Developing Agents for Applications Running on Sun™ Cluster Software

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
1 (800) 786.7638
1.512.434.1511

Contents

CHAPTER 1

Introduction

This paper explains how to use the SunPlex™ Agent Builder to develop agents for applications running on Sun™ Cluster software. It is oriented primarily toward potential agent developers who may or may not be familiar with the Sun Cluster product.

Sun Cluster 3.0 is Sun's next-generation clustering technology. Sun Cluster 3.0 focuses on delivering integrated availability, scalability, manageability, and ease of use with the core delivery platform — the Solaris™ Operating Environment. Built around Sun Cluster 3.0, the Solaris Operating Environment, and Sun™ server, storage, and network connectivity products and services, the SunPlex system is designed to manage application services for tightly coupled environments, optimizing the availability and scalability of these services.

The terms "agent" and "data service" are used interchangeably in this paper to refer to the callback methods that start, stop, and monitor the application. These methods, written in either 'C' or shell commands, enable applications to become highly available or scalable in the Sun Cluster environment. A *highly available,* or *failover,* application is online on, at most, one cluster node at a time. HA Oracle is an example of a failover application. A *scalable* application can be online on multiple nodes simultaneously. An example of an application that can be made scalable is the Apache Web server.

The paper first describes the prerequisites for making an off-the-shelf application highly available or scalable using the Sun Cluster APIs. It then presents an overview of the Sun Cluster resource management facility and its object model, followed by a short discussion of the process monitor facility (PMF). Next it discusses the Sun Cluster Resource Management API (RMAPI), the Data Service Development Library (DSDL) API, and the Agent Builder. Finally, it explains how to use the Agent Builder to develop agents for failover and scalable applications, including an example.

CHAPTER **2**

Prerequisites for Making Applications Failover or Scalable in the SunPlex™ Environment

Before implementing an agent, the agent developer must determine that the target application satisfies the requirements for making an application highly available or scalable in the SunPlex environment.

If the application does not meet the requirements, then modifications to the application source may be necessary. The requirements for making an application scalable are a superset of those for a failover application. In order to be highly available, the application must:

- Be either network-aware (client/server model), or nonnetwork-aware (client-less). Sun Cluster software cannot, however, provide enhanced availability in time-sharing environments where applications are run on a server that is accessed through `telnet` or `rlogin`.
- Be crash tolerant — capable of recovering application files (if necessary) when it is restarted after an unexpected failure. Furthermore, the recovery time after a crash must be bounded. Crash tolerance is a prerequisite for making an application highly available, because crash recovery (the ability to recover application files and restart) is a data integrity issue.
- Not depend upon the physical hostname of the node on which it is running.
- Operate correctly in environments where multiple IP addresses are configured. This includes 1) multihomed hosts, in which the node is on more than one public network, and 2) nodes where multiple, logical interfaces are configured on one hardware interface.
- Store data in the global cluster file system. If the application uses a hardwired path name for the location of the data, the agent developer may change that path to a symbolic link, that points to a location in the cluster file system, without changing application source code.

- Place application binaries and libraries locally on each node or on the global cluster file system. The advantage of residing on the global cluster file system is that a single installation suffices for all cluster nodes.
- Allow the client some capacity to retry a query automatically if the first attempt is cut off or timed out. If the application and protocol already handle the case of a single server crashing and rebooting, then they also will handle the case of a containing resource group being failed over or switched over.
- Not have UNIX® domain sockets or named pipes in the cluster file system, because the cluster file system currently does not support them. This restriction will be removed in a future release.

In addition to the prerequisites listed above, scalable applications must also:

- Have the ability to run multiple instances, all operating on the same application data in the cluster file system.
- Provide data consistency for simultaneous access from multiple nodes.
- Implement sufficient locking via a globally visible mechanism, for example, the cluster file system.

Additionally, application characteristics determine the choice of load balancing policy. There are two classes of scalable services: *pure* and *sticky*. A pure service is one where any instance of the application can respond to client requests. A sticky service is one where a client sends its requests to the same instance; those requests are *not* redirected to other instances.

CHAPTER **3**

Resource Management Object Model

The Resource Group Manager *(*RGM) facility supports the application programming interfaces for Sun Cluster software. It provides a mechanism for making cluster resources highly available and scalable by automatically starting and stopping these resources on selected cluster nodes, according to preconfigured policies, in the event of hardware or software failures or reboots. A provision is also made to automatically start and stop resource-specific monitors, which can detect resource failures and relocate failing resources onto another node, and may monitor other aspects of resource performance. The system administrator can perform runtime load balancing or facilitate maintenance tasks by moving groups of resources from one node to another. The RGM supports both failover resources, which can be online on up to one node at a time, and scalable resources, which can be online on multiple nodes simultaneously.

Resource Types and Resources

An RGM resource is generic; it represents any entity that can be brought online on a cluster node, such as a software application, a network address, or a volume manager disk group. Each resource must be configured in a resource group.

The *resource type* declares common properties and callback methods applying to all resources of the given type. *Callback methods* are programs associated with the resource, such as START, STOP, MONITOR_START, MONITOR_STOP, and others. START and STOP bring the resource online and offline, while MONITOR_START and MONITOR_STOP control a *resource monitor*, which can monitor the health and performance of the resource. Via an API function, a resource monitor can request that a failing resource be relocated from one of its current masters onto another node.

Some resource type implementations for Sun Cluster software are supplied by Sun, while others are provided by customers or third parties, using APIs and tools provided by Sun. Resource types are installed on the cluster by using standard tools of the Solaris Operating Environment such as `pkgadd`. They are made known to the RGM by the `scrgadm` command.

The system administrator may *instantiate* a resource type in a given *resource group* by configuring the resource. Each resource has a list of properties, some of which are configured by the resource type implementor, others by the system administrator at runtime. Each time a resource group is brought online on a node, the RGM will start all of the resources contained in it.

The *resource type registration* (RTR) file declares resource properties and sets limits and defaults for the values of these properties. The file is used with administrative commands to configure the resource type into the cluster.

Resource Groups

A resource group contains a set of resources, all of which are brought online or offline together on a given node or set of nodes.

When a resource group is brought online on a node, all of the resources in that group are activated through their callback methods. Similarly when the resource group is taken offline, all of the resources are deactivated through their callback methods. (Callback methods are described in the next section.)

A resource group can be configured to be online on one node at a time (failover) or on multiple nodes (scalable). Bringing the resource group online on a node means starting all of the resources in the resource group by invoking their START methods on the given node; bringing it offline means stopping the resources by invoking their STOP methods.

The nodes on which a resource group is currently online are called the *primaries* of the resource group. A resource group is said to be mastered by each of its primaries. Each resource group has an associated nodelist property which lists all of its *potential primaries* (or *potential masters*) — nodes that are capable of becoming primaries of the resource group.

The RGM responds to cluster membership changes (for example, node crashes, halts, or reboots) or to the failure of individual resources, by automatically relocating resource groups to different nodes. The RGM also enables the system administrator to manually switch resource groups from node to node for load balancing or to facilitate system maintenance.

RGM Reconfigurations and Basic Method Semantics

When the cluster state is stable, meaning that no nodes are in the process of joining or leaving the cluster membership, and no resource groups are in the process of being brought online or offline, the RGM has no work to do and is dormant. Several types of events can cause the RGM to become active:

- A node joins or leaves the cluster membership by booting, crashing, or halting.
- A resource monitor requests a change in resource group mastery by invoking the `scha_control` API function.
- The system administrator requests a change in the mastery of a resource group, a global file system or disk set, a global network address, or another global resource by invoking the `scswitch` utility.
- The system administrator triggers a change in the state of an online resource group or resource by editing its properties.

When any of these events occurs, the RGM becomes active on all cluster nodes. It examines the cluster state and pending `scha_control` or `scswitch` actions, to determine whether it needs to modify the state of resource groups or resources, for example, to bring resource groups online or offline on one or more nodes. The RGM then carries out these configuration changes, coordinating its actions across all the member nodes of the cluster. The period of time during which the RGM is active and resource groups are being brought online or offline is referred to as a *reconfiguration*.

CHAPTER **4**

The Process Monitor Facility (PMF)

The PMF provides a means of monitoring and restarting processes and their descendents. The total number of failures allowed can be specified, and may be limited to a specific time period. After the maximum number of failures has occurred within the specified period, a message is logged to the console and the process is no longer restarted.

If an action program has been specified, it will be called when the number of allowed failures has been reached. If the action program exits with nonzero status, the process nametag is removed from the PMF. Otherwise, the process is restarted with the originally specified parameters.

Processes that are started under control of the PMF are run as the uid of the user that initiated the request. Only the original user, or root, can manipulate the nametag associated with those processes. Status information, however, is available to any caller, local or remote.

All of the spawned processes, and their descendents are monitored. Only when the last process, or subprocess exits does the PMF attempt to restart the process.

CHAPTER **5**

Application Programming Interfaces Supported by the RGM

The RGM supports several layers of APIs, as shown in Figure 1. At the bottom is a set of low-level routines that enable an agent developer to access information about the resources, resource types, and resource groups in the system; request a local restart or a failover; and set the resource status. There is also a function for translating an error code returned by one of the Resource Management API (RMAPI) functions to the appropriate error message string, which can be accessed using the `libscha` library.

The Data Service Development Library (DSDL), `libdsdev`, is layered on top of the `libscha` library. The DSDL APIs extend and build on top of the RMAPI, providing a higher level integrated framework. Agents supplied by Sun use the DSDL APIs for most of their implementation.

The Agent Builder introduced in the next section is a tool that further simplifies agent implementation by automating as much of the development process as possible.

Note that the APIs described in this paper should be used by the resource callback methods and monitors for applications that do not need to be cluster-aware. These applications are expected to be used off-the-shelf, with no modifications to their source code. There exists, however, a class of applications that need to be cluster-aware. This class includes applications such as the iPlanet™ Application Server, BEA Weblogic Server, and IBM WebSphere. In order to integrate them with Sun Cluster software, the source code of the applications must be able to register for notification of certain types of cluster events. Cluster-aware APIs that will be callable directly by the applications are currently under development.

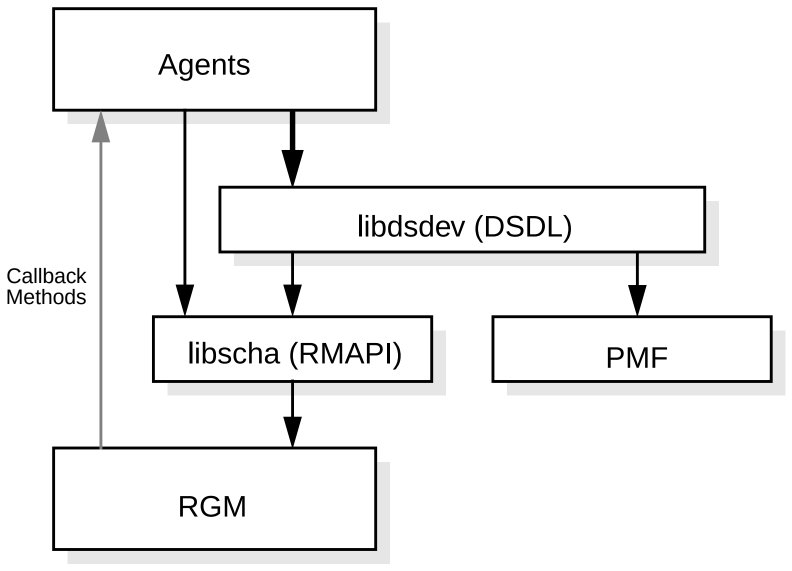


FIGURE 1 Sun Cluster Resource Management Architecture

Data Service Development Library (DSDL) APIs

The DSDL API, layered on top of the RMAPI, enables development of Sun Cluster agents using predetermined solutions to various Sun Cluster software integration issues. It allows the agent author to devote the majority of development time on high availability and scalability issues intrinsic to the application, not on integrating the application start-up, shutdown, and monitor procedures with Sun Cluster software. To successfully integrate an agent with Sun Cluster, these additional tasks must be performed:

- Manage the application configuration properties, including validation, retrieval, and error checking of various operations involved in these tasks.
- Integrate the service with the network address resources defined in the system. This includes enumerating the resources, retrieving and using the hostnames which are managed by the resources.
- Design and implement procedures for monitoring the service. These include process failure detection, handshaking with the application to make sure it is providing service to its clients, and taking corrective actions when it is not.

- Devise a scheme to handle service failures, and utilizing the `scha_control` API function to achieve failover of application resources. This requires a mechanism that keeps track of service failures and decides whether to attempt to correct the service problems locally or to fail over the resource group containing the service to some other node.

The DSDL provides solutions for the above tasks while enabling the agent author to retain a degree of control over the application's behavior.

Overview of DSDL Functions

The functions in the DSDL can be divided into the following sets of C functions. Shell command versions of the API will be supported in the near future.

- **Property functions.** These routines are convenience APIs for accessing various properties of the relevant resource, resource group, and resource type. The DSDL provides routines to parse the command line arguments. The library then caches the various properties of the relevant resource, resource group, and resource type.
- **Network probe functions**. These routines enable a fault monitor to connect through TCP to the underlying application to perform basic monitoring. The routines are convenience APIs that allow the caller to specify a timeout and do some basic error checking and recovery.
- **PMF functions**. The library creates and uses implicit nametag values, as well as implicit values for the restart interval, retry count, and action script. Most importantly, the library ties the process death history (as discovered by PMF), into the application failure history (as detected by the fault monitor) to make the decision to failover or restart.
- **Fault monitor functions**. These routines provide a predetermined model of fault monitoring by storing the failure history and evaluating it in conjunction with the retry count and retry interval.
- **Utility functions**. These routines consist of syslog and debugging-related routines.

CHAPTER 6

The SunPlex Agent Builder

The preceding section hints that it is easier to develop an agent using the DSDL API than the RMAPI. The SunPlex Agent Builder is a tool that further simplifies development of agents by automating their creation and packaging. This includes creating the Resource Type Registration (RTR) file and implementing the callback methods defined in the RTR file.

The Agent Builder asks the user a few simple questions and generates a set of source files. The user has the option of generating source code in the C programming language or the ksh shell language. The agent developer can modify the generated files, if necessary.

CHAPTER 7

Using the SunPlex Agent Builder

The SunPlex Agent Builder is a tool that automates the creation of agents to be run under the Sun Cluster Resource Group Manager. It is available in the SUNWscrtw package.

The Agent Builder generates and customizes the RTR file.

In addition, it generates customized utility scripts for starting, stopping, and removing an instance or resource of the target resource type. (These utility scripts should not be confused with the START and STOP resource callback methods described earlier). The start script registers the resource type and creates the necessary resource groups and resources. It also creates the network address resources (LogicalHostname or SharedAddress) to be used by the application to communicate with the clients on the network. The stop script stops and disables the resource. The remove script undoes everything the start script did — it stops and removes the resources, resource groups, resource type, and agent from the system.

The Agent Builder packages everything into Solaris Operating Environment package that can be installed on all nodes of the cluster.

Supported Applications

The Agent Builder supports network-aware as well as nonnetwork-aware (or client-less) applications.

The Agent Builder supports applications having more than one independent process tree that needs to be individually monitored and restarted by PMF. For more information on creating such agents, see the section, “Creating Resource Types with Multiple Independent Process Trees.”

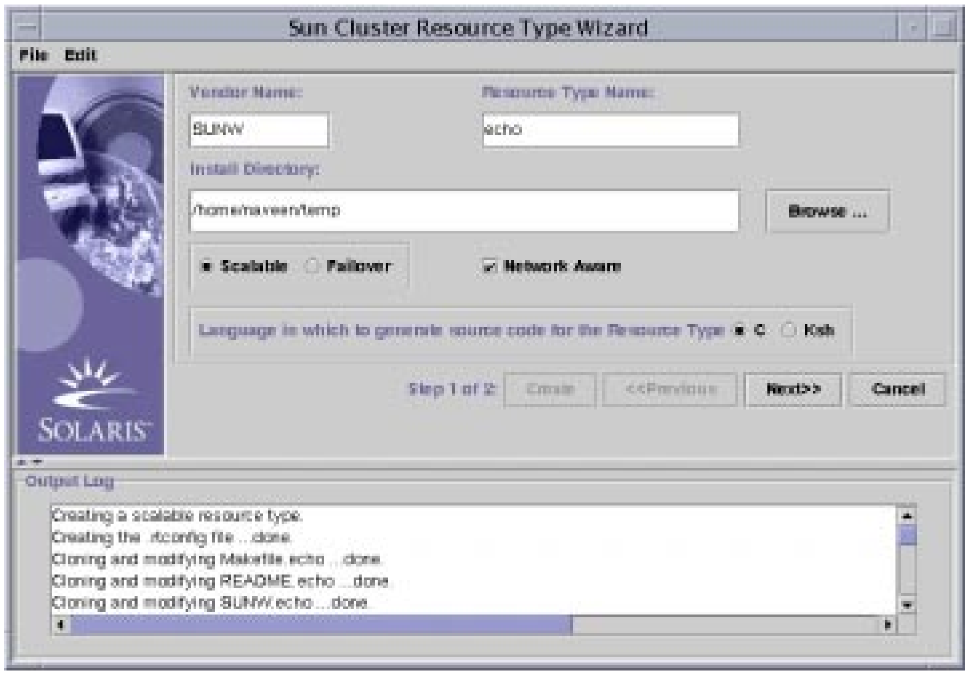
An additional option allows source code to be generated in C or ksh. The latter is useful for those without access to a C compiler, or who are more comfortable with ksh. For ease of extensibility and richer underlying functionality, Sun encourages the use of C. Agents with ksh source code do not support nonnetwork-aware applications or applications with multiple process trees.

Using the GUI Version of the Agent Builder

Creation of agents using the Agent Builder is a two-step process. In the first step, the user is asked to input some basic information about the resource type to be generated:

- **Resource type name.**
- **Vendor id**. The stock symbol, or some other identifier that uniquely identifies the vendor.
- **Installation (or root) directory**. This is where the Agent Builder will create a subdirectory formed by the concatenation of the vendor_id and the resource_type_name. For example, if the vendor_id is “SUNW,” and the resource_type_name is “ftp,” the Agent Builder creates a subdirectory SUNWftp under the installation directory specified. SUNWftp contains everything that is generated for the target resource type. The installation directory is initialized to the directory where the Agent Builder was started.
- **Failover or scalable**. Is the target resource type failover or scalable?
- **Network aware**. Does the base application use the network to communicate with its clients?
- **C or ksh**. Which language is used for generated source code.

Figure 2 shows a snapshot of the “Create” screen. (These screen shots may change when the GUI is finalized.)



Vendor Name: SUNW

Resource Type Name: echo

Install Directory: /home/naveen/temp

[✓ Scalable] Failover

[✓ Network Aware]

Language in which to generate source code for the Resource Type [✓ C] Ksh

Output Log

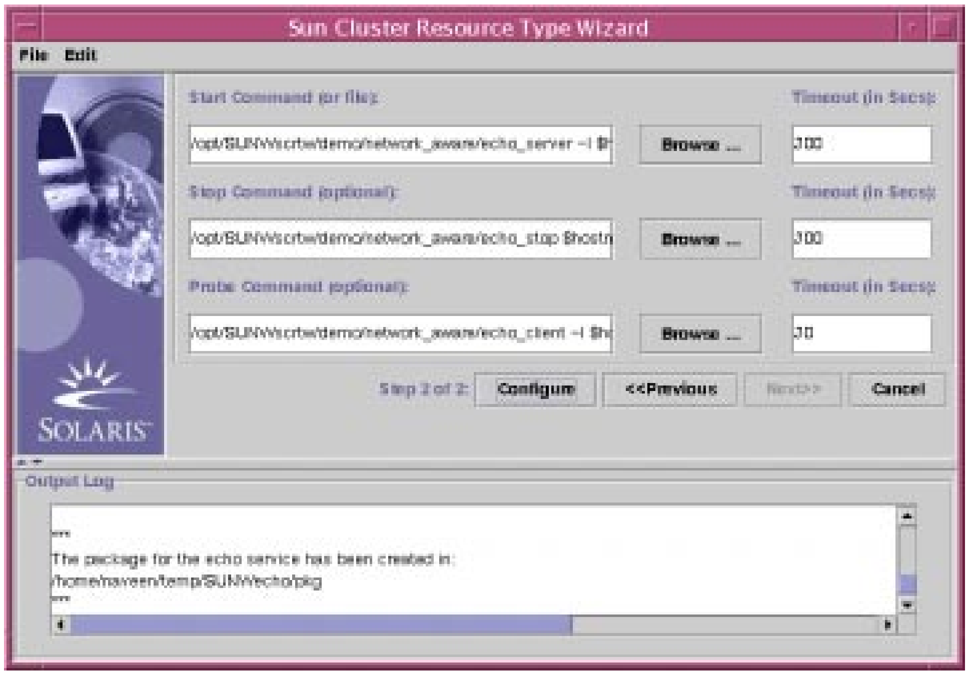
```
Creating a scalable resource type.
Creating the .rtconfig file ...done.
Cloning and modifying Makefile.echo ...done.
Cloning and modifying README.echo ...done.
Cloning and modifying SUNW.echo ...done.
```

FIGURE 2 Create screen in step one of the Agent Builder.

In the second step, the user is asked to input various configuration parameters for the target resource type:

- **Start command**. The only mandatory input on the second screen, this is the full command line that can be passed to any UNIX shell to start the application. The complete command line must include everything needed to start the application, for example, hostnames, port numbers, path to configuration files, and so on. The completed command line must not be quoted (“”).
- **Stop command.** Optional input that specifies the command to stop the base application. If this input is omitted, the generated code uses signals to stop the base application.
- **Probe command**. Optional input that specifies a command which can be periodically invoked to perform a health check on the application and return appropriate exit status (zero for success and nonzero for failure). This command would typically be a simple client of the base application. If it is omitted, the generated code simply connects and disconnects to the port specified and, if that succeeds, declares the application healthy. The probe command can only be used with network-aware applications and is disabled for nonnetwork-aware applications.

- **Timeouts**. Include timeouts for the start, stop, and probe commands and are initialized to 300, 300, and 30 seconds, respectively.



Start Command (or file): /opt/SUNWscrtw/demo/network_aware/echo_server -l $h
Timeout (in Secs): 300

Stop Command (optional): /opt/SUNWscrtw/demo/network_aware/echo_stop $hostn
Timeout (in Secs): 300

Probe Command (optional): /opt/SUNWscrtw/demo/network_aware/echo_client -l $h
Timeout (in Secs): 30

Step 2 of 2:

Output Log

```
***
The package for the echo service has been created in:
/home/naveen/temp/SUNWecho/pkg
***
```

FIGURE 3 Configure screen in step two of the Agent Builder

Using the Command Line Version of the Agent Builder

The command line version of the Agent Builder utilizes the same two-step process with different commands to be run at each step, `creatert` and `configurert`. (These command names are subject to change.) It generates the same code as the GUI version of the tool.

Developing an Agent for a Scalable Application Using the SunPlex Agent Builder: An Example

This section shows how easy it is to develop a scalable agent for the Apache Web server that ships with the Solaris Operating Environment. The procedure for developing a failover resource type simply omits the ‘`-s`’ command line option in step two.

The assumptions are:

- Apache meets all the requirements described in the section “Prerequisites for Making Applications Failover or Scalable on Sun Cluster Software”
- The application has been installed locally on all nodes of the cluster and both the starting and stopping of Apache in the init scripts have been disabled
- The document directory has been configured to be in the cluster file system
- The resource monitor will probe Apache using the public domain `wget` utility
- The hostname used by clients to talk to the Apache server is *webserver*

To complete the process, perform the following simple procedure:

1. On the development machine, create a local directory for the agent:

```
mkdir -p /<development path>
```

2. On the development machine, issue the Agent Builder command to create a scalable resource type:

```
/opt/SUNWscrtw/bin/creatert -V DEMO -T http -s \
-d /<development path>
```

3. On the development machine, issue the Agent Builder command to configure the resource type:

```
/opt/SUNWscrtw/bin/configurert \
-s "/usr/apache/bin/apachectl start" \
-m '/usr/bin/wget -o /dev/null -O /dev/null http://$hostnames/cgi-bin/location' \
-d /<development path>
```

4. On all cluster nodes, install the DEMOhttp package that was generated by the Agent Builder:

```
pkgadd \
-d /<shared path>/<development path>/DEMOhttp/pkg DEMOhttp
```

5. On one cluster node, configure an instance of Apache using the start script that was generated by the Agent Builder:

```
/opt/DEMOhttp/util/starthttp -s -h <webserver> -p 80/tcp
```

Selecting the Type of the Generated Source Code

Source code may be generated in C or ksh. The latter is useful for those without access to a C compiler, or who are more comfortable with ksh. For ease of extensibility and richer underlying functionality, Sun encourages the use of C. Agents with ksh source code do not support non-network aware applications or applications with multiple process trees.

The fault monitoring model employed by the ksh agents is also slightly different from the C agents.

Using Agent Builder Defined Variables in the `START`, `STOP`, and `PROBE` Commands

The Agent Builder allows the use of certain variables in `START`, `STOP`, and `PROBE` command lines whose values will be substituted at runtime (i.e. when run on the cluster). This enables these commands to be tied to cluster-specific or configuration-dependent data. One such example is the hostname (or IP address) where the application will be listening and serving client requests. It is not possible to determine this value at the time of creating the agent through the Agent Builder. However, many applications require the value of hostnames to be specified on the command line. Additionally, a resource probe needs to know which server address to use to assess the health of the application. This is achieved by specifying these variables in the $*var_name* format on the `START`, `STOP`, and `PROBE` command lines. For example, as explained in “Creating Agents for the Demo Applications,” to start the echo_server, the hostname must be specified on the command line, as follows:

```
/opt/SUNWscrtw/demo/network_aware/echo_server -p <port_no> \
-l $hostnames
```

In the above example, `hostnames` is an Agent Builder-defined variable whose value will be substituted with the LogicalHostname or SharedAddress hostname (essentially an IP address) as configured in the `Network_resources_used` resource property of this resource type. If more than one hostname is configured in the `Network_resources_used` property, the value of the hostnames variable will contain all of them, separated by commas. If additional parsing is required in such a case, that may be done in a wrapper script.

Currently, `hostnames` is the only variable supported by the Agent Builder. In the future, along with support for adding new extension properties, those properties will also be available in the `START`, `STOP`, and `PROBE` command lines in the form of Agent Builder-defined variables. These variables will be accessible using the *$var_name* syntax.

Quitting and Restarting the Agent Builder

The two-step agent creation process through the Agent Builder is restartable. In other words, it's possible to quit after supplying and executing information in the first step, and restarting the Agent Builder. The rest of the information in step two can be supplied later. To do this, either the Agent Builder must be restarted from the same directory where it was run originally, or the installation directory field must be selected to point to the original directory.

Reusing Information from Another Agent to Generate a New Agent

By exploiting the capabilities described in "Quitting and Restarting the Agent Builder," it is possible to reuse the information previously entered for one agent to generate a new agent. This can be achieved by running the Agent Builder from the same directory where the previous agent was created. This results in the Agent Builder reading in all the information from the previous agent and using that information to initialize the input fields. The installation directory can be changed and modifications made to the previous values before generating the new agent.

This feature is particularly useful when a user wants to prototype a simple failover application but doesn't have a C compiler, so ksh is used instead. However, over time, ksh becomes a limiting factor in meeting the growing needs of the application, because source code generated in C is more powerful and provides access to a richer set of APIs. Not wishing to recreate the ksh agent from scratch in C, the procedure described above can be used to clone the agent into C.

Making Modifications to the Generated Source Code

Based on the input of a handful of important parameters about the target agent, the Agent Builder generates both source code and the RTR file.

To keep the agent generation process simple, the input taken by the Agent Builder is not meant to be exhaustive. To add more sophisticated features (for example, validation checks for additional properties) or tune parameters not exposed by the Agent Builder, changes must be made to the generated source code and RTR file. In addition to generating the source files and the RTR file, the Agent Builder generates the makefile with appropriate targets. The targets can be used to recompile the source code and regenerate the agent package. The Agent Builder supplies pointers to potential places where application-specific code might be added. These places are marked in the generated source code by the comments of the form:

/*** Insert your data service specific code here ***/

As described in “Using the GUI Version of the Agent Builder,” the Agent Builder creates a root directory based on the `Vendor_id` and the `Resource_type_name`, which stores everything related to the target agent.

The source files generated by the Agent Builder, based on user input, are under the `src` subdirectory under this root directory.

The binary files generated by compiling these C source files are in the `bin` subdirectory. In the case of ksh these files are identical to the ones under the `src` subdirectory.

The `util` subdirectory contains the utility scripts, which can be used to start, stop and remove an instance of the target resource type. See the section “Using the Agent Builder” for more information on utility scripts.

The `pkg` subdirectory is where the final installable package is created.

The `docs` subdirectory contains customized, text man pages for the three utility scripts.

The `etc` subdirectory contains the RTR file, which is of the form `Vendor_id.Rt_name`. For example, if `vendor_id` is SUNW and `rt_name` is ftp, then the RTR file is named `SUNW.ftp`

If any changes are made to the source code, they can be recompiled using the following command:

```
% make
```

For ksh source files, this operation involves simply copying the updated source files in the `bin` directory.

After compiling the agent source code, the package for the agent can be regenerated using:

```
% make pkg
```

As a result of the previous command, the older copy of the package in the `pkg` subdirectory is overwritten.

If changes are made to the RTR file only, then the second `make`, package creation, is the only one required.

Creating Agents for the Demo Applications

The SUNWscrtw package comes with two demo applications. One is network-aware and the other is nonnetwork-aware. Agents can be created for these applications using the Agent Builder.

Network-aware Applications

The `/opt/SUNWscrtw/demo/network_aware` directory contains `echo_server`, a simple, network-aware application that listens to a specified hostname and port number.

Echo_server Usage Message

Following is the usage message from `echo_server`:

```
Usage: echo_server [-p <port>] [-l <bind address>] [-f <logfile>] [-h]
```

Echo_server Arguments

`-l` Address to bind to (defaults to none)
`-p` Port to listen to (defaults to 5000)
`-f` Log file to write to (defaults to none)
`-h` Display usage

The same subdirectory also contains a client for `echo_server` called `echo_client`.

Echo_client Usage Message

Following is the usage message from `echo_client`:

```
Usage: echo_client -l <address> [-p <port>] [-n <iterations>] \
    [-i <interval>] [-h]
```

Echo_client Arguments

`-l` Address to connect to
`-p` Port to connect to (defaults to 5000)
`-n` Number of times to connect to the server (defaults to a continuous loop)
`-i` Time interval in seconds between connections (defaults to 0)
`-h` Display usage

When a connection with `echo_server` is successful, `echo_client` prints out the date, time, and sequence number of the packet sent, as well as the physical hostname of the server. The latter is useful to demonstrate load balancing for scalable applications.

The easiest way to install the base application (the binaries for the `echo_server)` is to add the entire SUNWscrtw package to each node of the cluster.

The following can be used as the START, STOP, and PROBE commands when creating the agent for the echo_server:

START Command	`/opt/SUNWscrtw/demo/network_aware/echo_server \` `-p <port_no> -l $hostnames`
STOP Command	`/opt/SUNWscrtw/demo/network_aware/echo_stop $hostname`
PROBE Command	`/opt/SUNWscrtw/demo/network_aware/echo_client` `-p <`*`port_no`*`> -l $hostnames -n 1`

Applications that Are Not Network-aware

The /opt/SUNWscrtw/demo/non_network_aware directory contains a simple script, print_host, that prints the date, time, and name of its physical host to syslog every five seconds. Since it does not need a network connection to perform its job, it is an ideal candidate to demonstrate the capabilities of the Agent Builder for nonnetwork-aware applications.

The following can be used as the START, STOP, and PROBE commands when creating the agent for print_host:

START Command	`/opt/SUNWscrtw/demo/non_network_aware/print_host`
STOP Command	Leave blank. Signals will be used to stop the application.
PROBE Command	As a nonnetwork-aware application, it does not have a probe command.

Installing the Demo Package

After creating the package for the demo agent, add it to all nodes in the cluster.

Then, from any one node in the cluster, use the START utility command (with appropriate arguments) to run the application under the control of the RGM.

Removing the Generated Agent's Package

There are two ways to remove the generated agent's package:

- Run the Agent Builder remove script on one node of the cluster. When it has finished, run `pkgrm` (simultaneously, if desired) on all nodes.
- Run `pkgrm` on one node of the cluster. When it has finished, run `pkgrm` on the remaining nodes.

Creating Agents with Multiple Independent Process Trees

The Agent Builder can be used to create agents for applications that have more than one independent process tree. These process trees are independent in the sense that they will be individually monitored and restarted by PMF. This is possible because each is started with its own nametag under PMF control.

To produce the agent for an application having more than one process group, a text file must be created. Each line in the text file must specify the command to start the various process trees. This text file should then be specified in the Start Command text field in the Configure screen of the GUI, or to the `configurert` command. Make sure that the text file does not have execute permissions. This enables the Agent Builder to distinguish between a simple executable script containing multiple commands, and a text file containing the commands to start different process trees.

CHAPTER **8**

Future Enhancements

Some of the features being considered for future releases of Sun Cluster include:

- **Cluster-aware APIs** that are directly callable by applications.
- **RTR file editor**. The RTR file is an important component of the generated agent package. Currently, the only way to edit and change the properties not exposed by the Configure screen is to manually edit the RTR file. A GUI-based tool is planned that will let the user edit the properties in the RTR file and also do sufficient upfront checking to ensure that the values supplied are correct and consistent.
- **New extension properties**. Currently, the only way to add new extension properties to the RTR file is to do it manually. The planned GUI-based tool will automate this process.
- **Perl**. Generation of perl code, in addition to C and ksh.
- **Runtime variables**. Access to more runtime variables in the `START`, `STOP`, and `PROBE` commands.

CHAPTER 9

References

The following references are available at docs.sun.com:

Sun Cluster 3.0 Concepts, Nov. 2000, Sun Microsystems, Inc.

Sun Cluster 3.0 Data Services Developers' Guide, Nov. 2000, Sun Microsystems, Inc.

Sun Cluster 3.0 Data Services Installation and Configuration Guide, Nov. 2000, Sun Microsystems, Inc.

Data Service Enabling Technologies, Sun Cluster 3.0 "Cool Stuff" CD, Nov. 2000, Sun Microsystems, Inc.

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

1 (800) 786.7638
1.512.434.1511

www.sun.com

February 2001