# 4

# PLANNING FOR DEVELOPMENT

## The Importance of Planning

Now that we've had a quick look at making an application highly available, we should take some time out to say a few words about planning for the development of Sun Cluster agents. As with most development tasks, the success or failure of a project to develop a data service agent is greatly affected by the amount of planning that goes into it, and we have found that a typical medium-complexity agent-development project would normally involve at least three to four days of planning and documenting, especially if the application is not already well understood.

Since ultimately data service agents provide an automated way to start, stop, and monitor applications, it is important for the developer to understand how to achieve these tasks programmatically. For this reason, an understanding of the application is usually more important than any particular experience with the Sun Cluster product. It's also very important to realize that there are certain requirements that applications must fulfill in order to be easily made into highly available services. Consequently, one of the first tasks in any project to develop a Sun Cluster data service agent is to qualify the application.

## Qualifying an Application

There are a number of characteristics used to determine whether an application can be made highly available using the Sun Cluster 3 framework. If the application does

not possess these attributes, then making it highly available may require modification to the application itself rather than simply creating start and stop methods within the framework.

Here are the key points to consider when making an application highly available.

- Data service access
- Crash tolerance
- Bounded recovery time after crash
- File location independence
- Absence of a tie to the physical identity of the node
- Ability to work on multi-homed hosts
- Ability to work with logical interfaces
- Client recovery

Let's look at each of these in detail to see what's required.

## Data Service Access

Generally speaking, the only applications that can be effectively made highly available under the Sun Cluster framework are client-server applications that receive *discrete* queries from an IP-network-based client. Examples of this type of service include databases and Web (HTTP) servers.

Applications requiring a *persistent* network connection are slightly less-suited to the HA model, since continuing the service after a failover requires a reconnection, which usually means you lose any state that was created up to the point of failure. Examples of this type of service include terminal connections, telnet, and FTP.

This does not mean that persistent-connection data services cannot get any advantage from being included in the HA framework. Because the HA failover emulates a very fast reboot of the server, the application becomes available again more quickly than it would on a stand-alone server. It does mean, however, that the client software or end user must be able to cope with this kind of break in the service. For example, an FTP client that automatically retries a download if the connection is broken and continues from where it left off would overcome this problem.

Some applications do not use any kind of network connection for client access, or may not have any kind of client access at all. In these cases, there may be some difficulty in making the application highly available using the Sun Cluster framework, particularly if some sort of fixed line (such as a serial terminal connection) is required. This is because the Sun Cluster framework has built-in mechanisms for migrating IP network addresses between nodes, but does not have a similar facility for other access methods. Fortunately, in modern computing environments, most applications subscribe to the IP client/server model or have limitations that can be worked around quite easily (see "Getting Around Requirements" later in this chapter).

### Crash Tolerance

Since the Sun Cluster HA model essentially emulates the fast reboot of a failed system, the application must be able to put itself into a known state when it starts up. Generally, this means that the application state should be committed to persistent storage (on disk) rather than being held in RAM. Depending on the application, it may be necessary for some sort of automatic rollback recovery or consistency check to be performed each time the application starts, to ensure any partially completed transactions are properly taken care of. If an application uses techniques such as a two-phase commit to write data to disk, then it is usually more easily able to recover after a system failure.

Regardless of how they write persistent data, some applications still require manual intervention to start (classic examples include applications utilizing some form of security, and that require a passphrase to be entered at startup). For these applications, it may be possible to achieve a work-around (such as piping responses in from a file). Care must be taken, however, to pay attention to the security and procedural implications of this sort of work-around.

Typically, if an application is automatically started at boot time (with a script in `/etc/rc3.d`, for example) then it is usually safe to run as a cluster-controlled application without much (or any!) modification.

### Bounded Recovery Time

When applications are restarted by the Sun Cluster framework, there needs to be some reasonable (and predictable) limit on how long it will take to recover. Obviously, the amount of time required will depend on what sort of application is involved; for example, a large OLTP database will almost certainly take longer to return to a consistent state than a small Web server due to the relative frequency of data changes.

Some limit is needed because the cluster framework needs to be able to determine when an application is *not* going to restart, which is particularly important if the cluster is attempting to restart the application on the same node. After the limit has expired, the cluster can take appropriate action (such as failing the application over to a different node).

### File Location Independence

The location of files and data used by the data service is very important, since this information has to be shared by each node that will potentially run the application. For this reason, application and configuration data locations (or file paths) should not be hard-coded into the application itself.

Applications should store any changeable data, including configuration information, on the shared storage of the cluster—either on a globally mounted filesystem, a failover filesystem, or on globally accessible raw devices. This ensures that the data and behavior of the application are consistent across nodes of the cluster. The upshot

of this is that there must be some way of defining to the application where the data and configuration files are stored in the filesystem, such as a command-line argument to the application program binary.

If the paths to data or configuration files are hard-coded into a program, then you can sometimes use symbolic links to overcome the problem. However, be aware that if an application completely removes and recreates a given file that has been redirected using a symbolic link, the behavior may not be as expected. For example, if the directory `/var/myapp/data` is actually a symbolic link to a globally mounted directory `/global/myapp/data`, then an application accessing `/var/myapp/data/foo` will get the correct file (see Figure 4.1). However, if the application unlinks the directory `/var/myapp/data` and recreates it, the symbolic link may be destroyed. This means that new data will be created in a directory `/var/myapp/data` that is not accessible to any other nodes in the cluster (see Figure 4.2).

You may also want to consider where to store your application binaries, and there are good arguments each way as to whether application binaries should be installed on shared storage or installed individually on the local storage of each node. With binaries stored locally, it is possible to perform rolling upgrades of the application software by performing the upgrade on the standby node and then manually switching control to that new node as the original node is upgraded. This maintains the availability of the data service to clients, but introduces possible management problems if a failure occurs partway through an upgrade or if the data format is different between application software releases. With binaries installed on shared storage,

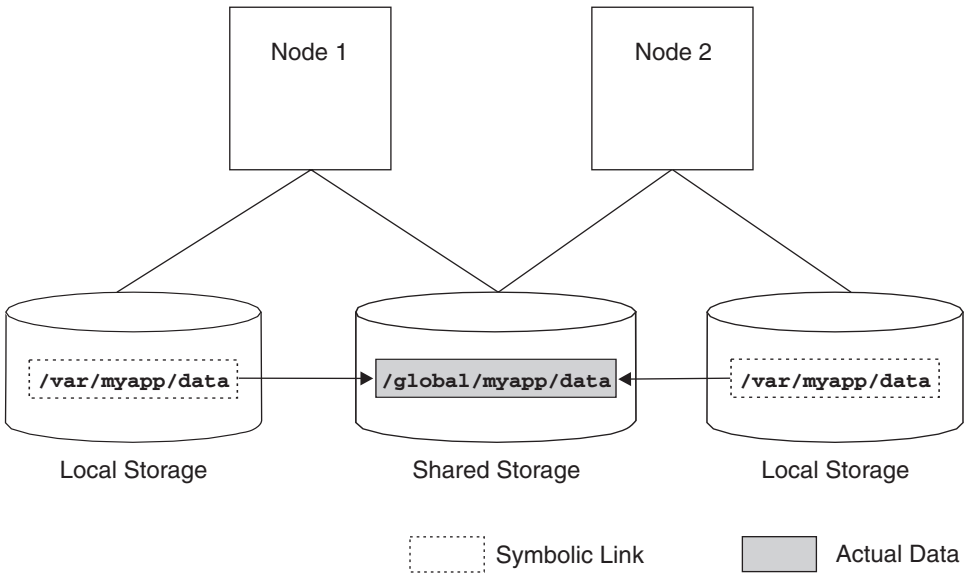


**Figure 4.1** Using symbolic links to access global data

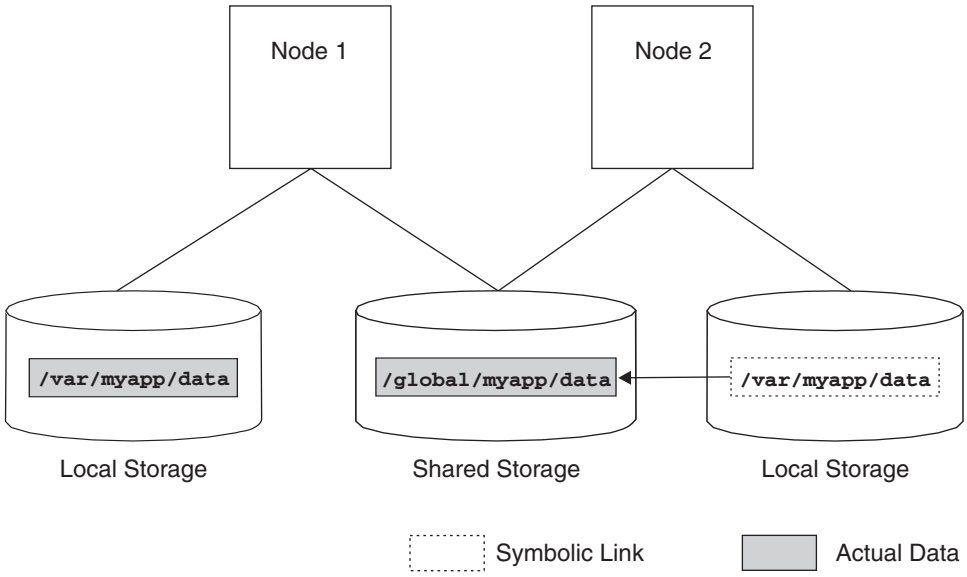


**Figure 4.2** Symbolic links after directory recreation

there is only one copy of the software to manage and an upgrade is performed by scheduling downtime for that service as the installation occurs. This consideration comes down to operational preference, rather than any technical reasoning.

## Absence of a Tie to Physical Identity of Node

Central to the understanding of Sun Cluster systems is the concept that a highly available IP address (otherwise known as a *logical host*) may operate on one of a number of physical computers at any time, and may in fact move to another computer under certain circumstances. For this reason it is important that applications be able to use the logical hostname rather than the physical name of the host on which it is running. In short, the question to ask yourself is: Can the application provide its service using a hostname that is not the physical hostname of the node?

If an application's configuration is dependent upon the physical hostname, then it almost certainly cannot be made highly available, since failing the application over to a node with a different physical hostname would render it inoperable.

If a program binds its network connection to the special address `INADDR_ANY`—a wildcard term that means the application will bind to all available IP addresses on the node simultaneously (see `in(3HEAD)`)—then this usually satisfies the requirement.

### Ability to Work on Multi-Homed Hosts

A multi-homed host is a host that is connected to more than one public network. Each node may have multiple interfaces, allowing the cluster and its data services to appear on more than one network and also allowing for hardware redundancy. An application should not assume that it must bind itself to the first network interface it can find because this may not be the correct one. In short, the question to ask is: Can the application cope with a host that has more than one network interface?

A data service that binds to a host's IP address must be flexible enough to bind to any and all of the IP addresses specified by its related logical host resource. The simplest way to do this is to have the application bind to `INADDR_ANY` (most modern network applications do).

In some circumstances, however, this approach is not desirable. Consider the case where two data services provide some service on different IP addresses but on the same IP port. If at some point both services are mastered by the same physical node, having either service bind to *all* IP addresses would mean the other could not bind to that address and the service will fail. In this case, or in the case where binding to `INADDR_ANY` is not possible, the application must have some configuration option to specify which IP address or port to bind to.

### Ability to Work with Logical Interfaces

In some instances, a cluster node may control more logical hosts (and therefore IP addresses) than it has physical network interfaces. To cope with this, the network interfaces are assigned more than one IP address, a technique sometimes called IP aliasing. IP addresses are dynamically added to and removed from physical hosts as they master the logical hosts, by adding *logical interfaces* to the physical network interfaces already on the node. Logical network interfaces are labelled the same as physical interfaces, but with an additional number part, for example:

| | |
|---|---|
| hme1, qfe3 | physical interfaces |
| hme1:1, qfe3:2 | logical interfaces |

The data service should be able to deal with a given physical interface having more than one IP address. Again, `INADDR_ANY` usually makes this an easy task, but occasionally an application will try to manipulate network traffic in particular ways that make it unable to manage more than one IP address per interface. In particular, an application may not recognize the logical portion of an interface (for example, :1), and incorrectly perform operations on the physical interface instead (for example, hme1). In these cases, it may not be possible to make the application highly available. In short, the question to ask is: Can the application cope with more than one IP address on a single network interface?

### Client Recovery

As indicated in the previous discussion of the data service access, the most effective HA data services include some capacity in the client to automatically retry a query

when the first one is cut off or times out. If an automated retry facility is not feasible, then the end user at least has to be comfortable with the concept of manually retrying a query, such as when an HTTP query from a WWW browser fails.

### Requirements for Scalable Services

If you want to make your application into a scalable service (that is, a service that operates on multiple nodes at the same time), you will have to consider a number of additional requirements. We'll leave these until we investigate scalable services in detail in Chapter 11, "Writing Scalable Services."

## Getting Around Requirements

For the most part, the requirements for applications to be made highly available are already handled in well-written Solaris applications. Unfortunately, however, there are many applications, especially those ported from other platforms or created by niche developers, that are not well written. They make so many assumptions about the environment in which they will run that they cannot be easily made into highly available data services. This problem is compounded when there is no access to the source code of the application and the creator is unwilling (or unable) to make fixes.

Nevertheless, these applications are frequently a vital piece of an enterprise IT environment, and so must be made highly available somehow. In some instances, it may be possible to circumvent the restrictions and create a Sun Cluster data service, although these work-arounds can often be complicated. Here are some approaches you may need to try to work around problem applications:

- Use the `LD_PRELOAD` environment variable to cause the runtime linker to insert custom replacements for functions called by application.[1] This is particularly useful for changing the behavior of standard function calls that result in the application getting the wrong information. One good example would be to use `LD_PRELOAD` to define a different version of `gethostname(3c)`, so that it returns the current *logical* hostname, rather than the *physical* hostname of the node.
- Use symbolic links to redirect file paths for data and/or configuration information, but keep in mind the risks of having the links replaced (see "File Location Independence" earlier in this chapter).
- Redirect startup input data from stored files, but be careful of security implications of storing passwords or similar data.

1. This is a relatively advanced subject, and the best place to find out more about `LD_PRELOAD` is the Solaris *Linker and Libraries Guide* at http://docs.sun.com.

You should also remember that it is not strictly necessary for a data service to operate over an IP network for it to be made highly available, despite the apparent emphasis on this in the preceding rules. An IP network connection is the most common way to access services on a Sun Cluster system, but it is quite possible to make other kinds of applications highly available.

## Determining Scope

Once you have determined that your application can be made highly available using Sun Cluster, you should consider the scope of the project you want to attempt. The scope can vary widely depending on the requirements of your particular situation, and it can obviously affect the amount of time it will take to develop the agent and have the data service running successfully in the cluster.

The chief topics to consider when determining scope are:

- Cluster awareness
- Simple failover or scalable
- Fault recovery
- Complex monitoring
- Enterprise architecture considerations

### Cluster Awareness

Most applications that are made highly available in a Sun Cluster framework have no internal knowledge of the cluster framework or even if they are running in a cluster. For these applications, the agent consists of wrapper programs used by the cluster to control the starting and stopping of the application. The in-memory state of the application is unlikely to be communicated between cluster nodes or maintained across failovers to different nodes.

This sort of application could be termed *cluster compatible*, since they can run in the cluster environment (having satisfied the qualification requirements discussed earlier in this chapter), but they don't change their internal behavior. They are often reasonably easy to integrate into the Sun Cluster framework.

At the other end of the scale are *cluster-aware* applications. These applications actually communicate directly with the Sun Cluster framework and change their internal behavior based on the cluster state. Furthermore, these applications may run instances simultaneously on multiple nodes of the cluster and communicate among those instances to maintain in-memory state even in the event of a node failure. One example of this sort of application is the Oracle 9i Real Application Clusters (RAC) database server. It should be fairly obvious that this sort of application is

usually quite complex and requires more complex handling to integrate it into the Sun Cluster environment.

For most projects, the application will be cluster compatible, but if you decide to make one of your own applications cluster aware, you should allow more time for the development effort. Cluster-aware applications are discussed in more detail in Chapter 13, "Developing Cluster-Aware Applications."

## Failover or Scalable

The Sun Cluster 3 environment allows for two types of data service: *failover* and *scalable*. A failover service runs on one node in the cluster at any given time and if that node crashes the application will start on a different node. To clients of the failover service, the restart will appear as if the server rebooted very quickly, but there will still be some delay (however small) between when the service stops on the first node and restarts on the second node.

By contrast, a scalable service takes advantage of the global file service (GFS) and shared IP addresses to run an application on multiple nodes at the same time, providing extra processing power through horizontal scaling and load balancing. A scalable service can also provide much better availability than a normal failover service because even if one node fails the other nodes are still running the service and can often accept new connections immediately.

In general, if you can make your application into a scalable service, you can more fully exploit the capabilities of the Sun Cluster 3 system. However, failover services more easily fit a wider range of applications without resorting to application code changes.

Scalable services are covered in more detail in Chapter 11, "Writing Scalable Services."

## Fault Recovery

When planning the scope of your data service project, you should consider the process of fault recovery, particularly when a cluster node fails.

If client applications do not retry connections to a failed service, then downtime can be extended beyond what is required by the cluster because end users will have to manually detect and recover from the failure. This should be identified during the initial phase of agent development, since it may affect the entire scope of the project.

If an application to be made highly available is itself dependent on the availability of some other service (for example, a database), then this must also be taken into account. In some cases, supporting programs must be developed to check for the availability of the service before the application is started. There may also be special requirements on the fault monitoring programs to take action (such as restarting the application) if the required service is itself failed over.

### Complex Monitoring

Fault monitoring and the associated application management can be the most complex part of the entire agent design and development process. While there are usually only very few ways to start and stop an application, there are often countless ways to monitor its behavior or remedial action if required. Questions that you should ask when determining the scope of fault monitoring include:

- Should we only check for node failure?
- Should we check if the application has crashed?
- Should we check if the application has hung?
- Should we check if the application provides incorrect data?
- How frequently should we check?
- How many retries should we allow before taking action?
- How do we detect failures outside the cluster environment?

This is where previous knowledge of an application is extremely useful, if not vital. It is also the area where the scope of the project can balloon enormously.

### Enterprise Architecture

No cluster system exists in isolation, and external elements in your IT infrastructure may affect service availability, which would render pointless any work to make the application available. Before starting work on a data service agent, you should investigate your IT infrastructure to check that there are as few single points of failure as possible, including network access, power, and so forth. You should also consider software components in your infrastructure: for example, investigating whether using a transaction processing monitor (TP monitor) would assist in speeding client recovery times.

## Gathering Application Information

Once you have determined that an application suits the cluster environment, and have determined exactly what the scope of the project will be, the next step is to gather information about the application so that you can start to build the associated resource type. At the most fundamental level, this means you will need to determine exactly how to start, stop, and monitor the application automatically.

### Start

Determining how to start the application is the most important part of developing an agent, since it is the only part of the process that the cluster framework cannot

handle by itself. You should keep in mind that starting an application doesn't just mean knowing the path to the program, but often includes a number of other components, for example:

- Command-line arguments
- Configuration files
- Environment variables
- User
- Timeout

If there is a chance of multiple instances of an application running on the same cluster, it is particularly important to pay attention to the command-line arguments and configuration files that must be used to differentiate between these instances. These variable items might then be turned into extension properties of the final resource type (see Chapter 6), so that an administrator can control these aspects of the application when configuring the cluster.

In some cases it may be necessary to write a wrapper script to start an application with the correct set of configuration variables, and this is especially true if an application must be started as a particular user other than "root." For Solaris applications that normally start at boot time, there may already be an appropriate script in the `/etc/init.d` directory that you can use directly or modify slightly to have the required effect. Remember that if you add an application to the cluster that is normally started at boot time, you should remove the original start and stop script(s) from the `/etc/rc?.d` directories.

The amount of time taken for your application to start is also important. By default, the cluster framework will wait 300 seconds (five minutes) for the application to start up before sending the first probe to check that everything is okay. If your application takes longer than this to start, then you will have to change this delay when creating the agent.

Finally, you should make a note of anything that your application depends upon, including the presence of particular data or filesystems, networks, or other applications. It may be that in order to make one application highly available, others will need to be made highly available as well. If your application depends on another, you can add to the startup code for your agent check for the availability of that service. You should also document this dependency so that administrators can configure the cluster framework appropriately so that applications start in the correct order.

Most applications start after the network has started, but in some cases it may be important to start (or partially start) applications before the network has been configured. As we will see later in the book, the cluster framework allows us to run actions before and after the network has been started, so it is important to make a note of what your application expects to happen and when.

## Stop

Stopping your application correctly is important to ensure data integrity when the service is failed over to another node. Not every failover occurs as a result of a system crash, so knowing how to stop gracefully is a vital part of the agent software.

In many cases, it is possible to safely stop an application simply by sending a `TERM` signal to the running process. In fact if you do not specify a particular program to stop your application, the cluster framework will do this by default. However, if there is a program supplied with your application that can gracefully stop it, then it should be used.

As with starting an application, there are other factors to consider than just the command name, including:

- Arguments, environment, and user
- Timeout
- Effects of `kill(1)`
- When to stop

The arguments, environment, and user factors are the same as for starting an application (see "Start" earlier in this chapter), and the timeout is similar. As with starting the application, stopping the application is by default given 300 seconds (five minutes) to be successful.

The effects of `kill(1)` on the application are important because if, when creating an agent using the SunPlex Agent Builder or Generic Data Service (GDS) tools[2] you don't supply a specific command to stop your application, the cluster framework will use `kill(1)` by default. About 80 percent of the timeout will be used to send the `TERM` signal, and about 15 percent to send the `KILL` signal. Even if you do provide a stop command (using one of the tools or not), the timeout value you supply will be used to decide how long the cluster framework will wait for an application to stop before deciding that something has gone wrong. As with the start timeout value, the default stop timeout is 300 seconds.

You should also consider whether the application should be stopped before or after the public network connection is removed. In most cases the application will be stopped before the network connection, but there may be times when you want to be sure that the network has been removed before stopping the application. However, this would normally be the case only when your application doesn't use the network directly.

## Monitor

Understanding how an application works is key to integrating it into the cluster environment. Once you have worked out how an application will behave under vari-

2. See "Choosing a Tool" later in this chapter.

ous conditions, you can create a monitor program that will check for these conditions and return appropriate values to the cluster framework, which in turn are used to decide what actions if any should be taken by the cluster.

By default, no monitoring is done by the cluster framework. However, if you use one of the provided tools[3] to create a *network-aware* resource type, a simple service probe is automatically provided. This probe just attempts to connect to the service's IP address and port, and if successful assumes that the application is operating correctly. Obviously, most applications require more complex monitoring than this, and Sun Cluster allows you to create very sophisticated monitoring applications to return different fault and failure modes to the cluster framework.

When you create a monitor program, you are in effect creating a type of expert system, in which a piece of software performs the tasks that might otherwise need to be done by a human operator. For this reason, detailed analysis of the application flow and possible failure modes will aid in the development of your resource type.

One way of tracking what your monitor program must do to check that your application is running correctly is to use a flowchart like that shown in Figure 4.3. In this example, we start by having the monitor program request a known piece of information from the application. If the request times out without any response, then the monitor program will check to see if the cluster framework is controlling the application properly, and take appropriate actions. If the request for data is successful, then the monitor program will compare the response from the application to what it expects the response to be: If they are the same, then the monitor will assume that everything is okay, and will go to sleep for a while before starting the whole process again. If the data received from the application is not what the monitor program expected, then it will check to see if *any* data was received and take appropriate action, which may be to send a warning to human operators or to restart or even fail over the application.

As you can see, it is very important to understand how your application behaves before you can write a successful monitoring program. In some cases, the application you are trying to make highly available may already have a program you can use to check the status. In such situations, you simply need to ensure that the program will return zero if the application is okay, and 1 (or nonzero) if there is a problem. In other cases, you may require quite complex programming to assess the state of the application and decide what action to take, particularly if you want to take very specific actions when certain events (such as network failures, for example) occur. The Sun Cluster APIs provide the tools to retrieve a lot of information about the cluster environment itself, but it is your understanding of the application that will determine how successful your monitor program will be.

As with starting and stopping an application, running the monitor program may require specific command-line arguments and environment variables, need a

3. Either the SunPlex Agent Builder or the GDS.

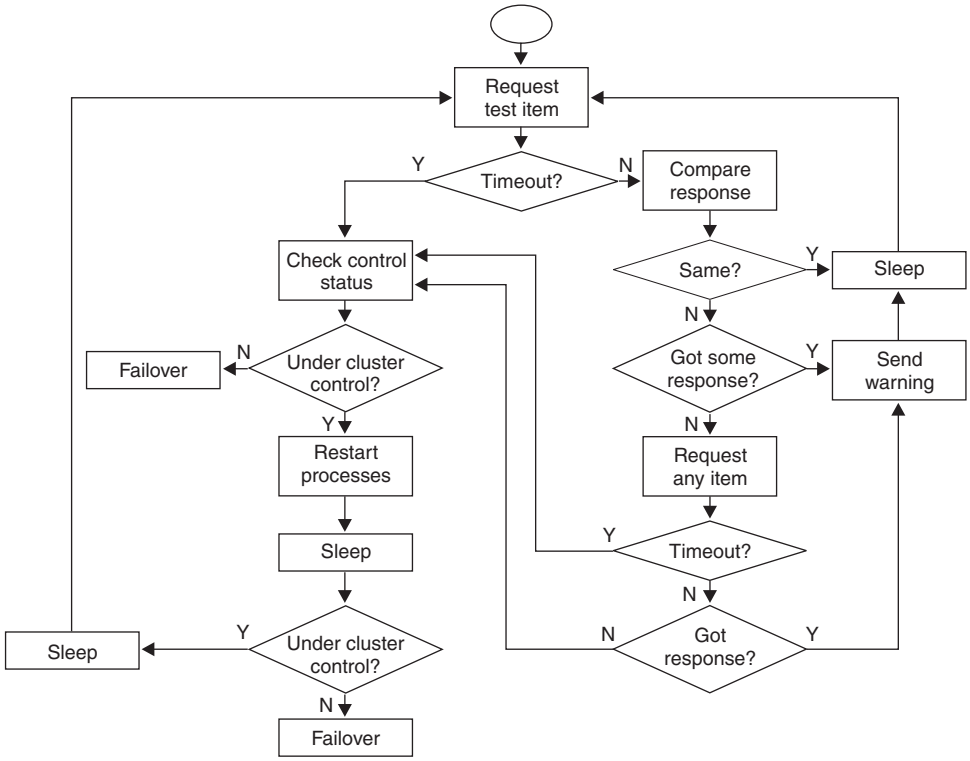


**Figure 4.3** Application monitoring flowchart

particular user ID, and require a certain amount of time to run. These factors need to be documented so that you can integrate them into your resource type.

## Choosing a Tool

Once you have gathered all of the information you need about the application, and if necessary created the wrapper scripts to start, stop, or monitor the application, you need to decide what tools to use to build the resource type itself. There are essentially three options:

- Generic Data Service
- SunPlex Agent Builder
- Develop source by hand

The one you choose will be dictated by the circumstances and requirements of the application as well as your own abilities and time constraints.

### Generic Data Service

The GDS was introduced in Chapter 3, "Getting Started." It is essentially a prebuilt resource type that can be customized at deployment time to suit any network-aware application. In most cases it should only be used for relatively simple applications, since it cannot be heavily customized.

In particular, the monitoring probe command must simply return an integer from zero for success to 100 for complete failure (a special value of 201 will cause a request for immediate failover to another node; see the man page for `SUNW.gds(5)` for details).

As we saw in Chapter 3, creating an agent using GDS simply involves creating a resource of type `SUNW.gds`, and assigning values to the properties:

```
START_COMMAND
STOP_COMMAND
PROBE_COMMAND
PORT_LIST
```

Resources and resource properties are discussed in more detail in Chapter 6, "Understanding the RGM."

When choosing a tool, it's worth considering that the GDS also generally has a more streamlined support model from Sun Microsystems because the programs that make up the GDS are part of the supported Sun Cluster packages.

### SunPlex Agent Builder

The SunPlex Agent Builder is a tool that can be used either from the command line or with a GUI to generate source code and build a Solaris package with your resource type. Since the tool generates source code, it is possible to use the Agent Builder to provide a template for further customizing. In particular, the builder generates the resource type registration (RTR) file,[4] which can be a complex undertaking by hand.

We'll be looking at the SunPlex Agent Builder in more detail in Chapter 5, "Developing with the SunPlex Agent Builder."

### Source from Scratch

If absolutely necessary, it is possible to write the code for your resource type completely from scratch, although this drastic approach is usually only really useful for cluster-aware applications.

4. The RTR file is discussed in Chapter 6.

In particular, if you decide to code from scratch, quite a few fairly repetitive functions must be coded, and the RTR file must be constructed by hand. Although this is not impossible, it can be time consuming and the combination of required, optional, defined, and undefined properties can become confusing. For this reason it's usually worthwhile to use the SunPlex Agent Builder to create a basic template, so that all of the required components are in place.

## Choosing a Language

The program that makes up a resource type can use any programming language, but two in particular are more usable—C and the Korn shell (`ksh`).

### C

The C programming language is a useful choice for developing resource type programs, since the programs themselves can be compiled and checked for errors before actually running them, which avoids the possibility of syntax errors causing problems on a live cluster.

Furthermore, there is a range of libraries available with the Sun Cluster environment that can be accessed with C, and the Data Service Development Library (DSDL) in particular can only be accessed from C (see Chapter 9, "Using the DSDL").

Additionally, when using a compiled language like C it is possible to distribute a binary-only version of your agent. This may be important if you need to retain the intellectual property or if you want to be sure that the end user of the agent cannot change the code.

When writing an agent using C, it's important to use the correct libraries and header files. The exact paths will vary depending on whether you are using the DSDL or the basic resource management API (RMAPI). The chapters describing these facilities (Chapters 7 and 9) explain how to find and specify the correct paths to libraries and headers.

### Korn Shell

Although the Sun Cluster environment provides API-like commands that can be accessed from any shell (including Bourne shell, bash, tcsh, and so forth), the SunPlex Agent Builder provides a way to automatically generate Korn shell (`ksh`) programs.

Shell programs are useful for prototyping a resource type, since they are easy to edit and modify, even on live cluster systems. In addition, it is not necessary to install a compiler to use shell programs.

On the other hand, agents developed with shell programs must be shipped to end users with readable source code (the scripts themselves), so if retaining intellectual property is a concern, or if you don't want users to be able to change the scripts, you should use C instead.

### Others

There's nothing stopping you from using another programming or scripting language to develop the programs used by your agents. In particular, you may want to use C++ instead of C or you may prefer a full-featured scripting language like Perl instead of `ksh`. If you do choose to use a different language, however, you must find your own way of accessing information about the cluster because there are, for example, no native C++ libraries or Perl modules provided at the time of writing.

## Summary

We've seen in this chapter that, as with any development project, you need to do a little planning before creating an agent. In particular, you must understand the capabilities of the application in question, since it may not be possible to make an application highly available if it has particular characteristics. Furthermore, defining the scope of your project—what the agent will and will not do—will help set realistic expectations when the system is put into production.

You should also understand how an application works: how it is started and stopped, and how it can be monitored without manual intervention. It's important to realize that understanding how an application operates is the most crucial part of integrating it into the cluster framework.

In this chapter we also looked briefly at the various tools available to you for developing agents. We'll look more closely at one of them in the next chapter, "Developing with the SunPlex Agent Builder."