OpenNMS Developer Guide

OpenNMS internals

2001

Preface

OpenNMS is the creation of numerous people and organizations, operating under the umbrella of the OpenNMS project. The original code base was developed and published under the GPL by the Oculan Corporation until 2002, when the project administration was passed on to Tarus Balog.

The current corporate sponsor of OpenNMS is The OpenNMS Group, which also owns the OpenNMS trademark.

OpenNMS is a derivative work, containing both original code, included code and modified code that was published under the GNU General Public License. Please see the source for detailed copyright notices, but some notable copyright owners are listed below:

- Copyright © 2002-2014 The OpenNMS Group, Inc.
- Original code base for OpenNMS version 1.0.0 Copyright © 1999-2001 Oculan Corporation.
- Mapping code Copyright © 2003 Networked Knowledge Systems, Inc.
- ScriptD code Copyright © 2003 Tavve Software Company.

Please send any omissions or corrections to this document to Tarus Balog.

Chapter 1. Chapter 1

Introduction to OpenNMS

1.1. What is OpenNMS?

<insert text>

1.2. Why OpenNMS?

<insert text>

1.3. Features

<insert text>

1.4. Why Develop for OpenNMS?

<insert text>

Chapter 2. Chapter 2

Technologies

2.1. Java 2

<insert text>

2.2. XML/XSL

<insert text>

2.3. Servlets

<insert text>

2.4. JSPs

<insert text>

2.5. RDBMS

<insert text>

2.6. JMS

<insert text>

2.7. Castor

<insert text>

2.8. JoeSNMP

<insert text>

2.9. RRDTool

<insert text>

Chapter 3. Chapter 3

Functional Product Description

3.1. Discovery

<insert text>

3.2. CAPSD

<insert text>

3.3. Eventd

<insert text>

3.4. actiond

<insert text>

3.5. notifd

<insert text>

3.6. Scheduler

<insert text>

3.7. Service Monitor (pollers)

<insert text>

3.8. RTC

<insert text>

3.9. Outage Manager

<insert text>

3.10. Data Collection

<insert text>

3.11. Reporting

<insert text>

3.11.3.11.1. Performance Reports

<insert text>

3.11.3.11.2. Availability Reports

<insert text>

3.12. Web UI

<insert text>

Chapter 4. Chapter 4

Extending OpenNMS

4.1. Customizing SNMP Data Collection

<insert text>

4.2. Adding Support for New Services

4.2.4.2.1. Introduction

The capabilities daemon (capsd) is responsible for scanning network interfaces found by the discovery daemon for the services/protocols they support and updating the database accordingly. Capsd will also periodically rescan managed interfaces to determine if a managed interface has had any additional services enabled since the last capability check.

The poller daemon is responsible for checking the status of each service on each managed interface at a regularly configured interval. If the status of the service has changed since the last poll an appropriate event is generated indicating the new status of the service on that interface.

OpenNMS provides a straight forward framework for extending the default set of services and protocols it can detecte and monitor. In order to extend OpenNMS to manage a custom service or protocol the following is required:

```
* Code a Capsd plugin capable of testing whether or
  not a network interface supports the desired protocol or service.
* Add a <protocol plugin> element defining the new service to the
  $BB_HOME/etc/capsd-configuration.xml config file.
* Code a Poller plugin capable of monitoring the current status
  of the desired protocol or service on a specified network interface.
* Add <service> and <monitor> elements defining the new service
  to be polled to the $BB_HOME/etc/poller-configuration.xml config
  file.

  - $BB_HOME refers to the OpenNMS install directory.
```

4.2.4.2.2. Creating a Capabilities Daemon (Capsd) Plugin

Writing the Plugin

Capsd uses plugins to perform capability checks on a device. A plugin is simply a Java class which implements the org.opennms.netmgt.capsd.Plugin interface. The following methods are defined in the interface and must be implemented:

```
public String getProtocolName()

 Simply returns the name of the service or protocol tested for by the
 plugin.  In the case of the FTP plugin the string "FTP" is returned.

public boolean isProtocolSupported(java.nnet.InetAddress address)

 Returns true if the device identified by the InetAddress parameter
 supports the protocol being tested.

public boolean isProtocolSupported(java.net.InetAddress address,
       java.util.Map properties)

 Returns true if the device identified by the InetAddress parameter
 supports the protocol being tested.  A second parameter, properties,
 is of type java.util.Map and provides a mechanism for overriding
 default configuration options such as timeouts, retries and port
 information.
```

NOTE: For an example plugin take a look at the FTP plugin in src/services/org/opennms/netmgt/capsd/FtpPlugin.java.

At runtime the capabilities daemon calls the isProtocolSupported() method of each loaded plugin passing it the java.net.InetAddress object of each interface found by the discovery daemon. Any services found to be supported on the interface will cause an entry to be added to the 'ifservices' table keyed by the interface's node identifier and IP address.

Plugin Integration

During initialization, the capabilities daemon reads the capsd-configuration.xml config file and inserts any new services into the 'services' table in the database. Further, during initialization, the capabilitiees daemon uses the plugin class name information defined in capsd-configuration.xml to load each of the plugins.

To add a new service edit capsd-configuration.xml and add a new <protocol plugin> element. Within the <protocol plugin> element is defined the service name, plugin class name, and any plugin specific properties.

Consider the following capsd-configuration.xml entry for the FTP service:

```
<protocol-plugin protocol="FTP"  class-name="org.opennms.netmgt.capsd.FtpPlugin" scan="on">
 <property key="userid" value="ftp"/>
 <property key="password" value="anonymous@"/>
</protocol-plugin>
```

The "userid,ftp" and "password,anonymous@" name-value pairs are passed to the plugin via the java.util.Map parameter of the isProtocolSupported() method described earlier.

4.2.4.2.3. Creating a Poller Plugin

Writing the Poller Plugin

The poller daemon uses poller plugins for polling managed interfaces for the current status of the services supported by the interface. A poller plugin is a Java class which implements the org.opennms.netmgt.poller.monitors.ServiceMonitor interface. The interface defines the following methods which must be implemented in the plugin:

```
public void initialize(java.util.Map parameters);

 Called by the poller daemon following instantiation of the
 plugin during startup.  If during initialization the plugin detects
 a critical error (such as a missing library) it can throw a
 java.lang.RuntimException.  If the plugin throws an exception during
 initialization the poller daemon will disable the plugin.

public void release();

 Called by the poller daemon during shutdown.  This provides a
 mechanism for the plugin to release any acquired resources.

public void initialize(org.opennms.netmgt.poller.monitors.NetworkInterface iface);

 Called by the poller daemon whenever the poller learns of a new
 interface which supports the service polled by the plugin.
 If desired, configuration information can be associated with the
 interface at this time prior the poller actually scheduling
 the interface.  If the plugin throws an exception during interface
 initialization the poller daemon will log an error and discard
 the interface.

public void release(java.net.NetworkInterface iface);

 Called by the poller daemon whenever an interface is being removed
 from the scheduler. For example, if a service is determined as being
 no longer supported by an interface then this method will be invoked
 to cleanup any information associated with that interface. This
 gives the implementor of the interface the ability to serialize any
 data prior to the interface being discarded.  If an exception is
 thrown during the release the exception will be logged, but the
 interface will still be discarded for garbage collection.

public int poll(java.net.NetworkInterface iface,
    org.opennms.netmgt.utils.EventProxy eproxy,
    java.util.Map parameters);

 Called by the poller daemon each time an interface requires a check
 to be performed as defined by the poller scheduler.  The poll()
 method is passed the interface to check, a reference to an
```

```
required by the plugin aside from the standard available/unavailable
generated by the poller daemon.  Additionally, a java.util.Map
is passed which may be used to access any service specific
configuration values as defined in the poller-configuration.xml
config file.
```

Poller Plugin Integration

During initialization, the poller daemon reads the poller-configuration.xml config file and uses the plugin class name information defined in <monitor> element blocks to load each of the plugins. The parameters associated with each service as defined in <service> elements within the configuration file are also read and used by the poller to determine how often a particular service is to be polled for the services which fall within the enclosing package.

To add a new service edit poller-configuration.xml and add a new <service> element to the desired package. Within the <service> element define the service name, how often it should be scheduled (interval), and a list of any parameters needed by the plugin defined within <parameter> elements. The configured parameters will be passed to the plugin via the java.util.Map object parameter of the poll() method. Typically such things as the port on which to test the service as well as timeout and retry information is defined here.

Consider the following poller-configuration.xml <service> entry for the FTP service:

```
<service name="FTP" interval="300000">
 <parameter key="timeout" value="3000"/>
 <parameter key="port" value="21"/>
 <parameter key="userid" value="ftp"/>
 <parameter key="password" value="anonymous@"/>
</service>
```

Next the class name of the poller plugin must be defined. This is done by adding a new <monitor> element to the poller-configuration.xml file. Within this element the name of the service and the class name of the poller plugin are specified.

Consider the following poller-configuration.xml <monitor> entry for the FTP service:

```
<monitor service="FTP"   class-name="org.opennms.netmgt.poller.monitors.FtpMonitor"/>
```

Configuration File Factories

Accessing the OpenNMS Database

OpenNMS database configuration information is specified in the $BB_HOME/etc/opennms-database.xml file. Defined within this configuration file are the OpenNMS database name, userid, and password. If database access is a requirement for a poller the class org.opennms.netmgt.config.DatabaseConnectionFactory may be used to get a database connection.

The DatabaseConnectionFactory class is a singleton class used to load the database configuration information. The class provides a convenience method for retrieving a connection to the database. The following code snippet illustrates how to use the DatabaseConnectionFactory class to load the database config and retrieve a connection.

```
/********* Begin *********/
...
import java.sql.*;
import org.opennms.netmgt.config.DatabaseConfigFactory;
...
public void initialize(NetworkInterface iface)
{
 // Get connection to the database so we can
 // retrieve information pertaining to the
```

```
 // passed interface
 //
 try
 {
  DatabaseConnectionFactory.init();
 }
 catch (Throwable e)
 {
  // Log error
  ...
  // Throw exception
  throw new RuntimeException("Database connection factory
    initialization failed, reason: " + e.getLocalizedMessage());
 }

 try
 {
  java.sql.Connection dbConn = DatabaseConnectionFactory.getInstance().getConnection();
 }
 catch (SQLException sqlE)
 {
  // Log error
  ...
  // Throw exception
  throw new RuntimeException("Unable to get connection to the database,
    reason: " + sqlE.getLocalizedMessage());
 }

 // Do stuff with the database connection
 ...

 // Close the connection
 try
 {
  dbConn.close();
 }
 catch (SQLException sqlE)
 {
  // Log the error
  ...
 }
 ...
}
/******** End **********/
```

NOTE: For additional example code which uses the DatabaseConnectionFactory take a look at the SNMP poller plugin: src/services/org/opennms/netmgt/poller/SnmpMonitor.java

Accessing SNMP Configuration Information

SNMP configuration information is specified in the $BB_HOME/etc/snmp-config.xml file. Defined within this configuration file are read/write community strings, retries, timeout, and SNMP version information specific to the OpenNMS installation.

If an SNMP-based poller is being written the class org.opennms.netmgt.config.SnmpPeerFactory may be used retrieve a JoeSNMP org.opennms.protocols.snmp.SnmpPeer object for a particular interface's IP address based on the content of the snmp-config.xml file. If a specific entry in snmp-config.xml cannot be found a default SnmpPeer object will be returned. An SnmpPeer object encapsulates the SNMP configuration information associated with a particular IP address and is used to construct an SNMP session (org.opennms.protocols.snmp.SnmpSession) with the remote node. Refer to the JoeSNMP documentation for further information on SNMP peer objects and sessions.

The SnmpPeerFactory class is a singleton class used to load the SNMP configuration information. The class provides a convenience method for retrieving an SnmpPeer object for a provided network address.

The following code snippet illustrates how to use the SnmpPeerFactory class to load the SNMP config and retrieve an SnmpPeer object for a specified network address.

```
/********* Begin *********/
...
import org.opennms.protocols.snmp.SnmpPeer;
import org.opennms.protocols.snmp.SnmpSession;
import org.opennms.netmgt.config.SnmpPeerFactory;
...
public void initialize(NetworkInterface iface)
{
 // Initialize the SNMP peer factory and use it
 // to construct an SNMP peer object for the
 // passed interface.
 //
 try
 {
  SnmpPeerFactory.init();
 }
```

```
  catch (Throwable e)
  {
   // Log error
   ...
   // Throw exception
   throw new RuntimeException("SNMP peer factory
     initialization failed, reason: " + e.getLocalizedMessage());
  }

  SnmpPeer peer = SnmpPeerFactory.getInstance().getPeer((InetAddress)iface.getAddress());

  // Use SnmpPeer object to initialize a JoeSNMP SNMP session
  //
  try
  {
   SnmpSession session = new SnmpSession(peer);
  }
  catch (SocketException sE)
  {
   // Error Handling
   ...
  }
  ...
  // Use SNMP session to query the remote node's MIB
  //
  ...

  // Close the session
  try
  {
   session.close();
  }
  catch (Throwable E)
  {
   // Log the error
   ...
  }
  ...
 }
 /******** End **********/
```

NOTE: For additional example code which uses the SnmpPeerFactory take a look at the SNMP poller plugin: src/services/org/opennms/netmgt/poller/SnmpMonitor.java

4.3. Web UI Design

4.3.4.3.1. Concepts You Will Need:

To get the most from this document and the WebUI code, you will need to be familiar with the following subjects: Java, Servlet, and JSP programming HTML and XML Servlet 2.3 and JSP 1.2 specifications Tomcat 4.0 setup and configuration HTTP methods, error codes, and headers

4.3.4.3.2. Overall Architecture

The basic idea behind our architecture is to separate the logic from the presentation as much as possible. This facilitates rapid feature addition, rapid feature redesign, and long-term maintenance.

To meet this goal, we use three distinct tools, each described in more detail below:

- **models.** non-visual objects for non-visual work like reading and writing to the database
- **JSPs.** generate HTML or XML to display to the user based on information retrieved by the models
- **servlets.** use the models to make changes to the system and then use the JSPs to display results to the user

All logic for database activity and performance data querying are encapsulated in non-visual models. Models allow reuse of the same code across several different user interface components. There are many good examples of this throughout the WebUI code. A few are the PerformanceModel, the RealTimeDataModel, and the OutageModel. Each of these models are used by three or more servlets or JSPs. Models are not allowed to generate visual content; they are for non-visual work only.

To create HTML or XML to display to the user, we use JSPs. They make simple calls to the models (and thus the database) to retrieve data to plug into their HTML or XML to display to the user. No

HTML or XML is generated outside of a JSP, and a JSP is not allowed to make changes to the database or other data. They are for display only.

To allow the user to make changes to data or other features, we write servlets that respond to an HTTP POST. The servlet code first takes the necessary action (in most cases changing the database or sending an event) and then redirects the request (with the results of the action) to a JSP for HTML generation and display.

An exception to the rule above is if a URL must return an output type other than HTML or XML (like an image or PDF), then we implement the URL as a servlet. In this context, the servlet acts like a JSP in the fact that it is purely used for creation of content for display.

4.3.4.4.3. Servlet Forwards and Includes versus HTTP Redirects

Forwards and includes are servlet/JSP mechanisms that allow Tomcat (or any servlet container) to ferry requests around different servlets and JSPs to generate the correct content before returning it to the user. Forwards and includes can be nested as deep as necessary, and they do not leave the web server until the content is fully generated. We often use forwards in servlets that handle only GET requests, and we often use includes in JSPs to standardize the content. For example, the header.jsp and footer.jsp are included in each JSP.

HTTP Redirects are actually a two-request process. The web browser makes the first request which is almost always a POST to a servlet. Then the servlet does something (like writes to a database or sends an event) and then sends a redirect response (HTTP response code 301 or 307) back to the web browser with a new URL. The second request is then made by the web browser to the URL in the redirect response. This second request is usually a GET request to a JSP. The first request does the work, and the second request displays the results.

Redirects are useful after using a POST, because if the user uses the back button or the refresh button, they will not get a cryptic error from the web browser because it does not want to (or cannot) resend the POST. This creates a much more seamless user experience, and is easier to code. The logic is in the servlet -- pure Java code, and the output is in the JSP -- almost pure HTML.

4.3.4.4.4. Internal URLs versus External URLs

Internal URLs are used in forwards and includes by Tomcat. They are relative to the webapp context URL. In our case, they are relative to the /opennms URL. These URLs contain leading slashes. If they do not, they are considered to be a sibling of the current URL and will not work.

External URLs are used as links to other pages by the web browser. They are made relative to our URL base (see the org.opennms.web.Util.calculateBaseUrl method). These URLs do not use leading slashes. If they do, they are considered relative to the root URL and will not work.

You must follow the convention of setting up the HTML BASE tag in the HTML header on every page or the external URLs will not work. This at the very least will cause problems with loading images from included JSPs, and your page's header will display incorrectly. At worst, however, many or all of your HTML links will not work either.

4.3.4.4.5. Security Roles and Protected URLs

In the OpenNMS system, we use the BASIC HTTP authentication mechanism, and there are two major security roles. The default security role is the "OpenNMS User" role, and all users belong to that role. All URLs of the OpenNMS WebUI are protected at least by this default role.

Other security roles are configured in the WEB-INF/web.xml file and the /opt/OpenNMS/etc/magic-users.properties file. The web.xml file defines which URLs are protected by a role, and the magic-user.properties file defines which users are authorized in each role.

Another important user role is the "OpenNMS Administrator" role. All URLs under /opennms/admin/* are protected by this role. Currently only the special "admin" user is allowed access to this role.

4.3.4.3.6. Internal Structure of a JSP

A good example to look at is the web/web/element/node.jsp. One of the major goals in our JSP writing is to keep the Java code as far away as possible from the HTML content, and the format of our JSPs reflect this.

The JSP @page directive is first. Followed by the declaration for any class variables and the init method (if required). This declaration section usually defines and initializes the JSP's model which handles all non-visual code.

The scriptlet for the data-gathering code is next. This section is where all of your Java code that is not directly related to displaying HTML should go. It usually consists of a short check of the request parameters and then calling a model method or two. Remember, keeping your Java code in one area keeps the code portable in case you need to move it into a non-visual object or servlet. It also helps you fight the temptation to add logic throughout the body of your JSP. We try to keep our JSPs short and dumb.

Next is the HTML header. This section defines the page's title, base URL, and includes our global stylesheet. Please take notice of the HTML BASE tag, the org.opennms.web.Util.calculateBaseURL method, and the web/web/includes/styles.css stylesheet.

After the header is the body tag and then an include of our header JSP. This JSP include tag gives the header.jsp enough information to display itself properly, including the page's title, breadcrumbs to give the user an idea of where they are in the page hierarchy, and links to get back to earlier pages.

The body of our JSPs is a table that gives it a little indention on either side. Notice the empty columns with only non-breaking spaces ( ). When we completely move over to Cascading Style Sheets in a future release, we will use CSS to create indention.

In the body, use the data you collected in the main scriptlet to display here. Try to keep the Java code in this section to only JSP expressions, if conditionals, and for loops.

After the body is the footer include, which mirrors much of the header include, but without the breadcrumb information.

Then at the bottom, below the HTML, if you need any convenience methods for display purposes (creating HTML or URLs, but not reading the database or any other sort of non-visual logic), then put a JSP declaration section that creates these convenience methods.

4.3.4.3.7. Internal Structure of a Servlet

A good example servlet is the org.opennms.web.nodelabel.NodeLabelChangeServlet.

Our servlet structure is actually very similar to Java portions of our JSPs. The servlet has a model class variable that is initialized in the init method. Then the doGet or doPost method checks the request

parameters, does a database lookup or makes a database change through a model call, but then it breaks from the JSP structure by calling either a forward or a redirect instead of creating its own results content.

If the servlet is handling POST methods, it will use a redirect. If the servlet is using a GET method, it will use a forward.

4.3.4.3.8. Log4J Logging

In the WebUI, we use a dual servlet-logging and Log4J-logging system. Actually, we are in the process of moving to a Log4J-based logging scheme, but it is likely that there will always be some amount of logs going through the servlet logging API into Log4J.

Logging to Log4J through the servlet API is acheived by the use of a Tomcat-specific logger class that wrappers a Log4J category. Using the servlet API instead of Log4J may be more intuitive to servlet/JSP programmers who are not familiar with Log4J, but using Log4J gives us many more features. Log4J gives us control over what level of logging is used; whether a given log message is a debug message, an informative message, a warning, or an error message; how often the log files should roll; and how large the log files should grow. Log4J can also send logs to other destinations other than text files. It can also add various timestamps and log adornments through configuration files without changing code.

Currently the web logs end up in one of two log files, but the file count will probably increase as we adopt Log4J throughout more of the WebUI and create finer-grained Web log categories. Almost all WebUI logs today end up in the web.log file in the OpenNMS log directory (historically one of /var/log/opennms or /opt/OpenNMS/log). The only exceptions are authentication logs messages which end up in the webauth.log file.

4.3.4.3.9. Performance Reporting System

The SNMP polling system gathers information and stores it in round-robin databases (RRD files) which are named in a convention that uniquely identifies the interface from which the data was collected. The WebUI then queries the RRD file directory to see which interfaces have data collected, and then presents that list to the user. The user selects an interface from the list, and the corresponding RRD is queried to find out what information was collected. Different types of network devices yield different types of performance data, so our reporting system is designed to be flexible. From the list of data collected, a list of prefabricated reports is created and presented to the user.

The user then, having selected an RRD, a prefabricated report, and a date range, requests the report. The WebUI sends the report parameters to the RRDGraphServlet which executes a command-line program called rrdtool (see http://www.rrdtool.org). The rrdtool reads the RRD information and creates a graph in a PNG image. That image is sent back to the user to view.

The prefabricated reports are actually just command-line options to give to the rrdtool to create a graph. They are read from the rrdtool-graph.properties file in the /opt/OpenNMS/etc directory.

The user is also able to create a "custom" or "ad hoc" performance report by picking and choosing the data sources inside the RRD to query. The user works through a wizard that allows him to choose and name each data source, give the graph a name, and even choose in which colors each data source should graph. This uses some templates from the rrdtool-graph.properties file, but all the values are filled in the template from the users choices. Then the same RRDGraphServlet serves up the resulting PNG graph image.