-Source site

API

org.cougaar.core.qos.metrics.MetricsService

`Metric getValue(String path, VariableEvaluator evaluator, Properties qos_tags)`

The most general query function in the Metric Service. The `path` specifies the metric to be returned. For a description of the syntax, see the Path section. The `evaluator` is used to handle shell-variable-style references in generic paths. For more details on this, see VariableEvaluator. The `qos_tags`are for future use.

`Metric getValue(String path, Properties qos_tags)`

As above but with no variable replacement.

`Metric getValue(String path, VariableEvaluator evaluator)`

As above but with no QoS tags.

`Metric getValue(String path)`

As above but with neither variable replacement nor QoS tags.

`Object subscribeToValue(String path, Observer observer, VariableEvaluator evaluator, MetricNotificationQualifier qualifier)`

The most general subscription function in the Metric Service. The `path` specifies the metric to subscribe to. For a description of the syntax, see the Path section. The `observer` is the entity which will receive a callback when the specified metric changes. The `evaluator` is used to handle shell-variable-style references in generic paths. For more details on this, see VariableEvaluator. The `qualifier`w is used to restrict the frequency of callbacks, for example by specifying the smallest change that should trigger one. For more details, see MetricNotificationQualifier. The value returned by this method is a subscription handle and should only be used for a subsquent call to `unsubscribeToValue`.

`Object subscribeToValue(String path, Observer observer, VariableEvaluator evaluator)`

As above but without callback qualification.

`Object subscribeToValue(String path, Observer observer, MetricNotificationQualifier qualifier)`

As above but without variable replacement.

`Object subscribeToValue(String path, Observer observer)`

As above but with neither variable replacement nor callback qualification.

`void unsubscribeToValue(Object handle)`
Terminates a previously established subscription. The handle is as returned by one of the `subscribeToValue`calls.

Cougaar

Cougaar Agent Architecture Open-Source site

Configuration

Configuration

The Metrics Service is made of many components all of which are optional. If no components are added the Metrics Service will act as a black-hole with MetricsUpdates writing, but no call-backs or queries returned from the MetricsService (reader). The following types of components can be configured. The specific components are listed in the following sections along with their function, insertion point and module (jarfile).

Metrics Servlets

The following components allow viewing of Metrics through servlets. (See Using Metrics Servlets). The components are in core.jar and should be loaded into the NodeAgent.

org.cougaar.core.qos.metrics.MetricsServletPlugin
Loads the Metric Service Servlets, which are: AgentLoadServlet, RemoteAgentServlet, NodeResourcesServlet, MetricQueryServlet, MetricsWriterServlet,

CPU Load Components

The following components collect metrics about the CPU consumption of Agents. The components are in core.jar and should be loaded into the NodeAgent.

org.cougaar.core.thread.AgentLoadSensorPlugin
Sensor for measuring the CPU load for Agents.

org.cougaar.core.thread.AgentLoadRatePlugin
Converts the raw CPU sensor measurements into Metrics

org.cougaar.core.thread.TopPlugin
servlet for viewing running threads (TopServlet). Also loads the RogueThreadDetector.

Message Load Components

The following component collect metrics about the message traffic into and out of Agents. The components are in the mtsstd.jar and should be load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

org.cougaar.core.qos.metrics.AgentStatusAspect
Sensor for Measuring the message flow into and out of Agents (This Aspect is always loaded in the Base

template)

org.cougaar.mts.std.StatisticsAspect
Sensor for measuring the size of messages

org.cougaar.core.qos.metrics.AgentStatusRatePlugin
Converts the raw message sensor measurements into Metrics

org.cougaar.mts.std.StatisticsPlugin
Servlet for viewing raw message sensor counters, which include: StatisticsServlet, AgentRemoteStatusServlet, AgentLocalStatusServlet

Agent Mobility and QuO Components

The following Components pull the topology service to detect when Nodes and Agents move. Also, they offer a service for QuO Sysconds to subscribe to Metrics. They should be loaded in the insertion point Node.AgentManager.Agent.MetricsServices.Component

org.cougaar.core.qos.rss.AgentHostUpdaterComponent
Periodically polls the topology service and updates the internal Metric Service models to keep the Host-Node-Agent containment hierarchy up to date.

org.cougaar.lib.mquo.SyscondFactory
Factory for creating QuO Sysconds which track Metrics.

Persistence Size Components

The follow component measure the memory consumption of Agents. These components are in the core.jar and need to be loaded into every Agent.

org.cougaar.core.qos.metrics.PersistenceAdapterPlugin
Sensor for measuring the Agent persistence size.

Gossip Components

The gossip subsystem disseminates metrics between nodes. (See Using Metrics Servlets). The components are in qos.jar

org.cougaar.core.qos.gossip.GossipAspect
Piggybacks Metric requests and Metric Values on messages. Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

org.cougaar.core.qos.gossip.SimpleGossipQualifierComponent
Limits which Metrics should be requested or value forwarded Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

org.cougaar.core.qos.gossip.GossipFeedComponent
Metrics Service Feed for updating metrics from remote Nodes. Load at the insertion point Node.AgentManager.Agent.MetricService.Component

org.cougaar.core.qos.gossip.GossipStatisticsServiceAspect
Collects gossip overhead statistics Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

org.cougaar.core.qos.gossip.GossipStatisticsPlugin
Load Servlet for viewing Gossip overhead statistics. Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

RSS-Resource Status Service

The Metric Service need access to configuration files to define the expected network and host capacity. The Metrics-Sites.conf file is required and the Metrics-Defaults.conf file is optional. Example files are in the overlay at …/configs/rss or source in in qos/configs/rss. Also, the data feed with the name "sites" has the special purpose of defining the sites themselves (i.e.subnet masks), so other Metrics Keys should not be put in this conf file.

The cougaarconfig: url scheme means the files are on the cougaar config path. Otherwise, the url is normal and will just down load the conf files. Putting the configuration files on a web server is useful for a cougaar applications which run at a site with a complicated topology. Also, Network and Host management systems can update the files with real data. (Note the conf files are down-loaded only once at startup)

org.cougaar.core.qos.rss.ConfigFinderDataFeedComponent
Two Components should be loaded with the following parameters
`"name=sites","url=cougaarconfig:Metrics-Sites.conf"`
`"name=hosts","url=cougaarconfig:Metrics-Defaults.conf"` The insertion point is Node.AgentManager.Agent.MetricsServices.Component

Acme Scripts

Acme rule scripts are available for loading the Metrics Service. The rules are in directory `qos/csmart/config/rules/metrics`. The `rule.txt`list the recommended rules to load the standard Metrics Service. Addition rules must be added in order to measure the length of messages (serialization) and setup the network/host configuration files (rss). The rule directories have the following purpose:

`sensors`
Adds servlets that look at the raw sensors. These rules do not load any of the Metrics Service runtime, so they should not impact performance. These rules are useful for debugging and we recommended that they should alway be loaded.

`basic`
Loads the basic Metrics Service which includes some potentially high overhead components, such as the Agent-to-Agent traffic matrix and Gossip.

`serialization`

The basic MTS serialization rule for measures message length. This rule conflicts with other serialization rules, such as bandwidth measurement, security, and compression. These other aspects must be loaded in a specific order which is explained in the `metrics-only-serialization.rule` Only one serialization rule should be loaded.

`rss`

Supplying network and host configuration data to the RSS is very network specific. Each test site has its own rules which tap into the local sources of system information. If you are running at the TIC, you should load the rules in `rss/tic`

Cougaar

Cougaar Agent Architecture Open-Source site

Design and Architecture

The core underlying infrastructure used by the Metrics Services is the *Resource Status Service* (RSS). The RSS accepts low-level tagged input in the form of key-value pairs, which it can process very efficiently. The entities which handle this input are known as *DataFeeds*. This low-level data is then propagated through a hierarchical graph of data-encapuslating nodes known as *Resource Contexts*, using both forward and backward chaining. Each Resource Context supports one or more *DataFormulas*, each of which in turn can calculate a value using the encapsulated data as well as data derived from other DataFormulas . DataFormulas , in other words, form a dependency chain that transforms and processes data. The final result of all this is raw sensor-like data coming in and processed domain-relevant data going out.

The specific subclasses of Resource Context in any given RSS depends on the domain. In the case of COUGAAR, interesting Resource Contexts represent Agents, Nodes, Hosts, inter-Agent data flows, inter-Host data flows, etc. Similarly, the specific subclasses of DataFeed in any given RSS depends on the raw data sources that are available. In the case of COUGAAR these would include URLs that provide default values in property-list syntax, real-time measured host data (load average, number of open sockets, etc), and data specified directly through the MetricsUpdateService.

Queries into the MetricsService use path specifications to indicate the location of a given Resource Context in the RSS knowledge base, and a formula on that Resource Context. If the Resource Context itself can handle the query it will do so. If not it will pass it to its parent. This makes it possible to ask, say, an Agent Resource Context for Node or Host data.

Each COUGAAR Node includes its own RSS, which is primarily responsible for handling local data in the Node. In order to provide a more global view, the GossipAspect was added. Gossip defines a new kind of DataFeed which is used for inter-RSS communication, and which piggybacks requests and responses onto ordinary COUGAAR message traffic. This allows Nodes to share metrics without requiring any new form of inter-Node communication.

As an example of how data moves through the RSS, the rest of this section describes the flow of raw sensor data into the RSS from a COUGAAR Component via the MetricsUpdateService , and back out again as processed higher level data to some other Component via the MetricsService.

For more details on keys, see Keys. For more detail on paths, see Paths.

And for more information on creating new metrics & the sources of data into these metrics, see Adding New Metrics.

Example Metrics Query Client

```
/*
 * Stand alone java client that opens up a URL connection, reads an object stream a
 * outputs the result (will be a java ArrayList of 'Path|Metric's )
 * Usage: java -cp . ExampleMetricQueryClient "http://localhost:8800/\$3-69-ARBN/me
 * Must specify 'format=java' as the default is a string of xml printed out to brow
 */
package org.cougaar.core.examples.metrics;
import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ByteArrayInputStream;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

// This example requires quoSumo.jar, qos.jar and core.jar
public class ExampleMetricQueryClient
{
  // One arg here - the query string "http://host..." see usage above
  public static void main( String args[] ) throws IOException
  {
    String url = null;
    try {
      url=args[0];
      URL servlet_url = new URL(url);

      // open up input stream and read object
      try {
 // open url connection from string url
 InputStream in = servlet_url.openStream();
 ObjectInputStream ois = new ObjectInputStream(in);

 // read in java object - this a java client only
 HashMap propertylist = (HashMap)ois.readObject();
 ois.close();
```

```
  if (propertylist == null) {
      System.out.println("Null Property List returned");
      return;
  }
  // can do anything with it here, we just print it out for now
  Iterator itr = propertylist.entrySet().iterator();
  while (itr.hasNext()) {
      Map.Entry entry = (Map.Entry) itr.next();
      System.out.println(entry.getKey() +"->"+ entry.getValue());
  }
       } catch (Exception e) {
  System.out.println("Error reading input stream for url " + url + " Make sure the s
  e.printStackTrace();
       }
     } catch(Exception e){
       System.out.println("Unable to acquire URL, Exception: " + e);
     }
   }
}
```

Cougaar

Cougaar Agent Architecture Open-Source site

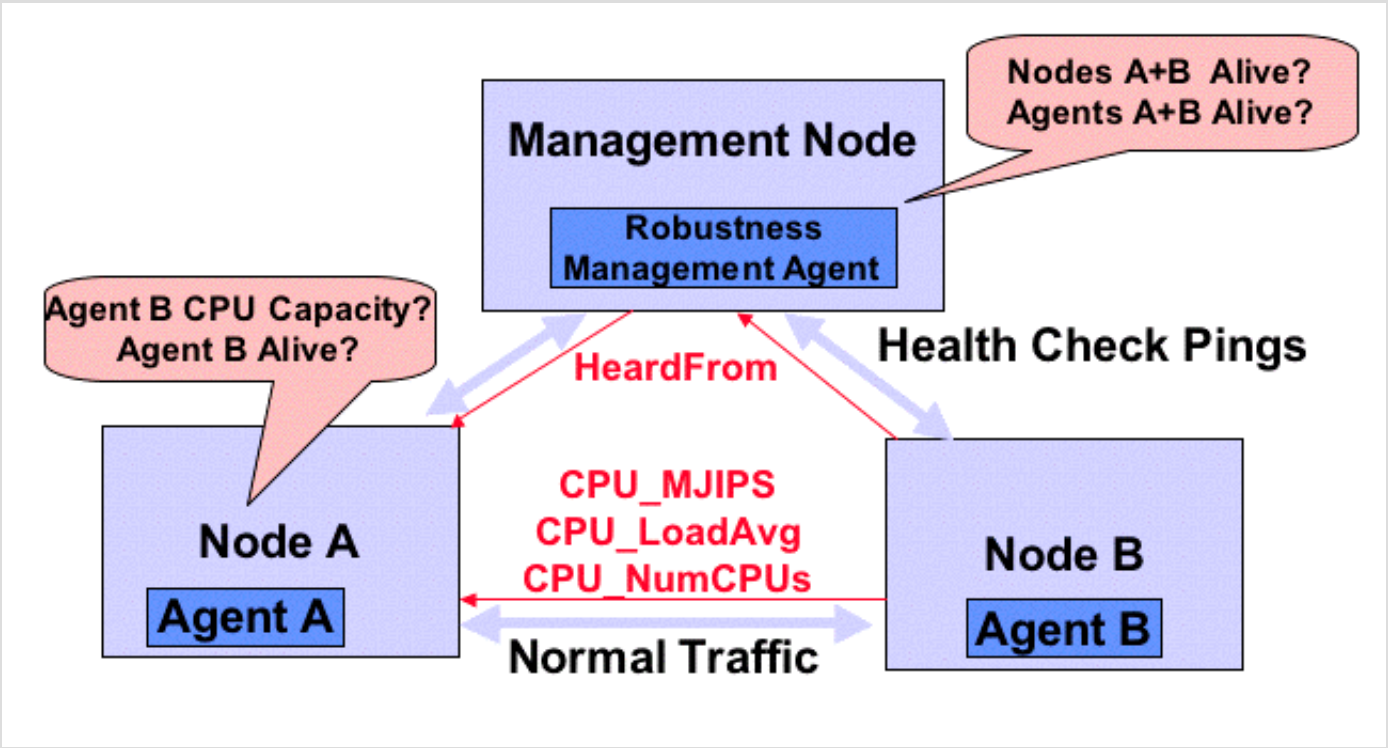
Gossip

Gossip-based Dissemination Design

The Gossip subsystem allows Metrics collected on one Node to be disseminated to other Nodes in the society. The Metrics are transfered by piggybacking Gossip objects as attributes of ordinary messages being sent between Nodes. Gossip takes two forms: requests for Metrics from a neighbor Node, and responses to those requests. Request are only made for Metrics used by the subsystems within the Node. Also, Gossip is sent only when the value of requested Metrics change. So if no Metrics are requested or the Metrics do not change, no Gossip is sent.

Gossip is spread based on the underlying message structure of the society. No additional messages are sent just to disseminate Gossip. Gossip can request metrics from neighbors which are N hops away which makes the Gossip system act like a limited range flooding algorithm. But in practice, N is set to only one hop.

The following figure shows the two types of Metrics sharing that can be accomplished by one-hop gossip.



Gossip Overview of Flow Between Nodes

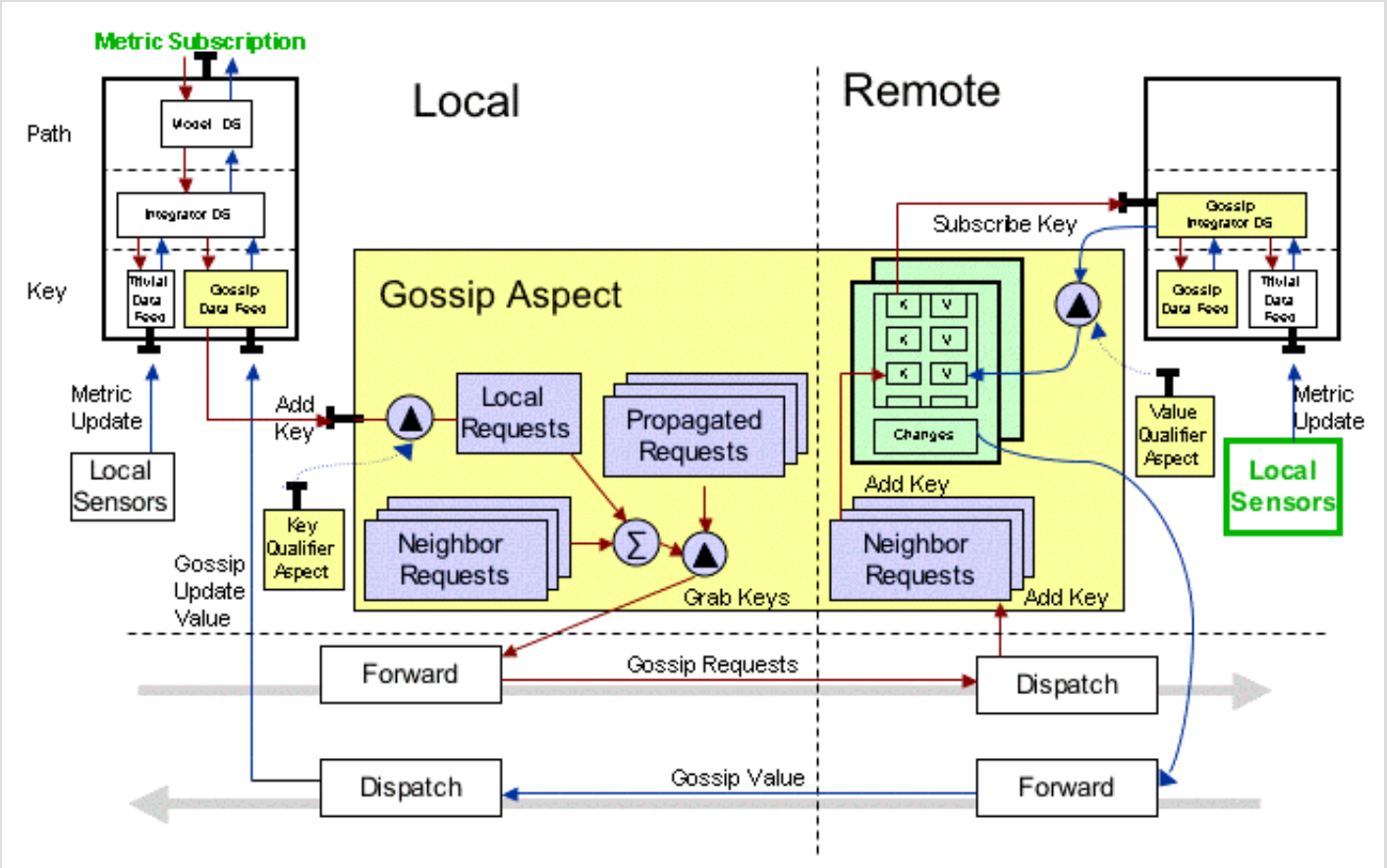
First, Metrics about the neighboring Node can be sent and received directly from the neighbor on normal traffic. The

first three messages exchanged between neighbors is enough to disseminate the static characteristics of both neighboring Agents/Nodes/Hosts, such as CPU capacity, Memory Size, Number of processors. Subsequent, messages will carry dynamic information, such as CPU Load Average.

Second, Gossip can also be sent indirectly through a chain of neighbors, which are all requesting the same Metrics. This means that one Node can have up-to-date Metrics about another Node, even though it has not recently received a message from the other Node. A typical example is a case wherethe intermediate Node contains a manager or name server Agent. These management Nodes tend to periodically send messages to a group of Nodes. The Gossip arrives on messages coming into the management Node and is propagated on all its out-bound messages. For example, a management Node may send health check pings to its managed Nodes. The sending pings will carry the dynamic Metrics for all the other managed Nodes to the pinged Node. (Note the recipient will only get information about the Metrics it has requested, even though the management Node may know addition information about other Metrics and other Nodes.) The reply from the pinged Node will carry the updated Metrics requested by the management Node.

Implementation

The Gossip implementation takes advantage of special characteristics of both the MTS and the Metrics service. Below is a figure which show the data flow between components which make up the Gossip subsystem. The red arrows are the flow of Key Requests and the blue arrows are Metric Value replies.



Gossip Internal Components

Processing Trace

The following example trace shows how a Metric Request on one Node results on an update for a metric locally, even though the information about the Metric was collected remotely.

1. A Metrics Service Client makes a request for a Metric via a Path. The Metric Formula that implements that Path, depends on many raw Metrics Values, who's names are specified by Metrics Keys.
2. The GossipDataFeed listens for requests for Keys and forwards them on to the GossipKeyDistributionService (implemented by the GossipAspect).
3. The GossipAspect checks if the Key qualifies to be sent using the GossipQualifierService (implemented by the SimpleGossipQualifierComponent).
4. The Key is added to a list of local Key requests.
5. When a message is sent to a neighbor, the GosipAspect check if there are any outstanding keys that need to be requested to this neighbors. The requests are put in a KeyGossip object and added to an attribute of the messages. (Note a separate lists of outstanding requests is kept for each neighbor, since Gossip is sent opportunistically and different neighbors have different message patterns)
6. On the receiver Node, the request is stored in the Neighbor request record.
7. A subscription is added to the local Metrics service for the request. The special GossipIntegratorDS is used to subscribe to the Key, so that the GossipDataFeed can ignore these requests as non-local and not request them (again) for its neighbors. The GossipQualifierService is used get a MetricsNotificationQualifier from the SimpleGossipQualifierComponent. The MetricsNotificationQualifier is given to the MetricService.subscribeToValue as a way to limit values returned. The qualifier sets thresholds on the change in value and credibility of requested gossip metrics. If the Metric is has not changed enough or is not credible, it is not sent.
8. The GossipIntegratorDS acts like an IntegratorDS and subscribes to Keys from all the DataFeeds. Key subscriptions can be satisfied by any DataFeed, such as the local host probe, the local metric service, or from the GossipDataFeed (neighbor's neighbors)
9. When the local value has changed, the subscription is updated. The changed values are remembered for each neighbor.
10. When a message is sent back to the original requester, the values changed for the requester put into the Message Attributes.
11. On the receive-side of the original requester, the changed values are removed from the message and published in the GossipUpateService (implemented by the GossipDataFeed).
12. The GossipDataFeed will compare the new value with the Key's current value, which could have come from any other neighbor. If the Metrics is old, lower credibility, or the same Value, the new Metric is ignored. Otherwise it becomes the current value for the Key.
13. The new value is propagated to the IntegratorDS formulas. These in-turn will propagate to other subscribing formulas, eventually calling the original Metrics service Subscriber.

Cougaar

Cougaar Agent Architecture Open-Source site

Keys

Metric Keys

This section discusses the structure of the Metrics keys name-space and defines common keys.

Metrics are written into the MetricsUpdateService as key-value pairs. While the key itself is just a string, we impose a structure to help organize the information, loosely modeled after SNMP MIBs. The value is a Metric, a structured object with slots for value, credibility, time-stamp, units, etc.

At runtime, each Node has a store of Key-Value pairs. When the MetricsUpdaterService receives a new Key-Value pair, it looks up the Key in the store. If the value has changed, all the Metrics Formulas that have subscribed to this Key will be called-back with the new Value. The Metrics Service in each Node has one and only one effective Value for each Key, which is derived through a complicated set of integration rules based on credibility and timestamps.

Normally, Key-Value pairs are not read from the Metrics Service, but are only processed by internal formulas. One debugging trick to view the effective value of a key is to the Path Integrater, which takes a Key as a parameter and returns the effective metric. Currently, there is no way to list all the Keys available to a Node.

Key Syntax

A Metrics Key is a string divided into fields using the Key Separator, e.g.

```
Host_128.33.15.114_CPU_Jips
```

The current Key Separators is "_". But your code you should use the string constant `KEY_SEPR` defined in `org.cougaar.core.qos.metrics.Constants`:

```
"Host" +KEY_SEPR+ "128.33.15.114" +KEY_SEPR+ "CPU" +KEY_SEPR+ "Jips"
```

Notice that some of the fields are types (e.g. Host, CPU, Jips) and others are identifiers (e.g. 128.33.15.114). The types are fixed and case sensitive. Identifiers are variable and define branches in naming hierarchy.

In the following section we will use the convention of writing identifier fields in brackets [], and using "_" as the Key Separator

`Host_[Host IP]_CPU_Jips`

Host

Host Keys start with identifying the host and then have several optional field for different host characteristics. Host identifiers are raw IP V4 addresses, not domain names.

`Host_[Host IP]_CPU_Jips`
: CPU capacity in Java Instructions Per Second. JIPS is determined through a benchmark.

`Host_[Host IP]_CPU_loadavg`
: CPU load average, i.e. the average number of processes that are ready to run.

`Host_[Host IP]_CPU_count`
: The number of CPUs in this host

`Host_[Host IP]_CPU_cache`
: Size of CPU level 2 cache

`Host_[Host IP]_Memory_Physical_Total`
: Total Physical Memory

`Host_[Host IP]_Memory_Physical_Free`
: Free Physical Memory

`Host_[Host IP]_Network_TCP_sockets_inuse`
: TCP sockets in use

`Host_[Host IP]_Network_UDP_sockets_inuse`
: UDP sockets in use.

IP Flow

Network Keys start with identifying the IP Flow and then have several optional field for different network characteristics. End-point addresses are raw IP V4 addresses and not domain names.

`Ip_Flow_[src IP]_[dst IP]_Capacity_Max`
: The maximum bandwidth (kbps)for path across the network, i.e. with no competing traffic.

`Ip_Flow_[src IP]_[dst IP]_Capacity_Unused`
: The expected available bandwidth (kbps) for a path across the network, i.e. the max minus the competing traffic.

Site Flow

Sites are an IP subnetwork which can be represented with a simple mask (number of bits with leading 1's). Site_Flows between sites can be used as defaults, instead of specify specific Ip_Flows. For example,

```
Site_Flow_128.89.0.0/16_128.33.15.0/28_Capacity_Max
```

Site_Flow_[src IP/mask]_[dst IP/mask]_Capacity_Max
: Maximum bandwidth (kbps) between Sites. The current formulas can model asymmetric bandwidth between Sites, by publishing Site_Flows for both direction. If only one direction is published, the formulas will assume the bandwidth is the same in both directions.

Agent

Agents are identified by their message Id.

Agent_[Message ID]_HeardTime
: System.currentTimeMillis() time-stamp for the last time some component has heard from this agent.

Agent_[Message ID]_SpokeTime
: System.currentTimeMillis() time-stamp for the last time some component has attempted to speak to this agent.

Node

Nodes are identified by their message Id. Currently, there are no Node specific Keys

---

Cougaar

Cougaar Agent Architecture Open-Source site

Local Agents Servlet

Local Agent Messages Servlet

NodeA Message Transport *Local* Agent Status

Date: Fri May 09 15:55:56 EDT 2003

Agent *Local* Status for Agent NodeA

Status	**3**
Queue Length	**0**
Messages Received	**2**
Bytes Received	**0**
Last Received Bytes	**0**
Messages Sent	**2**
Messages Delivered	**2**
Bytes Delivered	**2297**
Last Delivered Bytes	**741**
Last Delivered Latency	**0**
Average Delivered Latency	**0.0**
Unregistered Name Error Count	**0**
Name Lookup Failure Count	**0**
Communication Failure Count	**0**
Misdelivered Message Count	**0**
Last Link Protocol Tried	**null**
Last Link Protocol Success	**null**

To Change Agent use cgi parameter: ?agent=agentname

RefreshSeconds: 0

Cougaar

Cougaar Agent Architecture Open-Source site

Message Statistics Servlet

NODE1 Message Statistics

Node: NODE1
Date: Tue Nov 09 20:49:58 GMT 2004

Messages sent and received by all agents on node NODE1

	Sent	Received
Avg Queue Length	2.93	NA
Total Message Bytes	37294698326	26463063269
Total Header Bytes	14350542597	14355581175
Total Ack Bytes	14355582657	15192973871
Total Bytes	66000823580	56011618315
Total Intra-node Messages	0	0
Total Message Count	29049185	29049186

Sent Message Length Histogram

Size	Count
100	0
200	0
500	1
1000	7119
2000	29042066
5000	1
10000	0
20000	0
50000	0
100000	0
200000	0
500000	0
1000000	0
2000000	0
5000000	0
10000000	0

— Message Statistics Servlet Snapsho

AvgQueueLength

Average of the number of messages waiting on all Destination Links. The average is calculated using a decaying average

Total Bytes

Count of the number of bytes set from all agents on this node. This statistic does not include inter-node messages, which are not serialized

Total Count

Count of Messages sent from all agents on this node. This count includes both inter-node messages and intra-node messages

Message Length Histogram

Count of Messages sent from all agents on this node, organized by size. The bins are labeled by the max size in the bin

MetricNotificationQualifier

org.cougaar.core.qos.metrics.MetricNotificationQualifier

This qualifier is used to restrict the frequency of callbacks, for example by specifying the smallest change that should trigger one. The qualifier is used as part of the Metrics Service subscribeToValue() method, an instance is given for each subscription. In order to filter the notifications, the MetricsNotificationQualifier will need some state, such as the value of the last notification.

`boolean shouldNotify(Metric metric)`

Returns true iff the given metric has changed enough to generate notifications to listeners.

MetricsUpdateService

org.cougaar.core.qos.metrics.MetricsUpdateService

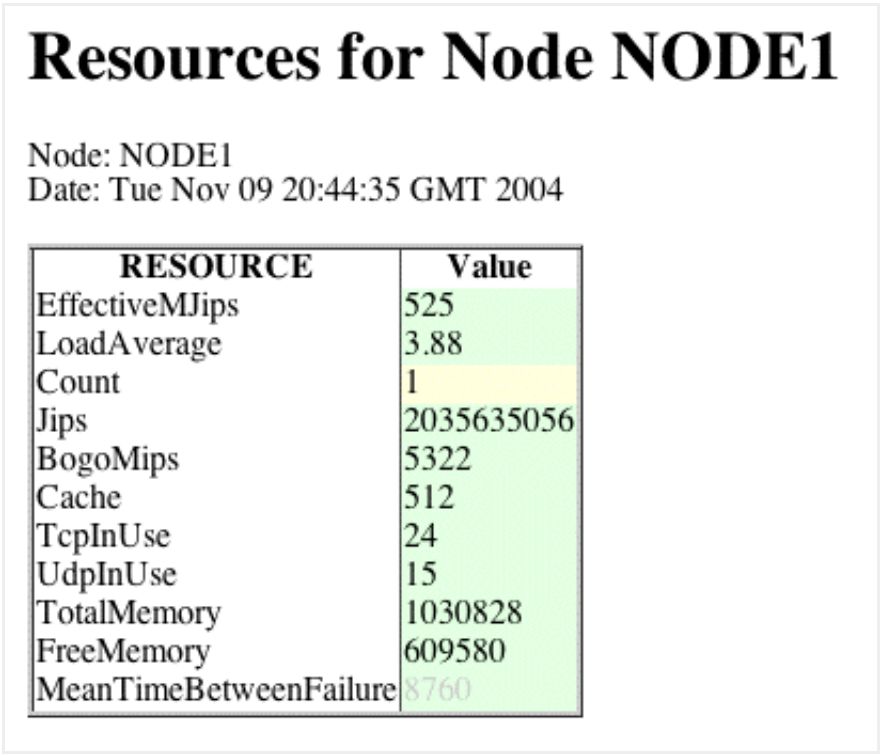
```
void updateValue(String key, Metric value)
```

Adds a new or updated metric to the Metrics Service. The syntax of a key is described in the Keys section.

Cougaar

Cougaar Agent Architecture Open-Source site

Node Resources Servlet



Resources for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:44:35 GMT 2004

RESOURCE	Value
EffectiveMJips	525
LoadAverage	3.88
Count	1
Jips	2035635056
BogoMips	5322
Cache	512
TcpInUse	24
UdpInUse	15
TotalMemory	1030828
FreeMemory	609580
MeanTimeBetweenFailure	8760

— Node Resources Servlet

EffectiveMJips

Effective CPU that one thread will experience. This is a model of the Linux CPU scheduling algorithm. The model takes into account the number of CPUs, MJIPS for each CPUs, and the load average of the host.

Load Average

Host Load Average is the number of processes running in Host.

Count

Number of Processors

TCP In Use

The number of TCP Sockets

UDP In Use
The number of UDP Sockets

Total Memory
Host Memory

Free Memory
Free memory

Cache
Size of a CPU’s level 2 cache

Cougaar

Cougaar Agent Architecture Open-Source site

Paths

Metric Paths

This section discusses the structure of the Metrics Paths and defines common Paths.

Metrics read from the MetricsService are the values from a real-time model of the status of the Cougaar Society and system resources. The MetricsService allow access to *formulas* in the model. Formulas are relative to a *Resource Context*, which define instances of modeled entries and there relationship between each other.

Containment is a major relationship which is modeled. Child context inherit all the formulas of its parent. The useful containment relationship in Cougaar are `Host->Node->Agent`. For example, Agents have all the formulas of their hosts, and when agents moves to a new host the metrics track new hosts values.

Path Syntax

A Path is a series of Contexts followed by a Formula. If there is more than one child Context of the same type in the parent Context, then the child Context is narrowed by Parameters. Each Context has a fixed number of parameters and parameter order is important. The Context type is used as the name of the context. Context and Formula names are case sensitive. For example:

```
IpFlow(128.33.15.114,128.33.15.113):CapacityMax
```

The separator between Contexts and formulas is the ":" and the separator between parameters is the ",". But your code you should use the constant `PATH_SEPR` defined in interface `org.cougaar.core.qos.metrics.Constants`:

```
"IpFlow(128.33.15.114,128.33.15.113)" +PATH_SEPR+ "CapacityMax"
```

The following sections will use the convention of writing variable parameter fields in brackets [] for variable parameters; using ":" as the Path Separator; and "," as the parameter separator. The syntax is as follows

```
ContextType1([parameter1],[parameter2]...):ContextType2([parameter]):Formula(parame
```

The Averaging Period are parameters for some Formulas. The Averaging Periods are: "10", "100", and "1000". This

section will use the convention 1xxxSecAvg to represent the averaging period.

Also, please use the constants interface in core module `org.cougaar.core.qos.metrics.Constants` whenever possible. This will allow compile time detection of errors, when the Metrics Service interface inevitably changes in the future.

Host Context

Host Contexts can be accessed at root level and take one parameter which is the Host address. The Host address can be IP V4 address or the domain name. Hosts contexts are not contained in other Contexts.

`Host([host Addr]):EffectiveMJips`
The expected per thread Millions of Java Instructions Per Second. The formula takes into account the base CPU JIPS, the number of CPUs, and the Load Average on the Host.

`Host([Host Addr]):Jips`
CPU capacity in Java Instructions Per Second. JIPS is determined through a benchmark.

`Host([Host Addr]):LoadAverage`
CPU load average, i.e. the average number of processes that are ready to run.

`Host([Host Addr]):Count`
The number of CPUs in this host

`Host([Host Addr]):Cache`
Size of CPU level 2 cache

`Host([Host Addr]):TotalMemory`
Total Physical Memory

`Host([Host Addr]):FreeMemory`
Free Physical Memory

`Host([Host Addr]):TcpInUse`
TCP sockets in use

`Host([Host Addr]):UdpInUse`
UDP sockets in use

Node Context

Node Contexts can be accessed at root level and take one parameter which is the the Node's Message address. The Node is also contained within a Host Context and inherits all the Host Context's formulas.

`Node([Node ID]):CPULoadAvg(1xxxSecAvg)`

The average number of threads working for all component on this Node during the Averaging Period.

`Node([Node ID]):CPULoadMJips(1xxxSecAvg)`
The average MJIPS used by all the threads for all components on this Node during the Averaging Period.

`Node([Node ID]):MsgIn(1xxxSecAvg)`
The average messages into all Agents on this Node during the Averaging Period.

`Node([Node ID]):MsgOut(1xxxSecAvg)`
The average messages out of All Agents this Node during the Averaging Period.

`Node([Node ID]):BytesIn(1xxxSecAvg)`
The average number of bytes (for all messages) into all Agents on this Node during the Averaging Period.

`Node([Node ID]):BytesOut(1xxxSecAvg)`
The average number of bytes (for all messages) out of All Agents on this Node during the Averaging Period.

`Node([Node ID]):VMSize`
Size in Bytes of the Node's VM

`Node([node Addr]):Destination([Agent ID]):MsgTo(1xxSecAvg)`
Message per second from all agents on the Node, to the destination Agent

`Node([node Addr]):Destination([Agent ID]):MsgFrom(1xxSecAvg)`
Message per second to any agent on the Node, from the destination Agent

`Node([node Addr]):Destination([Agent ID]):BytesTo(1xxSecAvg)`
Bytes per second from all agents on the Node, to the destination Agent

`Node([node Addr]):Destination([Agent ID]):BytesFrom(1xxSecAvg)`
Bytes per second to any agent on the Node, from the destination Agent

`Node([node Addr]):Destination([Agent ID]):AgentIpAddress`
Ip address of destination Agent

`Node([node Addr]):Destination([Agent ID]):CapacityMax`
Maximum capacity of the network between Node and destination Agent.

`Node([node Addr]):Destination([Agent ID]):OnSameSecureLAN`
True if Node and Destination are on the same secure LAN. The current formula just checks if the network capacity between the Node and Agent is greater than or equal to 10 Mbps.

Agent Context

Agent Contexts can be accessed at root level and take one parameter which is the the Agent's Message address. The Agent Context is also contained within a Node Context and inherits all the Node Context's formulas. When an Agent moves to a new Node, the Agent Context will be moved the the corresponding Node, i.e. all the re-wiring is automatic, but may be slightly delayed due to discovery issues.

`Agent([Agent ID]):LastHeard`
Seconds since some component has heard from this agent. This can be from any source such as *successful*communication, acknowledgment, or gossip.

`Agent([Agent ID]):LastSpoke`
Seconds since some component attempted to Speak to the Agent. The attempt does not have to be successful. For example, if a failed message is retied, LastSpoke will be the time of last retry

`Agent([Agent ID]):SpokeErrorTime`
Seconds since last error in communications. This is a large number with 0.0 credibility, if no error has occurred.

`Agent([Agent ID]):CPULoadAvg(1xxxSecAvg)`
The average number of threads working for this Agent during the Averaging Period.

`Agent([Agent ID]):CPULoadMJips(1xxxSecAvg)`
The average MJIPS used by all the threads for this Agent during the Averaging Period.

`Agent([Agent ID]):MsgIn(1xxxSecAvg)`
The average messages into this Agent during the Averaging Period.

`Agent([Agent ID]):MsgOut(1xxxSecAvg)`
The average messages out of this Agent during the Averaging Period.

`Agent([Agent ID]):BytesIn(1xxxSecAvg)`
The average number of bytes (for all messages) into this Agent during the Averaging Period.

`Agent([Agent ID]):BytesOut(1xxxSecAvg)`
The average number of bytes (for all messages) out of this Agent during the

`Agent([Agent ID]):PersistSizeLast`
Size in Bytes of the last persistence file for this agent

IpFlow Context

IpFlow Context can be accessed at root level and takes two parameters which is the source and destination host addresses. The Host address can be the IP V4 address or the domain name.

`IpFlow([SrcAddr],[DstAddr]):CapacityMax`
Maximum Bandwidth between two hosts

`IpFlow([SrcAddr],[DstAddr]):CapacityUnused`
Unused Bandwidth between two hosts

AgentFlow Context

AgentFlow Context can be accessed at root level and takes two parameters which is the source and destination agent message address.

Since 11.2 the AgentFlow context is no longer used, because it added too many formulas to the metrics service when societies are large.

`AgentFlow([SrcAddr],[DstAddr]):MsgRate(1xxxSecAvg)`
Messages per Second from Source Agent to Destination Agent.

`AgentFlow([SrcAddr],[DstAddr]):ByteRate(1xxxSecAvg)`
Bytes per Second from Source Agent to Destination Agent.

Remote Agents Servlet

NodeA Message Transport *Remote* Agent Status

Date: Fri May 09 15:58:27 EDT 2003

Agent *Remote* Status for Agent AgentB

Status	**3**
Queue Length	**0**
Messages Received	**773**
Bytes Received	**0**
Last Received Bytes	**0**
Messages Sent	**773**
Messages Delivered	**773**
Bytes Delivered	**998489**
Last Delivered Bytes	**1293**
Last Delivered Latency	**197**
Average Delivered Latency	**108.9337908185235**
Unregistered Name Error Count	**0**
Name Lookup Failure Count	**0**
Communication Failure Count	**0**
Misdelivered Message Count	**0**
Last Link Protocol Tried	**org.cougaar.core.mts.RMILinkProtocol**
Last Link Protocol Success	**org.cougaar.core.mts.RMILinkProtocol**

To Change Agent use cgi parameter: ?agent=agentname

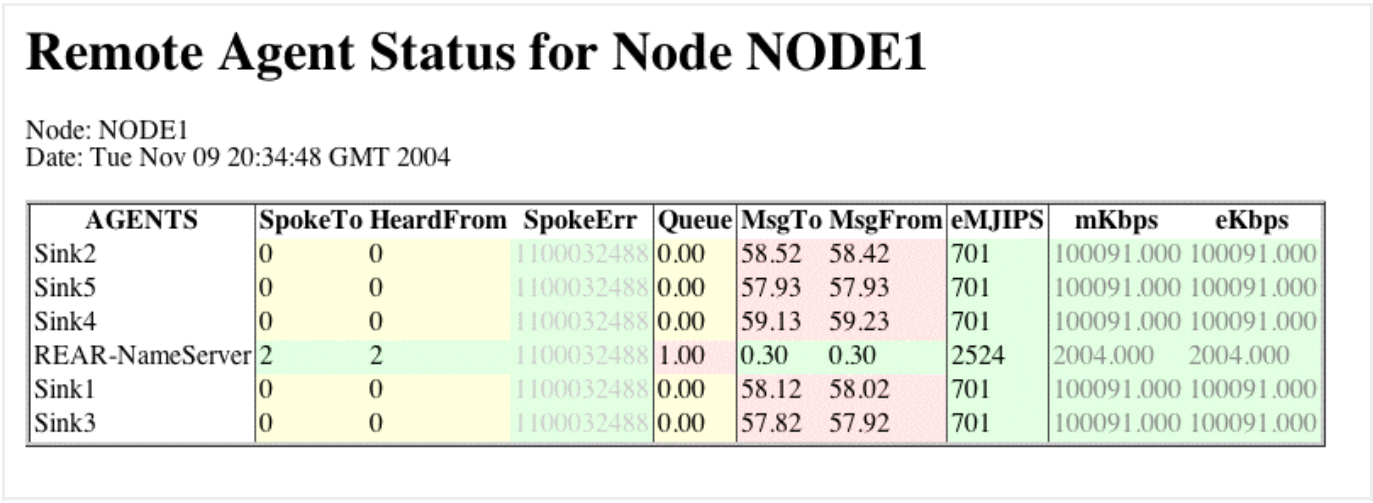
RefreshSeconds: 0

— Remote Agent Messages Servlet

Cougaar

Cougaar Agent Architecture Open-Source site

Remote Status Servlet



Remote Agent Status for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:34:48 GMT 2004

AGENTS	SpokeTo	HeardFrom	SpokeErr	Queue	MsgTo	MsgFrom	eMJIPS	mKbps	eKbps
Sink2	0	0	1100032488	0.00	58.52	58.42	701	100091.000	100091.000
Sink5	0	0	1100032488	0.00	57.93	57.93	701	100091.000	100091.000
Sink4	0	0	1100032488	0.00	59.13	59.23	701	100091.000	100091.000
REAR-NameServer	2	2	1100032488	1.00	0.30	0.30	2524	2004.000	2004.000
Sink1	0	0	1100032488	0.00	58.12	58.02	701	100091.000	100091.000
Sink3	0	0	1100032488	0.00	57.82	57.92	701	100091.000	100091.000

— Remote Agent Status Servlet

Spoke To

number of seconds since this node has spoke to this agent.

Heard From

number of seconds since something has heard from this agent (includes other node via gossip)

Spoke Error

number of seconds since the last communication error

Queue

instantaneous queue length for messages waiting to be sent to agent. (Includes messages in the process of being sent)

MsgTo

message per second from all agents on this node to agent

MsgFrom

messages per second from agent to any agent on this node

eMJIP

effective Million Java Instruction Per Second that a single thread on the agent host.

mKbps

maximum kilobits per second for the network path between this node and the agent

eKbps

expected kilobits per second for the network path between this node and the agent

Cougaar

Cougaar Agent Architecture Open-Source site

Sensors

Adding New Metrics to the Metrics Service

The Metrics Package has helper classes for collecting raw sensor values, converting them into metrics, adding them into the path model, and displaying them in a servlet. In this section, we explain how these helper classes can be used to add new metrics to the Metrics Service.

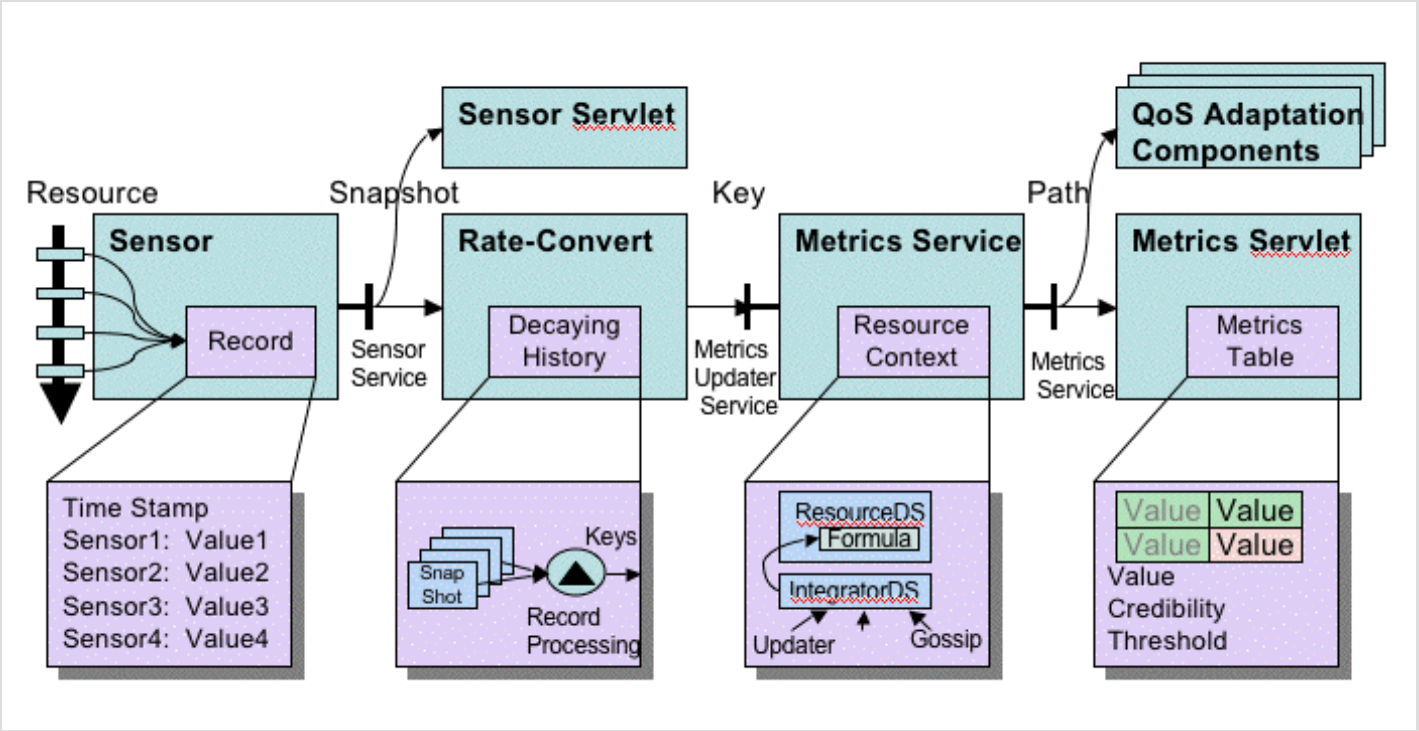
The Metric Service gets its data from several sources and integrates them into a up-to-date picture of the environment in which the Node finds itself. There are three ways of originating data into the Metrics Service, these are:

- Sensor data from Cougaar code measures the internal consumption of resource by Agents and Nodes. For example, the number of messages sent by an Agent or the amount of CPU or memory consumed by an Agent
- Network Management Configuration Files are used to get information about network characteristics from external Network Management Systems.
- Host Level probes are used to measure the characteristics of the Node's host

All these sources of data use the same "patterns" for capturing and processing the data. The processing of Metrics can be described as a data flow, which is processed by different components. Each component does a different job in converting raw sensor data into usable metrics. The figure below represents the processing of data as it flows through the Metrics Service:

- **Sensors** instrument the system and store raw statistics records. The records can be accessed via a service, which takes a snapshot of the record.
- **Rate-Converters** gathers snapshots of sensor records and converts them into metrics. The metrics are published into the Metrics service.
- **Metrics Service** models the environment. Internal to the model, formulas can depend on the raw metrics that are published into the Metrics service. The formulas can be accessed from the metrics service using paths.
- **QoS Adaptive Components** use the Metrics Service environment model to adapt to the Node's processing to changes in the environment.
- **Servlets** are used to view both Metrics and raw sensor records.

The patterns used to create these components are described in the following sections. Some explanations of this process are more easily understood when referencing specific pieces of the code, which will be highlighted in green for emphasis.



Sensor

Sensors have probes in the critical path of the basic operation of the Node. The probes must have very little overhead, so they have limited functionality. Probes only have time to detect that an event has happened and tally the results in a statistics record. The record holds many raw sensor values. There are 3 kinds of raw sensors values, which can reside in either the code or host probes. which are:

- ***Gauge***: This is an instantaneous value, such as utilization, current resource capacity.
- ***Counter***: Continuously increments some value like MsgsIn/Out or BytesIn/Out. org.cougaar.mts.std.AgentStatusAspect is an example of such a Sensor, continuously counting the MsgIn/Out & BytesIn/Out for local agents of a node, and the sums for the node itself. org.cougaar.core.qos.metrics.AgentLoadServlet is one gui where they user can observe this data.
- ***Integrator***: Counter that is a weighted summation of delta t * the value in that period. Integrators are used find the average value of gauge values over a time period. For example, Agent's CPU consumption is measured by the average number of threads used to process the Agent's code. (This is called the Agent's Load Average). org.cougaar.core.thread.AgentLoadSensorPlugin is an example of such a Sensor, continuously counting the MsgIn/Out & BytesIn/Out for local agents of a node, and the sums for the node itself.

An external clients of the Sensor component can sample the statistics record using a service offered by the Sensor. When a client samples data, it takes a **Snapshot** at a point in time, resulting in a collection of these snapshots which need to be processed to create usable Metrics. Notice the processing of the raw statistics is not done as part of the critical path, but done by an independent component with its own threads. The processing of snapshots is the subject of the next section.

Rate Conversion

Rate converter are clients to a sensor service and periodically take snapshots of the sensors record. The two snapshots

are processed to create meaningful metrics. The time between snapshots defines the averaging interval for the metric. The processing is different for each field in the record, depending on the kind of sensor value. For example to calculate rate of messages being sent by an agent, the sensor value for the count of messages from the agent is needed. The change in count is subtracted from two snapshots and the change in time is subtracted from the snapshot timestamps. The message rate is calculated by dividing the change in count by the change in time.

Rate converters have the following internal structure:

- ***Record Processing***: Data snapshots are collated and computed as a rolling average of changes in time & value. A Decaying History object is used to store the snapshots and call the record processing.
- ***Decaying History***: Is an object which coordinates the storage and processing of snapshots. It maintains a decay store of snapshots and call the processing of data snapshots for several averaging periods. (10, 100, 1000-second averages are common).
  org.cougaar.core.qos.metrics.AgentStatusRatePlugin
  is one plugin which implements the DecayingHistory's newAddition(), which updates the Metrics Service with the processed value.
- ***Key generation***: After the snapshots are processed and represent a value per some time duration, a Key is generated in relation to the rolling average of time, and used to insert this Metric into the Metrics Service.

Metrics Service Models

The Metrics Service maintains a model of the environment for the Cougaar society. The model includes contexts for the Hosts, Nodes, and Agents in the society. Each Node maintains its own instance of the model and updates it using Metrics published into the Metrics Updater Service.

- **Resource Context**: model a specific entity in the environment. Resource Contexts can contain child contexts. The children inherit all the attributes of its parent. For example, an Agent context is a child of Node, which is a child of Host. The Agent has a CPU count attribute that it inherits from its Host.
  org.cougaar.core.qos.rss.AgentDS is an example of a Resource
  Context.
- **Formula**: calculate attributes of a context. Formulas may depend on other formulas or raw Metrics published to the Metrics service. The org.cougaar.core.qos.rss.DecayingHistoryFormula is a helper
  class which create formulas that get their values directly from Metrics published by a Decaying History object.
- **IntegatorDS** integrate published metrics from many DataFeeds into a single value. Resource Contexts Formulas get the values for published Metrics by subscribing to Integrator formulas, instead of getting the Metrics directly from the feeds themselves. The IntegratorDS is part of the QuO RSS, which is outside of
  the Cougaar code.

Metrics Servlet

Evaluating the Metrics Service & Observing the system is usually done through GUIs in the Cougaar system, using Cougaar Servlets. In the Metrics Service, there resides many differnent servlets pertaining to communications, resource constraints, etc., please refer to Operation Using Servlets for the different kinds of servlets currently available.

Metrics Servlets all extend a Metrics Servlet template, which is formatted to make easy interpretation of the Metrics with tables, background & text colors. Here's a quick guide to interpreting a Metrics Servlet: Refer to the servlet page.

- **query**: This is a servlet window into the Metrics Service, or when a servlet is queried it reveals a current snapshot of those current Metrics Service values.
- **Value**: Dislayed as the base text, this is the Metrics name (e.g. AgentA Load)
- **Credibility**: Displayed as the color of the text: light gray to indicate that the metric value was only determined as a compile-time default; gray to indicate that the metric value was obtained from a configuration file; black to indicate that the metric value was obtained from a run-time measurement. A metric's credibility metric is an approximate measure of how much to believe that the value is true. Credibility takes into account several factors, including when, how, by whom a measurement was made.
- **threshold**: Displayed in the credibility, and when this this certain Metric's value crosses a threshold, it may be interesting enough to warrant attention. This is shown by the color of the background. A green background indicates that the value is in the normal range. A yellow background indicates that the values is in a typical ground state, i.e nothing is happening. A red background indicates that the value has crossed a metric-specific threshold and may be interesting.

Sensor Servlet

Sensor Servlets allow viewing of a raw sensor servlet. The main use of sensor servlets is for debugging.

QoS Adaptive Components

QoS Adaptive components subscribe to Metrics using the Metrics Service. How to create QoS Adaptive Components is out of scope for this section.

Constants

The same name for a metric must be used throughout all components of the metrics service. To aid consistency, we recommend using the metrics constants. The list of constants is documented in org.coguaar.core.qos.metrics.Constants

- **Metrics Name**: Quite obviously, the name of the Metric, as it applies to its measurement & key generation. Ususally, CPU_LOAD_AVG, CPU_LOAD_MJIPS, MSG_IN, MSG_OUT, BYTES_IN, BYTES_OUT
- **Averaging intervals**: Used in the Key of a Metric, referring to the averaging interval of value to time of that value. The current intervals are: 1_SEC_AVG, 10_SEC_AVG, 100_SEC_AVG, 1000_SEC_AVG.

Cougaar

Cougaar Agent Architecture Open-Source site

Servlets

Servlets

Cougaar includes optional servlet components to display metrics data in a browser. These servlets can be used to:

- View the real-time performance of the society, such as CPU load and message traffic,
- Debug a running application, for example to find blocked messages, and
- Illustrate more complex examples of the basic metrics service usage patterns.

The servlets are documented in more detail below. Here is a brief summary of the servlets, in order of most common use by developers:

- /metrics/agent/load [snapshot]:
  Current top-level view of the CPU load, message traffic, and persistence activity of all agents on the node.
- /metrics/host/resources [snapshot]:
  Current node resource usage (load average, sockets, heap size, etc)
- /metrics/remote/agents [snapshot]:
  Current message traffic to/from all agents on the node, including the most recent communication time and the number of queued messages.
- /threads/top [snapshot]:
  Although not a metrics servlet per-se, the thread service's "top" servlet displays running and queued pooled threads.
- /message/statistics [snapshot]:
  Cumulative message traffic by the node (aggregate of all agents on that node), plus a message size histogram.
- /message/between-Any-agent-and-Local-Agent?agent=*AGENT* [snapshot]:
  Cumulative message traffic by the specified local agent to/from any target.
- /message/between-Node-and-Agent?agent=*AGENT* [snapshot]:
  Cumulative message traffic by the node (aggregate of all agents on that node) to/from the specified target agent.
- /metrics/query?paths=*PATHS*:
  Read raw metric(s) as XML.
- /metrics/writer?key=*KEY*&value=*VALUE*:
  Write a raw metric into the metrics service.

The rest of this section describes each servlet in more detail:

/message/between-Any-agent-and-Local-Agent

This servlet displays status of communications from an agent on this node to ALL other agents in the society. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service. Click here for a sample display.

/message/between-Node-and-Agent

This servlet displays status of communications from ALL agents on this node to a specified agent. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service. Click here for a sample display.

/message/statistics

This servlet summarizes the Messages statistics for communications out of all agents on this node. This is a raw dump of the legacy MTS Message Statistics Service and does not use the Metrics service. Click here for a sample display.

/metrics/agent/load

This servlet shows the amount of resource consumption for each agent and service that is resident to this node. The resources include CPU, Communications, and Storage. This servlet is used to see which agents are resident on the node and their level of activity. The metrics are all average rates over the averaging interval. Click here for a sample display.

/metrics/host/resources

This servlet shows the status of Host resources for the Node. The most of basic values come from polling Linux /proc. MJIPS (Million Java Instructions per Second) comes from running benchmark. Click here for a sample display.

/metrics/query

This servlet allows the operator to query the MetricsService directly. The result can either be displayed as a web page as XML or returned to the invoker as serialized Java HashMap, depending on the value of the `format` uri argument. One or more query paths should be supplied as the value of the `paths` uri argument, with `|` as the separator.Usage:
“http://localhost:8800/$nodename/metrics/query?format=xml&paths=Agent(3-69-ARBN):Jips|Agent(3-69-ARBN):CPULoadJips10SecAvg”The ‘format’ argument is optional, but if left out defaults to xml return of metric data to the browser.
An optional Java version of a metrics query client was written and resides in core/examples/org/cougaar/core/examples/metrics/ExampleMetricQueryClient, returning a hashmap of path values from the query-specified node.

/metrics/remote/agents

This servlet shows the status of resources along the path for communications from any agent on the node to a specific agent. This servlet is useful for debugging. For example, if the Queue length is greater than one, then messages are backed up waiting to be transmitted to the agent. Since messages are usually sent right away

this indicates a problem along the path. Likewise, if he Node has not HeardFrom an agent recently, the agent or its node may have failed. Also, the table show the capacity of the network path to the agent and the agent's host capacity. Click here for a sample display.

/metrics/writer

This servlet allows the operator to write values into the MetricsService directly. The `key` and `value`should be supplied with uri arguments of the same name. For now the value must be parseable as a double. The metric will be entered with USER_DEFAULT_CREDIBILITY (0.3) and with the client host as the provenance.

Usage:

Specified by a prefix of protocol, host, port, nodename and path, followed by some key-value pair in the usual http::get parameter format.
e.g.:
"http://localhost:8800/$nodename/metrics/writer/?
key=Site_Flow_10.200.2.0/24_10.200.4.0/16_Capacity_Max&value=5600"

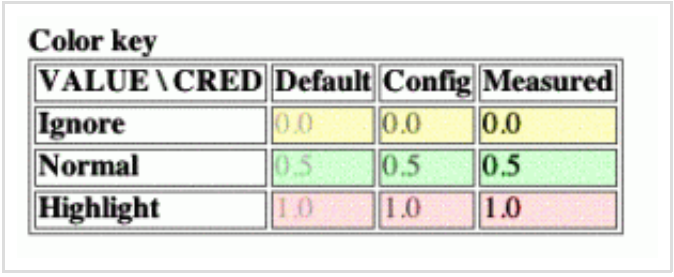
GUI Conventions

Several metrics servlets uses gui conventions to show three attributes of each metric. In addition, some are mouse sensitive: all the metric's attributes are displayed on the browser's documentation line.

The **value** of the metric is displayed as the base text.

The **credibility** of the metric is displayed as the color of the text: light gray to indicate that the metric value was only determined as a compile-time default; gray to indicate that the metric value was obtained from a configuration file; black to indicate that the metric value was obtained from a run-time measurement. A metric's credibility metric is an approximate measure of how much to believe that the value is true. Credibility takes into account several factors, including when, how, by whom a measurement was made.

When the value of a metric crosses a **threshold**, its value may be interesting enough to warrant attention. This is shown by the color of the background. A green background indicates that the value is in the normal range. A yellow background indicates that the values is in a typical ground state, i.e nothing is happening. A red background indicates that the value has crossed a metric-specific threshold and may be interesting.



Cougaar

Cougaar Agent Architecture Open-Source site

Use Cases

Use Cases and Examples

Example Metrics Reader

An example Metrics Service subscriber can be found in the core module.

```
package org.cougaar.core.qos.metrics;

/**
 * Basic Metric Service Client subscribes to a Metric path given in
 * the Agent's .ini file and prints the value if it ever changes
 */
public class MetricsClientPlugin
    extends ParameterizedPlugin
    implements Constants
{
    protected MetricsService metricsService;
    private String paramPath = null;
    private VariableEvaluator evaluator;

  /**
   * Metric CallBack object
   */
    private class MetricsCallback implements Observer {
      /**
       * Call back implementation for Observer
       */

      public void update(Observable obs, Object arg) {
            if (arg instanceof Metric) {
                Metric metric = (Metric) arg;
                double value = metric.doubleValue();
                System.out.println("Metric "+ paramPath +"=" + metric);
            }
        }
    }

  public void load() {
```

```
    public void load() {
      super.load();
      ServiceBroker sb = getServiceBroker();

      evaluator = new StandardVariableEvaluator(sb);

      metricsService = ( MetricsService)
        sb.getService(this, MetricsService.class, null);

      MetricsCallback cb = new MetricsCallback();
      paramPath = getParameter("path");
      // If no path is given, default to the load average of the
      // current Agent.
      if (paramPath == null)
          paramPath ="$(localagent)"+PATH_SEPR+"LoadAverage";

      Object subscriptionKey=metricsService.subscribeToValue(paramPath, cb,
                                                              evaluator);
    }
  }
```

Example Metrics Writer

An example Metric Service writer has the following form:

```
package org.cougaar.core.qos.metrics;

public class MetricsTestComponent
    extends SimplePlugin
    implements Constants
{

    public void load() {
        super.load();

        ServiceBroker sb = getServiceBroker();
        MetricsUpdateService updater = (MetricsUpdateService)
            sb.getService(this, MetricsUpdateService.class, null);

        String key = "Current" +KEY_SEPR+ "Time" +KEY_SEPR+ "Millis";
        // Publish the current time
        Metric m = new MetricImpl(new Long(System.currentTimeMillis()),
                                  SECOND_MEAS_CREDIBILITY,
                                  "ms",
                                  "MetricsTestComponent");
        updater.updateValue(key, m);
    }
}
```

Cougaar

Cougaar Agent Architecture Open-Source site

Using

Usings Servlets

Cougaar includes optional servlet components to display metrics data in a browser. These servlets can be used to:

- View the real-time performance of the society, such as CPU load and message traffic,
- Debug a running application, for example to find blocked messages, and
- Illustrate more complex examples of the basic metrics service usage patterns.

The servlets are documented in more detail below. Here is a brief summary of the servlets, in order of most common use by developers:

- /metrics/agent/load:
  Current top-level view of the CPU load, message traffic, and persistence activity of all agents on the node.
- /metrics/host/resources:
  Current node resource usage (load average, sockets, heap size, etc)
- /metrics/remote/agents:
  Current message traffic to/from all agents on the node, including the most recent communication time and the number of queued messages.
- /threads/top:
  Although not a metrics servlet per-se, the thread service's "top" servlet displays running and queued pooled threads.
- /message/statistics:
  Cumulative message traffic by the node (aggregate of all agents on that node), plus a message size histogram.
- /message/between-Any-agent-and-Local-Agent?agent=*AGENT*:
  Cumulative message traffic by the specified local agent to/from any target.
- /message/between-Node-and-Agent?agent=*AGENT*:
  Cumulative message traffic by the node (aggregate of all agents on that node) to/from the specified target agent.
- /metrics/query?paths=*PATHS*:
  Read raw metric(s) as XML.
- /metrics/writer?key=*KEY*&value=*VALUE*:
  Write a raw metric into the metrics service.

The rest of this section describes each servlet in more detail:

/message/between-Any-agent-and-Local-Agent

This servlet displays status of communications from an agent on this node to ALL other agents in the society. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service. Click here for a sample display.

/message/between-Node-and-Agent

This servlet displays status of communications from ALL agents on this node to a specified agent. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service. Click here for a sample display.

/message/statistics

This servlet summarizes the Messages statistics for communications out of all agents on this node. This is a raw dump of the legacy MTS Message Statistics Service and does not use the Metrics service. Click here for a sample display.

/metrics/agent/load

This servlet shows the amount of resource consumption for each agent and service that is resident to this node. The resources include CPU, Communications, and Storage. This servlet is used to see which agents are resident on the node and their level of activity. The metrics are all average rates over the averaging interval. Click here for a sample display.

/metrics/host/resources

This servlet shows the status of Host resources for the Node. The most of basic values come from polling Linux /proc. MJIPS (Million Java Instructions per Second) comes from running benchmark. Click here for a sample display.

/metrics/query

This servlet allows the operator to query the MetricsService directly. The result can either be displayed as a web page as XML or returned to the invoker as serialized Java HashMap, depending on the value of the `format` uri argument. One or more query paths should be supplied as the value of the `paths` uri argument, with `|` as the separator.Usage:

```
http://localhost:8800/$nodename/metrics/query?format=xml&paths=Agent(3-69-ARBN):Jips|Agent(3-69-ARBN):CPULoadJips10SecAvg
```

The ‘format’ argument is optional, but if left out defaults to xml return of metric data to the browser.
An optional Java version of a metrics query client was written and resides in core/examples/org/cougaar/core/examples/metrics/ExampleMetricQueryClient, returning a hashmap of path values from the query-specified node.

/metrics/remote/agents

This servlet shows the status of resources along the path for communications from any agent on the node to a specific agent. This servlet is useful for debugging. For example, if the Queue length is greater than one, then

messages are backed up waiting to be transmitted to the agent. Since messages are usually sent right away this indicates a problem along the path. Likewise, if he Node has not HeardFrom an agent recently, the agent or its node may have failed. Also, the table show the capacity of the network path to the agent and the agent's host capacity. Click here for a sample display.

/metrics/writer

This servlet allows the operator to write values into the MetricsService directly. The `key` and `value`should be supplied with uri arguments of the same name. For now the value must be parseable as a double. The metric will be entered with USER_DEFAULT_CREDIBILITY (0.3) and with the client host as the provenance.

Usage: Specified by a prefix of protocol, host, port, nodename and path, followed by some key-value pair in the usual http::get parameter format.

Example: `http://localhost:8800/$nodename/metrics/writer/?key=Site_Flow_10.200.2.0/24_10.200.4.0/16_Capacity_Max&value=5600`

GUI Conventions

Several metrics servlets uses gui conventions to show three attributes of each metric. In addition, some are mouse sensitive: all the metric's attributes are displayed on the browser's documentation line.

The **value** of the metric is displayed as the base text.

The **credibility** of the metric is displayed as the color of the text: light gray to indicate that the metric value was only determined as a compile-time default; gray to indicate that the metric value was obtained from a configuration file; black to indicate that the metric value was obtained from a run-time measurement. A metric's credibility metric is an approximate measure of how much to believe that the value is true. Credibility takes into account several factors, including when, how, by whom a measurement was made.

When the value of a metric crosses a **threshold**, its value may be interesting enough to warrant attention. This is shown by the color of the background. A green background indicates that the value is in the normal range. A yellow background indicates that the values is in a typical ground state, i.e nothing is happening. A red background indicates that the value has crossed a metric-specific threshold and may be interesting.

Color key

VALUE \ CRED	Default	Config	Measured
Ignore	0.0	0.0	0.0
Normal	0.5	0.5	0.5
Highlight	1.0	1.0	1.0

VariableEvaluator

org.cougaar.core.qos.metrics.VariableEvaluator

Path specifications in the Metrics Service support a limited notion of variables for which actual values are substituted dynamically. The syntax is taken from make: `$(var)`. The evaluation of such variables is handled by a user-supplied `VariableEvaluator` instance. The parsing and substitution is handled by the Metrics Service.

A useful base-class for classes which implement this interface `StandardVariableEvaluator`, which can supply values for the variables `localagent`, `localnode` and `localhost`.

`String evaluateVariable(String var)`

> Given a variable reference in a path this method should return the String to be spliced in as a replacement. The argument is the variable name itself, without the $ or the parens.