Mars 2.2.6 Hacker's Guide

Brian Trammell
`brian@lfrd.com`
Leapfrog Research and Development, LLC

March 1, 2004

Abstract

Describes technical aspects of the Mars network status monitor, and provides a guide for programmers wishing to extend Mars through the probe extension and plugin frameworks.

Contents

1 The Mars Configuration File

Though Mars was designed to be quick and easy to set up, using the Swing user interface to configure many services on many hosts can get tedious, especially if each host has many similar services. In this case, creating a Mars configuration file directly can be useful.

```
<mars:model xmlns:mars="http://www.altara.org/mars/xmlns/model/">
  <mars:hostlist>
    <mars:host
        name="Display name"
        address="DNS name or IP address">
      <mars:service
          name="Display name"
          port="TCP/UDP port"
          svctype="http|smtp|ftp|imap|pop3|ssh|tcp-connect"
          timeout="timeout in milliseconds"
          period="minimum wait time in milliseconds"
          notac="number of attempts before notification, 0-3">
        <mars:parameter name="name">value</mars:parameter>
        ... additional parameters, if any ...
        ... currently, only svctype http has a parameter, url,
        ... for the path to check on the web server ...
      </mars:service>
      ... additional services ...
    </mars:host>
    ... additional hosts ...
  </mars:hostlist>
  ... plugin config elements, currently: ...
  <mars:swingnotify
      enabled="true or false"
      notifyBackUp="notify service back up, true or false"
      beep="audible notification, true or false"/>
  <mars:mailnotify
      enabled="true or false"
      notifyBackUp="notify service back up, true or false"
      server="DNS name or IP address of SMTP server to use"
      address="List of comma-separated recipient addresses"
      fromAddress="Origin address (optional)"/>
   <mars:csvlog
      enabled="true or false"
      logfile="path to logfile"/>
</mars:model>
```

Figure 1: Mars configuration file structure

```
<mdef:definition xmlns:mdef="http://www.altara.org/mars/xmlns/def/">
  <mdef:svctype
      name="service type name"
      defaultPort="default TCP port">
    <mdef:param
        name="parameter name"
        label="parameter display label"
        default="default value or default variable"/>
    ... additional parameters, if any ...
    <mdef:script>
      ... see text for script syntax ...
    </mdef:script>
  </mdef:svctype>
  ... additional service types ...
</mdef:definition>
```

Figure 2: Mars probe definition file structure

The configuration file format fairly closely follows the data model as visible from the host/service tree - a hostlist contains hosts, which each contain services. The format of the file is given in figure 1.

Mars itself is a good generator of example files; just create a simple configuration, then open up the resulting XML file in your favorite text editor.

2 The Probe Definition File

Simple text-based TCP protocol probes are defined in the `mars-def.xml` file. On startup, Mars searches for this file in the Mars home directory, which is the current working directory by default, though this can be changed using the `--home` command-line option. If a `mars-def.xml` file cannot be found in the Mars home directory, Mars falls back to a default version of the file stored in the `mars.jar` archive itself.

This file's format is a little more complex than the config file's. Broadly speaking, the definition file lists a collection of service types, each of which has a list of parameters and a script that defines the protocol used by the service. The structure is shown in figure 2.

Each parameter has a name, by which it will be referenced in a `<mdef:param>` element within the script; a label, which will be shown to the user in the service editor; and a default, which is used as the parameter's value if the user leaves it blank. The default may be any constant string, or a default variable. The only currently supported default variable is `%%(remote-hostname)`, which resolves to the DNS name or IP address of the host on which the service is running.

Scripts are series of “send”, “expect”, and “fail” instructions, contained in `<mdef:send>`, `<mdef:expect>`, and `<mdef:fail>` elements, respectively. If a dialogue with a server matches all of a script's expect instructions in order without matching any fail instructions, the probe succeeds. The probe may time out waiting for an expect. If a fail instruction matches, the probe fails with a status of “bad reply”. Fail instructions must be immediately followed by an expect instruction, as fail instructions “expire” when the expect instruction immediately following them successfully matches.

Each send instruction contains mixed content. The following elements are valid within send instructions:

`<mdef:param name>` expands to the value of the named parameter as set for each service instance.

`<mdef:remote-hostname>` expands to the address of the host being probed.

`<mdef:local-hostname>` expands to the hostname of the local interface from which the probe is being run.

`<mdef:version>` expands to the Mars version number. This is useful for signaling Mars’ version to protocols that care about that sort of thing; it is used to accurately report Mars’ version on HTTP probes via the `User-agent` request header.

`<mdef:space>` encodes a single space. This is useful for adding leading or trailing spaces to mixed string content, as whitespace can be otherwise normalized away.

`<mdef:crlf>` likewise encodes a newline sequence.

Each expect or fail instruction contains either a literal text string or an `<mdef:regex>` element. These `regex` elements match the Perl-compatible regular expression defined in the `pattern` attribute, or filled in from the service parameter named in the `name` attribute

As with the config file, it’s probably best to look over the `mars-def.xml` file itself to get the big picture.

3 Building Mars

The Mars build system, contained in the files `build.xml`, `buildext.xml`, and `buildmac.xml`, is based on Apache Ant 1.5. The following targets are the most commonly used in building Mars.

all (the default target) compiles any (changed) source files in the `org` and `gnu` directories, stages any unstaged `.jar` files in `lib` into `lib/staging`, and creates or updates the `.jar` files in the distribution directory (`mars-2.2.4`). It also builds any optional extensions in the `extras` directory named by the `extras` property (see below), and places them in the `extras` directory in `mars-2.2.4`.

dist runs `veryclean` (see below), then `all`, then makes a `mars-2.2.4.tar.gz` binary distribution. On Mac OS X, this target also builds a Mac OS X installer package image in `mars-2.2.4/macpkg`, suitable for use with the Package Maker specification file `extras/macosx/Mars.pmsp`.

clean removes the distribution and object directories.

veryclean additionally removes the `lib/staging` directory, where jarfiles in the `lib` directory are staged for inclusion into `mars.jar` and `mars-j14.jar`. Use this to force the `lib/staging` directory to be rebuilt when you change `.jar` files in the `lib` directory. This almost never happens.

The `extras` property used by the build system (set on the Ant command line with `-Dextras=a,b,c,...`) is a comma-separated list of extras to build during the `all` target. The following values are supported:

all Build all extras.

https Build the HTTPS probe extension. This extension requires JSSE to be installed on Java versions earlier than 1.4.

jdbc Build the JDBC probe extension.

xmpp Build the XMPP (Jabber) notification plugin.

cdebug Build the Client Debugger plugin.

mailnotify Build the SMTP notification plugin.

swingnotify Build the UI notification plugin.

csvlog Build the CSV logging plugin.

The Mac OS X Integration plugin is always built by default on Mac OS X. Note that this implies that building Mars on Mac OS X requires Java 1.4.

4 Building Probe Extensions

Some services protocols' may not be expressible in the probe definition file, especially non-text-based or non-TCP services. These probes may be implemented as probe extensions to be dynamically loaded at started, much like plugins are. To implement a probe extension:

1. Implement a subclass of `Probe`. You should only need to implement a single method, `doProbe()`, which will take information about the service to probe via the protected `service` instance variable, probe the specified service, and return a `Status` instance describing the status of the service. See `SendExpectProbe` and `SendExpectClient`, which implement the probe infrastructure used by the probe definition file mechanism, for an example of a probe.

2. You should instrument your probe for the Client Debugger. This will allow you to verify your probe's functionality and allow users to view details your probe's operation in real time. Instrument your probe as follows:

   (a) At the beginning of `doProbe()`, call static method `Debug.getDebugger()` to get a new `ClientDebugger` instance. This method takes a single argument, a string which will be displayed to the user to identify this debug session. Try to use a string which will allow the user to distinguish multiple instances of your probe running concurrently.

   (b) Each time you send data, pass the data (or some textual abstract representation of it) to your `ClientDebugger`'s `send()` method.

   (c) Each time you receive data, pass the data (or some textual abstract representation of it) to your `ClientDebugger`'s `receive()` method.

   (d) To notify of status not related to sending or receiving (for example, some decision your probe makes), use `ClientDebugger`'s `message()` method.

   (e) When your probe completes, be sure to `close()` the `ClientDebugger`. This must be done or the debug session information will never be removed from the active sessions list. This call usually goes in a `finally` block within your probe.

3. Implement a subclass of `ProbeFactory` to create instances of your new probe, and to describe the parameters your probe requires to Mars. You will need to implement three methods:

   A constructor which calls `ProbeFactory()` with the name of the service type monitored by your probe. This service type name will be exposed to the user via the `Type` menu in the service editor.

   `createProbe(Service)` creates a new probe instance to monitor the specified service.

   `getDefaultPort()` returns the default port used by the service.

   If your probe takes any service parameters, you will also need to implement the following three methods:

   `getServiceParamNames()` returns an array of Strings containing the service parameter names required by this service type.

   `getServiceParamLabels()` returns an array of Strings containing the service parameter labels to be shown in the user interface, corresponding by position to the service parameter names.

   `getServiceParamDefault(String)` returns a String containing the default value for the named parameter.

4. Place your probe and probe factory classes, along with any support classes they require, into a `.jar` file. This file's name must begin with the string `probe_`, and its manifest must contain a key `MarsProbeFactoryClass` whose value names your `ProbeFactory` subclass. You can place multiple probes into a single `.jar`; simply name each `ProbeFactory` subclass in the `MarsProbeFactoryClass` separated by commas.

5. Place this file in the Mars distribution directory, and start Mars. Mars should detect your probe extension(s) on startup, and they should be available on the Type menu in the service editor.

5 Building Plugins

As probe extensions provide new inputs to Mars' monitoring engine, plugins provide a way to add new outputs. Each plugin can be configured to receive information about the results of each probe run, or about changes to each service's status.

Plugins receive status information via events recieved through three interfaces, `ProbeListener`, `StatusChangeListener`, and `NotificationListener`. Plugin housekeeping tasks are managed through a fourth interface, `Plugin`. Your plugin class must implement `Plugin`, and at least one of the three listener interfaces. Your class will not need to register as a listener over these interfaces; the plugin framework handles this for you as long as you have implement at least one of the interfaces.

The `ProbeListener` interface defines a single method, `probeRun(ProbeEvent)`, which is called by the controller every time a probe is run, even when the service status has changed. The `ProbeEvent` instance passed to this method contains information about the service that was probed via the `getService()` method, and information about that service's current status via the `getNewStatus()` method.

The `StatusChangeListener` interface also defines a single method, `statusChanged(StatusChangeEvent)`, which is called by the controller every time a service's status changes. `StatusChangeEvent` is a subclass of `ProbeEvent` supporting one additional method, `getOldStatus()`, which returns the previous status of the service.

The `NotificationListener` interface defines the `notifyStatusChanged(StatusChangeEvent)` method, which is called as specified by each service's notification retry count (`notac`) field after a service's status changes. The `StatusChangeEvent` instance is the same one that would have been sent via `statusChanged()` when the service's status initially changed.

Plugin configuration is done via editor panels displayed through the Plugins menu, and configuration data is stored along with the host/service tree in the Mars configuration file. The `Plugin` interface defines methods your plugin must implement to support these configuration functions:

A constructor taking no arguments, which should create a new instance of your plugin with a default configuration. The plugins shipped with Mars configure themselves with empty values assigned to each configuration parameter and a global "enabled" flag set to off, to ensure that the user must configure the plugin before its first run.

`getConfig()` returns a JDOM `Element` containing all of the configuration information in an XML element suitable for inclusion in the configuration file.

`setConfig(Element)` takes in a JDOM `Element`, previously returned by `getConfig()` and written to a configuration file, and configures the plugin based on its attributes and contents.

`getElementName()` returns the configuration element name within the Mars configuration XML namespace used by this plugin to identify its configuration.

`getDisplayName()` returns a string to be displayed in the plugins list in the Mars main window's Config tab.

`getEditor()` returns a `JComponent` implementation of the `Editor` interface used to configure the plugin via the user interface. Editors are often inner classes extending `JPanel`. They must provide the following methods:

> **A constructor** that builds the editor panel's user interface and fills in fields with values from the plugin's present configuration.
>
> `commit()` configures the plugin based on the information in the editor's editable fields. It may throw any `Exception` to signify illegal input.
>
> `getEditorTitle()` return a string to use as the editor dialog's title.

As of Mars 2.2.6, plugins may now request a tab in the Mars main window. This facility is designed for alternate visualization plugins. To use this facility, your plugin must implement the `Displayable` interface. `Displayable` defines a single method, `setPluginDisplay()`, which Mars will call when the user interface starts up (after configuration if a configuration file is specified on the command line; see below). This method takes a single argument, an instance of `PluginDisplay` which your plugin will use to control its tab in the user interface. This class has the following methods:

`setComponent()` takes a `JComponent` instance to display in the plugin's tab.

`setTitle()` set's the plugin's tab's title.

`show()` makes your plugin's tab visible (call this when your plugin is enabled).

`hide()` makes your plugin's tab invisible (call this when your plugin is disabled).

You must set the plugin's tab's component and title before calling `snow()`; otherwise, a runtime exception will result. See the source code of the Client Debugger in `extras/cdebug` for an example.

The `Displayable` interface is also used as a hint to Mars that it should not attempt to load your plugin in "headless" (`--nogui`) mode; if your plugin uses the user interface but does not need its own tab (as the `SwingNotifyPlugin` does), you can implement `Displayable` but make your `setPluginDisplay()` method a no-op.

Additionally, if your plugin class has an "enabled" state (as all the included notification plugins do), you may implement the `Enableable` interface to ensure that your plugin's enable state is properly displayed in the Plugins menu. `Enableable` defines a single method, `isEnabled()`, which returns `true` if the plugin is presently enabled, `false` otherwise.

When a new Mars configuration file is loaded, each plugin is configured according to the following protocol:

1. The plugin registry calls `getElementName()` on the plugin to determine the name of the XML element it uses for configuration.

2. If the configuration file contains that element, it passes that element to the plugin via `setConfig()`.

3. If no element of the given name exists in the configuration file, the plugin is *not* reconfigured. This allows older config files to be loaded with new plugins without resetting the plugin configuration.

Note that the only time Mars calls your plugin's `setConfig()` is when a configuration file is loaded that contains an element in the Mars namespace with the element name returned by your `getElementName()` method. In particular, this means that your plugin *will not be configured* when Mars starts up without a configuration file. Your plugin's no-argument constructor must leave the plugin in a state in which it will not crash or cause other adverse effects for Mars.

Have a look at the source for `SwingNotifyPlugin` for an example of a simple plugin.

To package your plugin for dynamic loading, place your plugin class along with any support classes it requires into a `.jar` file. The file's name must begin with the string `plugin_` and its manifest must contain a key `MarsPluginClass` which names your implementation of `Plugin`. You can place multiple plugins into a single `.jar`; simply name each `Plugin` subclass in the `MarsPluginClass` separated by commas.

Place this file in the Mars distribution directory, and start Mars. Mars should detect your plugin(s) on startup, and you should be able to configure them via the Plugins menu.

6 Architecture Overview

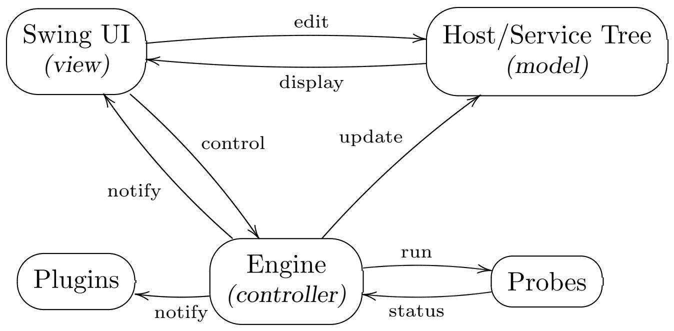

Figure 3: Mars architecture overview

Mars is built along the lines of a straight MVC architecture. Information about hosts and services lives in a Model (the `Host`, `Service`, and `Status` classes), which is edited and displayed by a View (the Swing user interface in `org.altara.mars.swingui` and the plugin extension framework), and updated by the asynchronously-running Controller (the classes in `org.altara.mars.engine`) which invokes probes (defined in the probe definition file and the probe extension framework) to determine status information. This general arrangement is shown in figure 3.

7 Class reference

The tables that follow list all the classes that make up Mars. This list should suffice to get you started hacking away at Mars, by answering the the frequent and essential questions "where does feature *x* live?" and "what does *y* do?".

7.1 Data Model

Core data model classes in `org.altara.mars`:

`Main` the entry point to Mars. It parses the command line, initializes and manages access to plugins, probes, and the model, and contains system-wide constants.

`MarsModel` is the root of the host/service tree; it contains the host list, and distributes host/service tree change notifications to the user interface.

`Host` represents a host Mars is monitoring.

`Service` represents a service Mars is monitoring.

`Status` represents the status of a service at a given point in time.

`ProbeFactory` is an interface implemented by concrete factories that can create probes for a given service type. It also contains a registry of such factories, and handles dynamic loading of probe extensions at startup.

`MarsModelListener` is an interface implemented by objects interested in edits to the MarsModel tree.

`InvalidDocumentException` is thrown during configuration file loading if the file is not a proper Mars configuration file.

`InvalidServiceTypeException` is thown during configuration file loading if a service's type is not supported by any probe.

7.2 Monitoring Engine

Monitoring engine classes in `org.altara.mars.engine`:

`Controller` coordinates the activities of the engine. It distributes event notifications, and manages the queue of pending probes and probe worker threads.

`ProbeWorker` runs Probes in separate threads.

`Probe` is the abstract superclass of all probes; classes which know how to speak a given protocol or set of protocols, and determine the status of a remote service.

`SendExpectProbe` is a thin wrapper around `SendExpectClient` used to probe text-based TCP/IP services. This is sufficient for all probes defined via the probe definition XML file.

`SendExpectClient` is a simple text-based TCP/IP client that knows how to send strings and expect strings in return. It is instrumented to return a Status so it can be used as a probe; it's also used by the mail notification plugin for a simple SMTP client.

`XmlProbeFactory` parses the probe definition file and sets up a `SendExpectProbe` for each service it finds.

`ProbeListener` is implemented by any object interested in probe run notifications.

`ProbeEvent` is sent to each `ProbeListener` to notify of a `Probe`'s being run.

`StatusChangeListener` is implemented by any object interested in service status change notifications.

`StatusChangeEvent` is sent to each `StatusChangeListener` to notify of a service status change.

`Notifier` manages the firing of `NotificationEvents`.

`Debug` contains static proxy methods for various debuggers

`ClientDebugger` defines methods for network client debugging (these are implemented by the Client Debugger plugin)

7.3 User Interface

User interface classes in `org.altara.mars.swingui`:

`MarsView` implements the Mars main window.

`MarsAbstractRenderer` provides common support to each Mars renderer class.

`ServiceTreeRenderer` renders the nodes in the host/service tree.

`ServiceTreeChangeAdapter` ensures that events received by the host/service tree run in the Swing dispatch thread.

`ServiceTreeContextMenuSupport` implements the service tree context menu.

`ServiceTreeKeyActionSupport` implements the service tree keyboard editing functions.

`FaultListModel` implements a list model for the fault list. This is necessary to support fault hiding.

`FaultListRenderer` renders the items in the fault list.

`FaultListContextMenuSupport` implements the fault list context menu.

`DetailListModel` implements the service details panel.

`ChangeListModel` implements the change list in the History tab.

`ChangeListRenderer` renders the items in the change list.

`ChangeListPanel` implements the History panel.

`PluginMenu` implements the Plugins menu.

`Editor` is implemented by Mars property editors: the host, service, and the plugin configuration editors.

`EditorDialog` displays an Editor in a dialog and handles error reporting.

`HostEditorPanel` implements the host editor.

`ServiceEditorPanel` implements the service editor.

`ServiceTypeComboBox` implements the Type menu in the service editor.

`ServiceParamEditor` implements the parameters pane in the service editor.

`ProbeThreadAdapter` ensures that events sent to a `ProbeListener` are handled within the Swing dispatch thread.

`StatusChangeThreadAdapter` ensures that events sent to a `StatusChangeListener` are handled within the Swing dispatch thread.

7.4 Plugins

Plugins and support classes in `org.altara.mars.plugin`:

`Plugin` is implemented by all plugins to provide plugin housekeeping services.

`Enableable` is implemented by plugins with an "enabled" state.

`Displayable` is implemented by plugins that use the Swing user interface or display their own tab in the Mars main window.

`PluginDisplay` is used by each `Displayable` plugin to control its tab in the Mars main window.

`PluginRegistry` manages loading, instantiating, and configuring plugins.

7.5 Utility classes

Utility classes in `org.altara.util`:

`Queue` implements a queue atop `LinkedList`.

`Worker` provides a managable interface to a thread for running a repetitive task; it makes up for the unsafety of the `Thread` control methods.

`StatusView` provides an interface for views with status bars or logs. It is used to implement the startup log in the Mars splash screen, and the Mars main window status bar.

`ExtensionLoader` dynamically loads classes from `.jar` files in a given directory.

`LoadExceptionHandler` is implemented by classes that can handle exceptions during the extension loading process.

`IconService` loads icons from a .jar file.

`ContextMenuSupport` provides common support for context menus on Swing components.

`ListContextMenuSupport` implements context menus on Swing `JList`s.

`TreeContextMenuSupport` implements context menus on Swing `JTree`s.

`KeyActionSupport` implements keyboard shortcuts on arbitrary Swing or AWT components.

`TreeKeyActionSupport` implements keyboard shortcuts on Swing `JTree`s.