Writing Scalable Services with Sun™ Cluster 3.0 Software

By Peter Lees - Asia Pacific Technical Operations

Sun BluePrints™ OnLine - October 2001

http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 816-2916-10
Revision 1.0, 10/08/01
Edition: October 2001

Please
Recycle

Writing Scalable Services with Sun™ Cluster 3.0 Software

The Sun™ Cluster 3.0 software release has introduced new features for availability and horizontal scalability, with the global file service, global networking, and scalable data services. This article is intended as a guide for system administrators and developers who are considering making an application into a scalable service using this new framework.

In addition to providing an introduction to the supporting features in the Sun Cluster 3.0 software release, this article describes both the technical requirements that should be considered when designing and programming an application to most effectively use the cluster framework, and also some of the tools available for creating scalable resources.

Scalable Services

The Sun Cluster 3.0 software release introduced the facility to control multiple instances of the same application a cross multiple nodes of a cluster. This is designed to allow access to the same data and use of the same network address, regardless of the physical node on which an instance is run. An application deployed in this manner is called a scalable service, because the capacity of the service the application provides can be dynamically increased by running the application on multiple nodes.

The other type of service possible on the Sun Cluster platform is the failover service. A failover service has one instance of an application active in the cluster at any given time. If the node on which the application is active fails, then the application is started on a new node. In contrast, a scalable service has instances of an application active on multiple nodes at the same time.

With a scalable service, instances of the application can be started on additional nodes at any time, with the facility for the extra capacity to be automatically utilized by clients accessing the cluster. Similarly, instances of the application can be stopped on any of the nodes at any time, enabling those system resources to be reallocated to different tasks, if required. In this case, clients accessing a scalable service should be automatically redirected by the cluster framework to use the reduced set of active nodes for that service.

When client systems try to access a scalable service, the Sun Cluster framework consults a load balancing policy to determine which physical node should receive the service request. Administrators can adjust the load balancing policy by applying a weighting to each of the nodes on a per-service basis (by IP address, protocol type, and port number). It is also possible to require clients to reconnect to the same physical node on subsequent requests, which may be necessary for certain applications.

Uses for Scalable Services

There are several reasons for using a scalable service in the Sun Cluster 3.0 environment:

- Increase capacity
- Increase availability and shorten failure recovery
- Manage application deployment

Increase Capacity

For some applications, increasing the number of computers running the service results in additional capacity. Architecting applications in this manner is particularly important when the application is single-threaded, and hence cannot easily take full advantage of the scalability of a single symmetric multiprocessor (SMP) system.

A scalable service increases the capacity of an application by enabling multiple copies of the application to run—while keeping the same data available across all copies.

Increase Availability, Shorten Failure Recovery

The other type of data service in the Sun Cluster environment is a failover service, which is comprised of an active node and at least one standby node. If the active node fails, one (and only one) of the standby nodes starts a new copy of the application, and accesses the data as it was left when the failure occurred. In some cases this can require integrity checks of the data, which can take a few minutes or up to several hours, depending on the nature of the application and the problem which caused the failover. During this period, between when the active node fails and the standby node is ready to respond to requests, the service is not available to client systems.

Because a scalable service has multiple instances of an application running simultaneously and accessing the same data, the failure of one node may not necessarily affect the other active nodes. Unlike a failover service, there should be no period when a scalable service is not available for new client connections. It is worth noting, however, that in this situation any sessions that have already been established with the failed node will be lost, and the client system must reconnect and possibly reissue transactions. However, client systems can immediately connect to a different cluster node, rather than having to wait for the consistency check to complete. Spreading client connections between the active nodes means the failure of one node has a lower impact on the total population of clients, improving the overall availability of the application service.

Manage Application Deployment

A scalable service can also manage the deployment of multiple copies of an application. For example, an application may be deployed on a single system during the testing phase, then added to more nodes as the project moves into the production phase.

The Sun Cluster interface to scalable services allows a single point of control for making applications available on multiple nodes.

Distributed Services and Round Robin DNS

Many Internet applications, such as DNS and IRC, already have a distributed mechanism that helps them to scale and resist single points of failure. However, there are advantages to using the Sun Cluster software instead of a distributed approach.

Although Sun Cluster software is not intended to be a complete replacement for this type of distributed architecture, using scalable services can offer an improvement in response times under certain situations. As an example, consider the case of a DNS server and a backup on a given network, with addresses 129.158.178.10 and 129.158.178.20. The `/etc/resolv.conf` file for a client on the network would read:

```
nameserver 129.158.178.10
nameserver 129.158.178.20
```

Any name lookup would first try 129.158.178.10, and then try 129.158.178.20, after a (possibly lengthy) timeout. Instead, if the DNS servers used a scalable service with a single IP address (for example, 129.158.178.11), then as long as one of the two servers was active, the client would receive a response immediately—without having to wait for a timeout. While this example does not address the problems of disaster recovery from site failure, it does illustrate how a scalable service can overcome the timeout problems associated with server failures in some distributed services.

Using scalable services can also offer a more effective solution than round robin DNS for load balancing. With round robin DNS, a given host name (for example, `www.example.org`) resolves to multiple IP addresses (for example, 129.158.178.101, 129.158.178.102, and 129.158.178.103). When the first client tries to connect to `www.example.org`, it is directed to 129.158.178.101. The second client is directed to 129.158.178.102, the third to 129.158.178.103, the fourth to 129.158.178.101 again, and so on. This approach, however, is problematic. The server to be accessed is determined at the time that the host name is resolved, causing the name server to have usually no knowledge of the host's state or active applications.

However, if the client reaches a scalable service with a Sun Cluster framework, the framework can determine which node is active and running the service in question. Thus, every request receives a response, as long as at least one instance of the application is available on the cluster.

The Sun Cluster framework provides a number of features which are vital for the development of scalable services. Applications can take advantage of most these features without requiring specific coding—meaning some normal, single-node applications can be made into a scalable service.

Detailed Look into Sun Cluster 3.0 Software

It is important to understand what facilities are (and are not) provided by the Sun Cluster 3.0 software. Effective use of these facilities can help reduce the effort required to make an application into a scalable service.

Resources and the Resource Group Manager

The interaction between applications and the Sun Cluster framework is managed by the Resource Group Manager (RGM) software facility. The RGM controls all of the logical components of the cluster that can be brought online and taken offline. These components are called resources, and examples include network connections, disk volumes, and applications.

Each resource has a corresponding resource type. A resource type can be thought of as a template for resources because it defines all of the properties of the resource, such as:

- The programs used to start and stop the application.
- Whether the application can be a scalable or failover service.
- How frequently the cluster framework should check that the application is still running.

These standard properties must be defined for all resource types. In addition, a developer can also define extension properties specific to an individual resource type. Extension properties are a convenient place to store application-specific information, such as the path to a configuration file. Resource types, and their properties and default values can be defined once in a Resource Type Registration (RTR) file. The system administrator can then create resources based on this type, supplying new values to the properties as needed. It should be noted that multiple resources of the same resource type can exist in a cluster configuration.

Resources which should act together, such as an application and its database store, should be assigned to the same resource group. A resource group is the smallest unit which can be assigned to a cluster node by the RGM. In other words, if two scalable service resources are put into the same resource group, then all of the nodes assigned to run that resource group will run both applications. FIGURE 1 illustrates the relationships between resource types, resources, and resource groups.

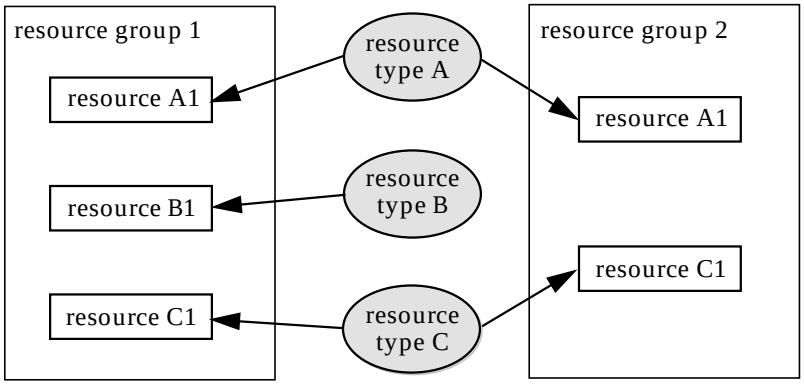

FIGURE 1 RGM Relationships

For more information regarding the RGM, consult the *Sun Cluster 3.0 Concepts Guide*.

Global Devices

Every node in the cluster can access tape, disk, and volume devices regardless of whether they are physically connected to a given node. The files in the `/dev/global` subdirectory provide a consistent path to access these devices (for example, `/dev/global/rdsk/d1s0` represents slice 0 of disk ID 1). The global device IDs (d#) make it possible to refer to any device in the cluster using a common name. For example, a SCSI disk may be connected to both host A as c1t0d0 and host B as c3t0d0, but is always referred to as d1.

Global devices may be directly accessed by applications, but such access would require careful coordination between applications trying to use the same device, and there is no facility within the Sun Cluster framework for user-level applications to manage this coordination. The Sun Cluster framework addresses this by using the global file service to provide access to common data.

Global File Service

The global file service allows data to be accessible to all nodes of the cluster using the same access path.

Although not a file system in its own right, the global file service is sometimes called the cluster file system. It is a technology to allow normal file systems (currently UFS and HSFS) to be mounted simultaneously on, and accessible from, all nodes of a cluster for reading and writing. A file system is mounted globally by using the `-g` option to `mount`(1M), or the `global` option in `vfstab`. Because the file system is mounted on the same path on each node (for example, `/global/foo`), access to the data from applications is consistent throughout the cluster.

Because the operations on the file system are proxied from the client node to the server node (the node directly attached to the storage device), the global file service is also sometimes referred to as the proxy file system (PXFS). This is important for certain disk-intensive applications, because there can be a performance improvement in locating an application instance on the server node of the file services. File system operations on the global file service are conducted as two-phase-commit transactions, and the difference in latency between performing the transaction over the cluster interconnect, rather than within the same node, can be quite noticeable. This is particularly true of append operations, in which new space must first be allocated for data before it is written (requiring another set of transactions).

Regular I/O operations (`fopen`, `fclose`, `write`, `seek`, etc) work on a global file service according to the normal file system semantics, regardless of on which node they are performed. File locking using `fcntl`(2) also works as expected, even across nodes. This mechanism can be used to provide the functionality of a distributed lock manager (DLM). It should be noted, however, that doors and pipes cannot be used for inter-process communication (IPC) across nodes by using the global file service.

To provide a POSIX-compliant environment for write operations, the file system should be mounted with the `syncdir` option. However, this can impact performance negatively. Without the `syncdir` option, the global file service could return ENOSPC errors from `close`(2) under certain failure conditions, which is the same behavior that is seen in NFS.

Storage device(s) used in a global file service should be hosted by at least two nodes, because this enables a level of redundancy to the file system. Metadata from transactions on the global file service are logged from the server node to the backup host of the file system, and use a two-phase commit model to help ensure that every operation completes properly. In the event of a failure of the server node, the backup host acquires ownership of the disk storage and replays its log of transactions, completing those that were in progress at the time of failure. The result is that applications on the other nodes do not need to specifically deal with the failover of the file system from one host node to another—although they may notice a longer than normal delay in operations while the failover is in progress.

Global Networking

Global networking enables a single IP address to be shared by all nodes of the cluster, allowing the cluster to be seen as a single entity on the network. An IP packet sent to a shared IP address is read from the network by only one node: the global interface node (GIN). The global interface (GIF) is the network interface on which the shared address is physically hosted. The GIN is the host on which the GIF currently resides. This node examines the packet and determines which node in the cluster should receive it, depending on the protocol (TCP or UDP), network port, and load balancing policies that have been applied to the scalable service associated with that address. The packet is then forwarded via the private interconnect to the instance of the scalable service on the appropriate node, which in turn responds to the request via one of its local public network adapters.

Global networking is the mechanism which allows instances of an application on multiple nodes to provide the same service. Applications must be bound to a shared IP address in order to provide a scalable service to the network.

It should be noted that each shared address resource in a cluster is itself actually a failover resource (not a scalable resource), because at any one moment, only one node will be the GIN and accept packets from the network. If the GIN fails, another node in the cluster can take over automatically.

Public Network Management and Network Adapter Failover

Although not providing a service particular to scalable services, Public Network Management (PNM) and Network Adapter Failover (NAFO) provide high availability to global IP addresses. The PNM software monitors the network adapters and will failover an IP address from one adaptor to another in the same NAFO group if it detects a failure.

PNM and NAFO are designed to be transparent to applications: a developer does not need to take them into account when determining which IP address to bind a service.

Dependencies

Many applications have dependencies upon external applications, such as databases. The Sun Cluster framework enables administrators to configure these relationships between applications, so for example, scalable service “X” will not start until resource “Y” has successfully started.

For more information, consult the man page for `scrgadm`(1M).

Weighted Load Balancing

When a scalable service resource is created on a cluster, the default load balancing for the service is an evenly distributed policy. This means that if there are three nodes on which the service is running, all three nodes will be given the same number of requests over time.

In some circumstances this even distribution between nodes is not desirable—if one node is much more powerful than the others, for example. It is possible for the administrator to alter the distribution of requests by applying percentage weightings to each node when defining a service. For example:

```
scrgadm-a -j myRes -g myRG -t myRT \
 -y scalable=true \
 -y network_resource_used=mySA \
 -y port_list=1800/tcp \
 -y load_balancing_property=LB_WEIGHTED \
 -y load_balancing_weights=10@1,30@2,60@3
```

This defines a scalable resource (`myRes`) which will receive 10 percent of requests on node 1; 30 percent of requests on node 2; and 60 percent of requests on node 3.

These weightings can also be modified dynamically while the service is in operation by using the `scrgadm`(1M) command to change the `load_balancing_weights` value. This is true even if no weightings were set when the resource was originally created.

Sticky Services

For some services, the behavior of the application precludes the use of the scalable services load balancing, even when weighting is implemented. One example is a client connection that makes use of in-memory cache on the server. Under normal load balancing, subsequent client connections may be directed to different nodes, losing the benefit of the cache. In this case, using a sticky load balancing policy ensures that all client requests to a service from the same IP address are sent to the same node (assuming the service is available on that node).

Some network services listen on a network port for incoming requests, then establish a new server on a different (random) port for each client to connect to. An example of this is a passive-mode FTP server. The problem with this method in a scalable service is that the next connection by the client may not be sent to the same node as has started the new server. In this case, the load balancing property of the scalable resource must be configured by the administrator as wildcard sticky. This requires all connections from a given IP address to be forwarded to the same cluster node, regardless of to which IP port the request is sent. However, this can upset the load balancing used by other services requested by the same client, and so the deployment of services requiring wildcard sticky load balancing should be planned carefully.

Scalable service resource types which require their load balancing properties to be sticky or wildcard sticky should have these properties set in the RTR file by the developer. Additional information regarding this and other load balancing properties is available in the *Sun Cluster 3.0 Concepts Guide* and the `scrgadm`(1M) man page.

Creating Scalable Applications

Some applications are more easily made into scalable services than others. When creating an application that will be used as a scalable service, developers should consider a number of factors. The application should:

- Have the ability to run multiple instances, all operating on the same application data.
- Provide application-level data consistency for simultaneous access from multiple nodes.
- Implement sufficient locking (via a globally visible mechanism, such as the global file system).

Applications with these above three qualities may be good candidates for becoming scalable services. Applications not demonstrating all of these qualities may need to be modified at the source level before they can be used as scalable services.

Cluster Awareness

Applications that run in a cluster environment can be classified into one of two types: cluster-aware and noncluster-aware. Cluster-aware applications include code specifically designed to work in a cluster environment, and can actively acquire and use information about its cluster during operation. An example of a cluster-aware scalable service is the Oracle® Parallel Server database, in which all database instances can communicate with each other to improve performance. Some cluster-aware applications may not need to have an associated resource type, but unless a resource type exists for an application it cannot be controlled by the Sun Cluster framework.

Any application which does not fit into the cluster-aware category is noncluster-aware. In this case, the application performs the same way in a clustered environment as it does in a nonclustered environment. An example of a noncluster-aware scalable service is the Apache Scalable Dataservice. Noncluster-aware applications require a resource type to be created for integration into the Sun Cluster framework.

Regardless of whether a given application is cluster-aware, the developer must consider many of the same factors when attempting to place it into a cluster.

Network Access Model

Because the ability of scalable services to load balance between nodes is closely tied to the global networking feature of the Sun Cluster 3.0 software release, access to a scalable service should be via the IP network, as is the case in the common client/server model. Non-IP-networks (for example, SNA) are not directly supported by the Sun Cluster 3.0 software release framework, nor are direct serial connections (such as dumb terminals).

It may be possible to fit the Sun Cluster 3.0 environment into an architecture which uses non-IP-network access modes, but normally this would be at least a moderately complex systems engineering task, and is outside the scope of this document.

File Location Independence

All files (configuration and data) must be in configurable locations; path names should not be hard-coded within the program. If they must, they should be changeable with some runtime variable, such as a command line option or environment variable.

In general, all data and common configuration files should be kept on the global file service, so that all application instances have access to the same set of data and configuration parameters.

When creating a cluster resource type for a scalable service, it is possible to define resource type properties to contain the path to a config file, or even command line options for the application. These properties can then be read and used by the cluster framework when launching the application.

Crash Tolerance

Because a scalable service is generally a type of HA service, applications should be able to resume sensibly or continue operation in the event of an application or server failure. To do this, applications may need to implement one or more of:

- Automated application recovery
- Data integrity checking
- Transaction monitoring
- Two-phase commit
- Atomic instructions
- Critical sections

Applications which implement their own clean-up procedures, and can be restarted without requiring special handling by an administrator, should have fewer problems integrating into the cluster framework.

Some of the problems of implementing crash tolerance can be avoided by following a read and write locking convention for all application file access. This can be used to prevent different instances of the application from writing or reading data in the wrong order. Locks can be placed on entire files or on records within a file, and work on a globally mounted file system the same way as one that is locally mounted.

Unfortunately, global devices cannot be effectively locked using `fcntl`(2), and there is no user-level facility within the Sun Cluster 3.0 framework to lock or reserve global devices between nodes. In order to safely access global devices, some other mechanism for coordination must be devised by the application programmer.

No Dependence on Physical Hostname of Server

Because a scalable service runs on multiple nodes at the same time, the physical hostname or IP address of a node should not be used in the application. Use `gethostbyname`(3NSL) to determine the network address for a scalable service, not `uname`(2) or `gethostname`(3C). It is also a good idea to bind network services to the specific shared global IP address of the scalable service, or at least use the INADDR_ANY wildcard. It is preferable to bind to a specific IP address, in case different versions of the same type of scalable service exist on the cluster. For example, two sets of Apache scalable services, by default, would try to bind to port 80 of all IP addresses on each node. Because only one service can be bound to a given port, different IP addresses would need to be used by each service.

Normally, the IP address, port, and protocol (TCP or UDP) for a scalable service must be defined when the scalable resource is created, to enable the cluster framework to arrange load balancing. Because this information is kept in resource properties[1], it can be supplied to the application whenever it's launched by the cluster.

Session State

The applications most easily made into scalable services are those using a sessionless client/server model, such as the http daemon. In this case, each request to the system is a discrete entity, and no state is retained on the servers. Session failure is not usually a problem in this environment, because the request can be retried without affecting the system state.

Applications which require the maintenance of state must provide some way of making the state highly available. The simplest way is to write the state to the global file system between transactions, so that any node can access the data. Depending on the application in question, more complex techniques must be used, for example remote procedure calls or shared-memory message passing.

1. NETWORK_RESOURCE_USED and PORT_LIST. See `scrgadm`(1M).

Tools For Developing Resource Types

As discussed previously, in order for an application to be controlled by the Sun Cluster 3.0 framework, a resource type for that application must be defined, usually by the developer. In this context, a resource type is made up of a resource type registration (RTR) file, the commands to start and stop the application, and (optionally) the commands used to check the status of the application. There are a range of tools available to aid the developer in the task of creating a resource type, each offering different degrees of complexity and flexibility.

Resource Management API

The Resource Management API (RMAPI) is supplied as part of the Sun Cluster 3.0 release. This C library (libscha) defines the fundamental data service operations and makes them accessible to C programs. The RMAPI can be used by Sun Cluster developers, application vendors, professional services organizations, and cluster customers to develop the callback methods (refer to the *Sun Cluster 3.0 Data Services Developers Guide* and `rt_callbacks`(1M)) of resource types, and is the same for both internal Sun developers and end customers. It is possible to install the RMAPI packages on a nonclustered system, enabling development outside of the cluster environment.

Binary program equivalents exist for each of the RMAPI functions, so that data service agents can be developed using shell scripts.

These core facilities offer fundamental access to the cluster framework, at a finer grain control than is possible with the data service development library (DSDL) API or any of the other methods described here, but are also complex to use, and require repetition of effort when developing new data service agents.

Typically, these functions are used by resource types to determine how an application should be started, stopped, monitored, and so on. However, they could also be used within a cluster-aware application to acquire information about the environment in which it is running.

The library functions are documented in the `scha_calls`(3HA) man page. The command line equivalents are documented in `scha_cmds`(1M).

DSDL API

DSDL is a convenience library for accessing information about the cluster. It is usually used in implementing the resource types for applications, rather than in the applications themselves.

The intention of the DSDL is to provide built-in error checking and to reduce the number of repetitive tasks required by the basic RMAPI. This enables the developer to concentrate on the logic of monitoring and controlling the HA application.

The DSDL is a supported feature, included as part of the Sun Cluster software, update 1. It requires the RMAPI package (SUNWscdev) in order to operate, but like the RMAPI, can be installed on a nonclustered system for development work.

SunPlex™ Agent Builder

The SunPlex™ Agent Builder is a tool for quickly building a simple scalable (or failover) resource type, given minimal information on the application.

There are two interfaces to the Agent Builder, a set of command line programs and a GUI front-end. To create a resource type, all that needs to be entered is:

- The name of the package to be built.
- The type (scalable or failover).
- The command to start the application.

Optionally, a command to stop the application and a command (returning 0 for success, 1 for failure) for probing the application for faults can also be supplied. Using this information, the Agent Builder can create C or Korn shell programs, and will create a package that can be added to a cluster using pkgadd(1M).

The source code for the resource type is included in the package, as well as some shell programs to quickly register, start, stop, and remove the resource type.

The Agent Builder has been designed to allow application developers to modify the resource type source code to provide more complex and integrated fault monitoring. The RTR file can also be modified to include additional extension properties.

The code produced by the Agent Builder is based on facilities provided by the DSDL. The DSDL package is a prerequisite for deploying resource types created with this tool, regardless of whether the resource type was built as a C program or Korn shell script.

The tool, previously known known as the Resource Type (RT) Wizard, is a supported feature of the Sun Cluster software, update 1. As with the DSDL API and RMAPI, the Agent Builder can be installed and used on a noncluster system for development.

Process Monitoring Facility

The Process Monitoring Facility (PMF) is mainly a high-availability feature, but is still relevant to developers of scalable applications. PMF is used by the cluster framework to track the execution of processes, including following any forks in execution. If the process being tracked and all of its descendants stop running, then the PMF framework will automatically take some action, such as restarting the process or calling a cleanup program.

PMF is not used within a program, but is activated at runtime by launching applications via `pmfadm`(1M). When developing a resource type for an application (scalable or failover), it is important to use PMF to provide automated recovery of an application instance within a node.

Consult the man page for `pmfadm`(1M) for more information.

Sessionless Services

This section defines sectionless services and discusses the implementation and how to access resource properties.

Definition of a Sessionless Service

A sessionless data service maintains no state in memory between client connections, and has no reference to past or other current connections. There is no interaction between sessions, either on the same node or other nodes in the cluster. If a client fails to connect to such a service, or if an error occurs part way, the client can usually reconnect and retry the operation and achieve the correct result. One example of a sessionless data service is a http server.

For sessionless services, the common point-of-reference within the cluster is the global file service. Because all nodes access the global file service using the same path, data can be shared between nodes and application instances in this manner. Because any node is able to read and write the same files, file locking should be used to ensure that consistency is achieved. File locking can be applied in exactly the same way to a cluster-aware application as to a single-node application. The only exception is that the process ID information in the flock structure returned by `fcntl`(2) is not necessarily correct for any given node. For example, if process 2345 on node 1 locks a file, the process (if returned in the flock structure for that lock) will be 2345, regardless of the node reading the lock information.

Implementing a Sessionless Scalable Service

Sessionless data services are usually the easiest applications to make scalable within a cluster. In fact, making a sessionless service scalable may not require any changes to the program itself, but merely a resource type to provide a plug into the Sun Cluster framework.

The steps needed to implement a resource type are:

- Implement the START and STOP methods for the application. This could be as simple as a shell script like those found in `/etc/init.d`.
- Create a RTR file to define the START and STOP methods, and set the “Scalable” property to “True.”
- Use `scrgadm`(1M) to:
  - Add the new resource type to the cluster.
  - Create a shared address resource that runs on the scalable service nodes.
  - Create a scalable resource group to contain the scalable service.
  - Create an instance of the new resource type, specifying the shared address resource as the hostname.
- Use `scswitch`(IM) to:
  - Start the shared address resource.
  - Start the scalable service resource.

These steps can be quickly and easily achieved using the SunPlex Agent Builder mentioned in the *Tools for Developing Resource Types* section.

To successfully deploy a scalable service in this manner, it is important that the application satisfies the requirements mentioned in the *Detailed Look into Sun Cluster 3.0* section.

Accessing Resource Properties

As previously mentioned, it is possible to access the properties of a scalable resource to determine runtime information, such as the path to a configuration file or the network port to which the application should bind.

If starting an application from a shell script, the `scha_resource_get`(1M) command can be used to find the required information. For example:

```
scha_resource_get -O PORT_LIST -R myRes -G myRG
```

This will return the protocol (TCP or UDP) and port number used by the resource `myRes` in the group `myRG`. This data can then be used as an argument to the command that actually starts the application. The resource name and resource group are automatically passed to the script (see the `rt_callbacks`(1M)).

The PORT_LIST property shown above is a standard system property that is available on all resource types. To access an extension property for a resource, it must be requested slightly differently. For example:

```
scha_resource_get -O EXTENSION my_config_file -R myRes -G myRG
```

This will return the extension property `my_config_file`.

Equivalent functions exist for accessing the resource properties using C, as shown in CODE EXAMPLE 1. The shell commands are described in `scha_cmds`(1M) and the C functions are described in `scha_calls`(3HA).

```
#include */usr/cluster/include/scha.h*
#define LOGICAL_PERNODE_B3 0xC1

struct in_addr
get_scha_ipaddress (const char *privatename)
(
struct hostent *hp = NULL;
struct in_addr res;
char **pp;

hp = gethostbyname (privatename);
for (pp = hp->h_addr_list; *pp := 0; pp++)
 (
 struct in_addr in;
 memcpy(in.s_addr, *pp, sizeof (in.s_addr));
 if (in._S_un._5_un_b.s_b3 == LOGICAL_PERNODE_B3)
 (
 return (in);
 )

)
// Should never reach this stmt
assert (0);
)
```

CODE EXAMPLE 1 Accessing Resource Properties Using C

Applications Using Inter-node Communication

Some applications require communication between nodes to effectively scale, notably those applications with an extended session in which communication between clients is required. One example of this may be a chat server, which must be able to pass messages to clients connected to other nodes, as well as the local node. This sort of application is always cluster-aware, as it must be able to determine which other nodes are running instances of the application and establish communication between instances.

In order to achieve this node-to-node communication, the cluster interconnect can be used as a highly-available network for normal IP traffic. Each node has a private hostname, identifying it on the interconnect. This private hostname can be determined using `scha_cluster_get`(3HA). Acquiring the `in_addr` structure requires some special handling (refer to CODE EXAMPLE 1). Because `gethostbyname`() returns two addresses (a pernode and a pairwise number), Sun Cluster 3.0 software requires the statement:

```
if (in._S_un._S_un_b.s_b3 == LOGICAL_PERNODE_B3)
```

The macro `LOGICAL_PERNODE_B3` is used to strip out the pairwise number, leaving the pernode number for the address. Changes to this interface are planned for future versions of Sun Cluster software.

Unfortunately the Sun Cluster 3.0 framework does not provide any generalized method for private communication between nodes through a cluster-wide interprocess communication or similar, unless specific hardware is available and the remote shared memory API (RSMAPI) is used. This means that normal inter-node IP network communication (such as RPC or sockets) must be used, and that the communication protocols must be created by the developer for each application.

To ensure there is no single point of failure, communication between nodes should use a peer-to-peer model—each node is both a client and a server on the private interconnect. This implies that threads must be used to facilitate communication between the client and server portions of the communication infrastructure.

Another problem with inter-node communication is that an application instance running on a node has to know which other nodes have instances of the same application from the same resource. Essentially, this means an application must be able to access some of the cluster metadata. This is particularly important if there is the chance of multiple sets of the same scalable service running on the cluster—each one in its own resource. There is no facility provided by the cluster framework for an application to automatically determine what resource or resource group with which it is associated. Instead, this information must be passed to the application via an external source at runtime, such as a command line argument or configuration file.

Each application instance should be aware of instances starting on new nodes, and also when instances stop. To do so, each application must maintain a list of active nodes, using the functions described in `scha_calls`(3HA).

Unfortunately, due to the nature of the private interconnect, it is not really possible to do point-to-multipoint network programming such as multicast or broadcast to implement inter-node communication.

Applications using inter-node communication may also need to share files, in the same way as the sessionless services described earlier. However, because application instances have the ability to communicate with each other, there may be some use in identifying which node has a lock on a given file.

In Sun Cluster 3.0 software, the system_id element returned in the flock structure of `fcntl`(2) can be used to determine the node owing the lock: the top two bytes give the node number. The process ID element of the flock structure refer to the process on that node. It should be noted, however, that this is not a committed interface, and may change in future releases of Sun Cluster software.

Remote Shared Memory (RSM)

The most sophisticated method of inter-node communication requires specific hardware to be usable. RSM can only be used in clusters with scalable coherent interconnect (SCI) for the cluster interconnect. This particular hardware is required to take advantages of the RSM protocols: all RSM operations must take place over this interconnect, and are not possible over Ethernet interconnects of any type.

RSM allows the direct access of memory on remote nodes. An application can export a segment of memory, which makes it available to applications on other nodes. The application instances can then use normal memory access functions to read and write the shared memory segments, making node-to-node communication very fast.

In this type of environment, the application must be cluster-aware, and can use the same strategies for determining node membership and resource information as described in the *Applications Using Inter-node Communication* section.

This API can be found on the *Sun Cluster 3.0 Cool Stuff* CD, and is unsupported software. The RSMAPI is experimental in the Sun Cluster 3.0 release, and not currently used by any commercial applications.

Conclusion

Scalable services can be written to take advantage of the facilities offered by the Sun Cluster framework, saving development effort for key areas such as global networking and global data access.

Various approaches can be taken to designing scalable applications, and the Sun Cluster framework provides a number of tools to aid in development, so long as appropriate design choices are made.

References

More information is available at the following web sites:

- *Sun Cluster 3.0 Architecture: A White Paper*

  `www.sun.com/software/whitepapers.html#cluster`
- *Sun Cluster 3.0 Concepts Guide*

  `docs.sun.com/ab2/coll.572.7`
- *Sun Cluster 3.0 Data Services Developers Guide*

  `docs.sun.com/ab2/coll.572.7`
- *Sun Cluster 3.0 software*

  `www.sun.com/software/cluster`

Acknowledgements

- Joe Bianco
- Sandeep Bhalerao
- Stephen Docy
- Evert Hoogendoorn

- Andrew Hisgen
- Skef Iterum
- Krishna Kondaka
- Christian Ramussen
- Marty Rattner
- Tim Read
- Ken Shirriff
- Hartmut Streppel
- Madhu Talluri
- Sushil Thomas
- Ashutosh Tripathi

Author's Bio: Peter Lees

Peter Lees joined Sun Microsystems in 1997 as a Systems Engineer, specializing in clustered systems and high availability. Since 1999, he has worked as the Asia-Pacific Cluster Technology Consultant, assisting customers and Sun Systems Engineers throughout Asia, including Australia, China, India, Japan, Korea, New Zealand, and Southeast Asia.