W h i t e p a p e r

now from

VCS 4.0 Agent Development by Example

Jim Senicka, Tom Stephens, and Eric Hennessey
Server and Storage Management Group

November 2004

TABLE OF CONTENTS

EXECUTIVE OVERVIEW

This document is intended for use by Independent Software Vendors interested in increasing application availability through the use of VERITAS Cluster Server (VCS). The intended goal is to provide information on developing support for applications within the VCS framework.

This guide provides basic instruction on how VCS uses agents to control applications by providing agent design concepts, “constructing agents by example” sections and a complete best practices guide for agent development.

Primarily intended for vendors wishing to create agents in shell scripts or Perl, this document does not delve into the development of VCS agents with C++. For detailed information on developing agents in C++, refer to the VCS Agent Developers Guide included with the standard VCS documentation.

DETERMINING APPLICATION CLUSTERING ABILITY

The first question that must be answered concerning an application is the application’s suitability to operate in a clustered environment. This does not mean the application must be “cluster aware”. It simply means the application follows some very basic principals for how it operates in a single server environment. Nearly all commercial applications can be placed under cluster control, as most meet the following guidelines:

DEFINED START PROCEDURE

The application must have a defined procedure for starting. The VCS agent developer must be able to determine the exact commands used to start the application, as well as any other outside requirements the application may have, such as mounted file systems, IP addresses, and so on.

For example, to start an Oracle database the VCS agent developer must know that this entails calling the server manager process with the correct Oracle user, and providing the correct Instance ID, Oracle home directory, and pfile.

The developer must also be able to clearly determine any outside environmental variables expected in the user’s environment when starting the application. To continue using the Oracle database example given above, the VCS agent developer must know how to set the LD_LIBRARY_PATH environmental variable prior to starting the database.

DEFINED STOP PROCEDURE

The application must have a defined procedure for stopping. This entails the ability to stop an individual instance of an application without affecting other instances.

Using an Apache web server for example, terminating all HTTPD processes on the system is unacceptable since it would stop any web server instance currently running. With Apache 1.3, the documented process for shutting down a specific web server instance involves locating the PID file written by that instance as it was started. The PID contained within this PID file is then sent a TERM signal. This will cause the master HTTPD process for that particular instance to halt all child processes, without affecting any other instance.

DEFINED MONITOR PROCEDURE

The application must have a defined procedure for monitoring the health of an individual instance with a high degree of confidence. In order to do this, the process for monitoring an application may involve several levels or procedures.

Using the Apache web server as an example, simply checking the system process table for the existence of "httpd" is unacceptable, as this would not uniquely identify an individual web server instance. A better solution would be to verify that the PID contained within the PID file exists within the system process table. To further increase the confidence of the monitoring procedure, the identified process should be examined to verify that it is the correct process, and that the system has not simply reused the PID contained within the PID file.

Additionally, it may be desired to perform even more procedures to verify an application's functionality. Given the example, simply verifying the existence of a process within a system process table may not provide reliable monitoring results. Situations where this may occur could be with a hung process. In such a situation, the process may exist within the system's process table, however would be unresponsive to user interaction. With a web server, this could be verified by connecting to the correct IP address and Port and testing if the web server responds to standard http requests. In a database environment, this may involve connecting to the database server and performing a series of SQL commands to verify the ability to read and write to the database.

In both cases, end-to-end monitoring is a far more robust check of application health. The closer the monitoring procedure comes to mimicking exactly what a user does the better that procedure is in identifying problems. This degree of monitoring does come at a price, however. End-to-end monitoring may increase system load and system response time. From a VCS agent design perspective, the level of monitoring implemented should be a careful balance between assuring the application is up and minimizing the impact on the system or application.

DEFINED CLEANUP PROCEDURE

In many cases, a method to "clean up" after an application must also be identified. If the VCS agent for an application is incapable of stopping the application in a clean manner, it may require the use of a more forceful method. For an example, if an Oracle database hangs and will not respond to a 'shutdown immediate' command, it may be required to utilize a more forceful method to stop the database. This may involve using a 'shutdown abort' command or sending a TERM signal to the specific Oracle processes for that particular instance.

In the event that a database is forcefully shutdown or fails unexpectedly, it may not have the ability to clean up resources used such as memory segments or semaphores. This may potentially cause an issue for the database if the database is to be restarted on that server.

The VCS agent developer must be able to identify what steps are required to forcibly terminate a particular application instance, and return the server to a known state where the application may be restarted on that system with a minimal of problems.

EXTERNAL DATA STORAGE

For all applications that expect to use data stored on disk, the application must be capable of storing all required data on shared disks. This may require specific application configuration options or the use of symbolic links.

For example, an application may only be able to be installed under /usr/local. This would require either symbolically linking /usr/local to a file system mounted from the shared storage device or actually mounting file system from the shared storage device onto /usr/local.

As an alternative to shared storage devices, the application may store data to local storage if that data store can be successfully replicated to storage on another node in the cluster. This is known as a replicated data cluster.

On the same note, the application must store data to disk, rather than maintaining it in memory. The take over system must be capable of accessing all required information. This precludes the use of anything inside a single system not accessible by the peer, such as NVRAM accelerator boards and other disk caching mechanisms contained in a local host. Disk caching for performance is acceptable, but it must be done on the external array and not on the local host.

RESTARTING TO A KNOWN STATE

The application must be capable of being restarted to a known state. This is probably the most important application requirement. During a switchover, the application is brought down under controlled conditions and started on another node. The requirements at this point in time are fairly straightforward. The application must close out all tasks, store data properly on shared disk and exit. At this time, any peer systems can startup the application from a clean state.

The problem scenario arises when one server crashes and another must take over. The application must be written in such a way that data is not stored in memory, but regularly written to disk. A commercial database, such as Oracle, is the perfect example of a well-written, crash-tolerant application. On any given client SQL request committing a change, the client is responsible for holding the request until it receives an acknowledgement from the server. When the server receives a commit request, it writes the changes to a log file, or “redo” file. This data is confirmed as being written to stable disk storage before acknowledging the client. At a later time, Oracle then de-stages the data from the redo log to the actual table space in a procedure known as check pointing. After a server crash, Oracle can recover to the last known committed state by mounting the data tables and “applying” the redo logs. This in effect brings the database to the exact point of time of the crash. The client resubmits any outstanding client requests not acknowledged by the server; all others are contained in the redo logs. One key factor to note is the cooperation between client application and server. This must be factored in when assessing the overall “cluster compatibility” of an application.

To look at this a different way, can the application in question come up properly if the server hosting the application were to fail? How about if the storage were plugged into a new server and the application started on the new server? If both of these questions are answered yes, then the application should work acceptably in a cluster.

LICENSING, HOSTNAMES, AND OTHER COMMON ISSUES

The application must be capable of running on all servers designated as potential fail over targets. This means no license issues, host name dependencies or other such problems would prevent the application from being started.

A common issue experienced is that an application’s license may be tied in some manner to the hostname of the system. Many customers have attempted to use scripts to change hostname on failover, however this typically causes far more problems than it solves. Changing hostnames can lead to significant management issues if multiple systems wind up with the same hostname after an outage. It is best to configure applications and obtain proper licensing so that an application may run on all hosts configured as fail over targets for that application.

Additionally, to facilitate the ability of an application to fail over between servers, the application must not be tied to an IP address which is statically linked to a particular system. Instead, it should be configured to use what is known as a virtual IP, or VIP. The VIP would be configured to fail over between the nodes within a cluster with the application.

Any required user or system accounts required for the operation of the application must exist on all fail over targets within the cluster. The permission settings for these accounts must be set the same across all fail over targets.

APPLICATION COMPATIBILITY SUMMARY

To run properly in a cluster, an application must be crash tolerant and host independent. This means the application will recover after a crash to a known state, in a predictable and reasonable time. It must be capable of doing so on a different host than where the crash occurred. From a very simplified perspective, clustering provides the ability to replace a failed host with a functioning host in a matter of seconds and restarting a given application. The application must support this action with no problems.

The application must provide a method to start, stop and monitor a specific instance of the application to properly support agent startup, monitor and shutdown functions.

The application must run well with other applications, or the developer must provide a complete set of exceptions.

All in all, most commercial applications written recently should not have problems with these criteria. Applications not capable of meeting these simple needs are not compatible with modern, cost effective, high availability architectures and may require redesign to properly support customer needs.

VCS CONCEPT REVIEW

VCS agents provide a very powerful capability to easily control a wide array of hardware and software resources. The abstraction of the agent framework makes it simple for a developer to provide support for new and changing applications. The following section provides details on several basic VCS concepts required to be understood prior to the development of a specific agent for an application.

RESOURCES

Resources are specific hardware or software entities, such as disks, network interface cards, IP addresses, web servers and databases. VCS manages the availability of resources within a cluster. This involves the ability to online (starting), offline (stopping), and monitor the health or status of the resource. Each resource is identified by a unique name within the cluster. Each specific resource contained within a VCS configuration is of a specific *resource type*.

RESOURCE TYPES

Resources are classified according to their *resource type,* and multiple resources can be of the same resource type. For example, a VCS configuration may contain two IP resources. Although these resources may define unique IP addresses, both resources are classified as an *IP resource type*.

How VCS manages a resource is specific to the resource type. With multiple IP resources defined for example, each resource is brought online by configuring the specified IP address on a specific network interface card. The procedure to perform this action does not change for each resource, only the attributes defining the specific IP address do.

For each resource type, VCS has a corresponding *agent*. The agent provides the resource type specific logic to manage resources.

RESOURCE TYPE CLASSIFICATIONS

Different types of resources require different levels of control. All agents fall into one of three categories (OnOff, OnOnly, or Persistent) depending on the functional characteristics of the agent. These categories are described in further detail below.

On-Off

Most resource types are classified as “OnOff” resources. These resources are resources that may be able to be brought online and offline. The FileOnOff resource type is one example of an OnOff resource. This particular resource type will create a file when brought online, monitor for the existence of the file, and delete the file when taken offline.

On-Only

Resources defined as OnOnly resources have the ability to be brought online and monitored by VCS, however are not able to be taken offline through VCS. One example of this type of agent is the NFS agent. When a NFS resource is configured and told to go online, VCS will online the NFS server process and monitor it periodically to ensure it remains online. However, when the VCS service group containing the NFS resource is told to go offline, the NFS server processes are not stopped.

Persistent

Persistent resources are resources that can not be brought online or offline, yet they must be monitored by VCS. The best example of this type of resource would be a NIC resource. As a NIC can not be brought online or offline through the operating system, VCS can not perform these functions either. However, the NIC does need to be monitored to ensure that it is able to function properly; else services contained on the system may be disrupted.

AGENTS

The actions required to manage an application differ significantly for different types of resources. VCS handles this functional disparity between different types of resources by using agents unique to each resource type. This makes it simple for application developers to integrate additional types of resources into the cluster framework.

An agent is an installed program designed to manage a particular resource type. VCS agents are “multi-threaded” which allows a single VCS agent to manage multiple resources of the same resource type on one host. For example, the Disk agent is capable of managing all Disk resources configured.

VCS AGENT ANATOMY

Agents are comprised of a series of discrete components. In this section, the basic functional components that comprise an agent are discussed.

VCS AGENT FRAMEWORK

The VCS Agent Framework is a set of predefined functions compiled into every agent that is common across all agents. This includes the ability to connect to the VCS engine and to understand common configuration attributes, such as the RestartLimit attribute. When building an agent in C++, the Agent Framework is compiled in with an include statement. When developing VCS agents in a script language such as shell or Perl, a precompiled binary called ScriptAgent, is used. This precompiled binary contains all the agent framework functions built in. Most of the complexity of an agent is handled by the framework and should not concern the agent developer.

When an agent starts, it performs a routine known as VCSAgStartup that is a part of the compiled framework. This startup routine initializes the agent data structures and connects to the VCS cluster engine. The agent then receives all necessary information it needs about configured resources from the engine. The agent then can manage a given resource via the agent entry points.

ENTRY POINTS

Agents carry out specific functions on resources on behalf of the cluster engine. The individual functions an agent performs are known as “Entry Points” or EPs. Agent entry points are sections of code to carry out specific functions on a resource, such as online, offline and monitor.

Entry points can be compiled into the agent itself, however in most circumstances, the entry points are implemented as individual Perl or shell scripts. This facilitates ease of modification and implementation for the agent.

There are several agent entry points that may be used when developing a VCS agent. Each of the available entry points are described below:

Online

The online entry point is used to bring a specific resource online from an offline state. It contains the code or commands that will perform the actual online function for the application.

This entry point is only responsible for bring the specified resource online. It is not responsible for monitoring the resource, or ensuring that the procedures performed within the EP were successful in bringing the resource online. This function is reserved for the monitor EP.

Offline

The offline entry point is used to bring a specific resource offline from an online state. It contains the code or commands that will perform the actual offline function for the application.

This entry point is only responsible for bring the specified resource offline. It is not responsible for monitoring the resource, or ensuring that the procedures performed within the EP were successful in bringing the resource offline. This function is reserved for the monitor EP.

Monitor

Determining if a resource is online or offline is the responsibility of the monitor entry point. This fact makes the monitor entry point the most important entry point used. Faulty code within the monitor entry point can produce false positive and false negative conditions. It is absolutely critical that the monitor entry point be thoroughly tested to prevent these conditions.

The monitor entry point is called periodically to verify the state of a resource. Under normal circumstances, the monitor is executed every 60 seconds when a resource is online, and every 300 seconds when a resource is expected to be offline. These values may be adjusted by the user to suit a particular environment.

In addition to being used to periodically monitor a resource's state, the monitor entry point is also used to determine the state of a resource at specific times. As an example, this entry point will be called during the initial start of VCS on a node to 'probe' or determine the status of any resources on that node. It is also used to validate that the online and offline entry points were successful in their operation after they have finished.

Clean

When a resource fails to come online, fails to go offline, or fails while in an online state the clean EP is called. The clean entry point is designed to bring down the application, forcefully if need be, and then "clean up" afterward to ensure the system is returned to a valid state. As an example, the clean function may remove shared memory segments or IPC resources left behind by a database after the database has been forcefully brought down.

Action

New to the VCS 4.0 agent framework, the action entry point enables an agent developer an additional level of flexibility. Unlike most of the other entry points, the action entry point is not called automatically by the VCS engine. Instead, it is invoked through the use of the 'hares' command.

It is used to perform specific actions in a short amount of time which do not involve any offline or online action. Typical actions that may be consist of backing up a database, placing a database in a restricted mode, or changing the direction of data replication.

Info

Also new to the VCS 4.0 framework is the info entry point. Used to obtain information about a online resource, it too is invoked from the hares command. Examples of how the info entry point can be used may consist of reporting the amount of free space available for a Mount resource. Or it could be used to check the status of replication for a given RVG resource.

Attr_changed

Although this entry point is not commonly used, it provides a method to react to a change of a resource's attributes. Attributes that will trigger the calling of this entry point when modified must be registered with the agent framework.

Open

Another one of the less common entry points used is the open entry point. Called whenever the VCS engine starts to manage a resource, this entry point can be used to perform actions prior to any of the other entry points. When configured, examples of when this entry point would be called would be when a resource was enabled, or when VCS first starts the agent.

Close

The opposite of the open entry point, this entry point is called when VCS stops managing a resource. Like the open entry point, this entry point is not commonly utilized.

Shutdown

Invoked before a VCS agent shuts down, this entry point is the last entry point available to the VCS agent developer. It is rarely used in a VCS environment.

TYPE DEFINITIONS

A resource type definition describes the information needed by an agent to control a resource. For C programmers, the type definition can be considered similar to a header file. It defines the data types of variables to provide error checking as well as provides the order that the variables (called attributes) are passed to the entry points. Bundled with the VCS product is a types.cf resource type definition file which describes standard resource types to the VCS engine. Other resource type definition files may be used to provide additional functionality for the VCS engine to manage new or different resource types.

AGENT INFORMATION FILES

With the release of VCS 4.0, a new feature was added to assist agent developers in the presentation and management of agent data and functions from within the VCS GUI. This feature involves the creation of a file known as an Agent Information File for each agent. This file is placed within the agent directory and must be named with the name of the agent followed by .xml (<agent_name>.xml).

Each Agent Information File contains information that defines the arguments that can be presented for that agent, as well as can define actions that may be disabled or enabled.

Creation of an Agent Information File is not a required component of a functional VCS agent. Without one, an agent can still be managed through the VCS GUI. Of course though, the additional features that are provided with the use of the Agent Information File will not be available for use.

MESSAGE CATALOG FILES

Message Catalog Files are used to provide the agent developer to create an agent that can be internationalized. This means that the messages that the agent will log can be in a language that matches the localization settings selected by the user.

There are two types of Message Catalog Files: Binary Message Catalogs (BMCs) and Source Message Catalogs (SMCs). Source Message Catalogs are plain text files that define all the messages that a given agent may output with an associated message id number. Binary Message Catalogs are generated from the information contained in the SMC.

Like Agent Information Files, a Message Catalog File is not a required component of a functional VCS agent. Agent developers should plan for the eventual use of a Message catalog File when creating any agent. Although it may not seem apparent during the early stages of agent development, requirements often change. A little prior planning will make the agent developer’s task easier in the event a change is required.

FIRST GENERATION AGENT EXAMPLE

The following section will detail the construction of a very simple VCS resource. To demonstrate the desired concepts, the FileOnOff resource type will be used as an example. As a simple resource type, the FileOnOff resource is designed to create a file with a specific name in a specific location when brought online. Monitoring of the resource consists of testing for its existence. Bringing the resource offline simply removes the file.

In the shipping version of VCS, the actual FileOnOff agent is implemented as a compiled C++ agent. For the sake of demonstration here, it is shown implemented with shell script entry points.

TYPE DEFINITION

The FileOnOff agent requires very little information. All that must be specified is the filename, including the full path. The required information for the type definition is demonstrated in the example provided below.

```
type FileOnOff (
       static str ArgList[] = { PathName }
       str PathName
)
```

The type definition uses the keyword "type" and the name of the resource type. In this example, the name of the resource type is 'FileOnOff'.

The type definition provides two very important functions. First it defines the sort of values that may be set for each attribute. With this example, the PathName attribute is defined as being able to contain values that are strings. This is specified by the use of 'str' (short for string) preceding the attribute name. The 'string' data type is one of many data types available for use. These are described in the VCS users guide in the section "VCS Attributes".

Second, the ArgList defines the order attributes are sent to the corresponding entry points. Using this simple example, the PathName attribute is the only attribute. More complex resource types contain more attributes.

When VCS wishes to online a FileOnOff resource, it will call the FileOnOffAgent with the online command, the name of the resource, and then the contents of the ArgList.

The agent strips off the command and passes the remaining arguments to the proper entry point. For example, the FileOnOffAgent would pass the name of this specific FileOnOff resource and the value of the PathName attribute to the online entry point. The name of the specific FileOnOff resource is pre-pended to the ArgList automatically by the VCS engine. There are situations with other entry points where the VCS engine will pre-pend additional information to the ArgList string. The important point to understand, however, is the contents of the ArgList are always presented in order as they are specified in the type definition file. This means that in this example, the first ArgList value is actually $2 in a shell script for the online entry point. If there were additional values in the ArgList, they would continue as $3, $4, $5 and so on.

RESOURCE DEFINITION

The individual resource is configured in the VCS configuration file, main.cf. It includes the name of the specific resource and any required attributes. Given the FileOnOff example provided, a resource definition may look like the following:

```
FileOnOff tmp_file01 (
     PathName = "/tmp/file01"
)
```

The resource definition in the configuration file specifies the resource type to be used (FileOnOff), gives a unique name for the resource (tmp_file01) and provides the value for the PathName (/tmp/file01) attribute. The cluster engine will obtain information from the resource definition, and pass this information to the agent based in the requirements in the resource type definition.

Notice that the value given to the PathName attribute is contained in quotes. This is to prevent the shell or interpreter from parsing the value in an undesirable manner.

AGENT ENTRY POINTS

The following provides sample code for each of the entry points to be used in the example FileOnOff resource. The examples given are purposely kept very simplistic, as they are only used to demonstrate certain concepts. It is important to keep in mind that if these were to be used as actual agent entry points, certain modifications would be desired. Logging, error checking, and code comments would be some of the desired modifications. These topics will be discussed later in this paper.

Online

The online entry point for this example of a FileOnOff resource is very simple. It expects the name of the resource as the first argument and the pathname as the second. It then creates the file in the specified path.

```
#!/bin/sh
RESNAME=$1
PATHNAME=$2

touch $PATHNAME

exit 0
```

Offline

The example offline entry point shown below accepts the same ArgList and deletes the file specified by PathName.

```
#!/bin/sh
RESNAME=$1
PATHNAME=$2

/bin/rm –rf $PATHNAME

exit 0
```

Monitor

The monitor entry point in this example accepts the same ArgList. Its function is different than the online entry point as it is designed to check if the file exists or not. If the file exists it returns an exit code of 110, which represents that the resource is online. However, if the file does not exist, the monitor entry point will exit with a value of 100 which signifies that the resource is offline.

```
#!/bin/sh
RESNAME=$1
PATHNAME=$2

if [ -f $PATHNAME ]; then exit 110;
   else exit 100;
fi
```

FILE PLACEMENT

After the above files have been created, they need to be placed into the correct location in order to be utilized by VCS. The type definition file is placed within the VCS configuration directory. This is the same location where the main VCS configuration file (main.cf) is stored.

The rest of the files will need to be placed in a directory created under %VCS_HOME% which is named the same as the name of the Agent. In this example, the script files would be placed under %VCS_HOME%/bin/FileOnOff.

AGENT OPERATIONS

This section will describe some of the basic operations that an agent performs. The concepts presented here are critical to the creation of a proper VCS agent.

ENTRY POINT SEQUENCE

The agent calls the entry points in a specific sequence depending on the nature of the action being carried out on the resource.

Initial Probe

When VCS first starts, the agent will run an initial monitor sequence referred to as a ‘probe’ and return the status of resources to the engine. This provides the engine with a baseline of the status of all resources in the cluster.

Bringing Resources Online

During an online operation, the agent calls the online entry point to bring the resource online. After the online entry point completes, the monitor entry point is called to test if the online function was successful. It is the responsibility of the monitor entry point, not the online entry point, to test if the online was successful.

If the resource fails to come online, and the agent is configured to retry the online operation, (OnlineRetryLimit > 0) the clean entry point is called and the online is attempted again.

Bringing Resources Offline

When performing an offline operation, the agent calls the offline entry point to bring the resource offline. After the offline entry point completes, the monitor entry point is called to test if the offline function was successful. It is the responsibility of the monitor entry point, not the offline entry point, to test if the offline was successful.

If the monitor reports the resource is still online after the offline entry point has been run, the clean entry point is called. Following this is another call of the monitor entry point. If the monitor still determines that the resource is online, the cluster declares an error and sets the resource status to “cannot offline”.

Periodic Monitoring

The VCS engine will have the agent call the monitor entry point periodically to ensure a resource remains in the state intended. By default, resource monitoring occurs every 60 seconds when the resource is expected to be online and every 300 seconds when the resource is expected to be offline. These values are configurable by the VCS user.

Resource Failure

If a resource is online and during one of the periodic calls of the monitor entry point, the monitor detects it is offline without having been commanded to go offline by the engine, this is considered to be a fault. In this case, the clean entry point will be called to ensure that the resource is fully offline and able to be restarted, if need be. Depending on the setting of the RestartLimit attribute, the resource may stay faulted, or be restarted.

USE OF ATTRIBUTES

Variables or “attributes” are used to control resources within a cluster. Values are assigned to these attributes which are then passed to the appropriate agent entry point for action. The use of attributes provides a high degree of flexibility for VCS.

ATTRIBUTE CATEGORIES

Attributes are categorized by their scope of control within the cluster. Some attributes only configure a specific resource, some affect the behavior of a resource type, and some are applicable to all resource types. The categories that an attribute may be belong to are:

Type Independent Attributes

Type Independent Attributes consist of attributes that can be defined for a resource, regardless of the resource type of the resource. These attributes are coded into the agent framework when the agent is developed. Attributes such as RestartLimit and MonitorInterval are examples of Type Independent Attributes. These attributes can be used with all resource types, and when set, will affect all resources of that resource type.

To provide an example, in order to change the value of the Type Independent Attribute 'MonitorInterval' from the default 60 seconds to thirty seconds for all resources of the FileOnOff type, it can be defined as follows:

```
type FileOnOff (
       static str ArgList[] = { PathName }
       str PathName
        static int MonitorInterval = 30
)
```

One important item to remember is that this will affect all resources of the particular resource type (FileOnOff). It will not affect the resources of different resource types.

Type Dependant Attributes

Type Dependant Attributes are those attributes, which only pertain to a particular resource type. These attributes are differentiated from Resource Specific Attributes (described below) as they are typically assigned values within the resource type definition. Attributes in this category are configured to be passed to the agent as part of the ArgList.

Consider the following example of a resource type definition:

```
type DiskGroup (
       static int OnlineRetryLimit = 1
       static str ArgList[] = { DiskGroup, StartVolumes, StopVolumes, MonitorOnly,
MonitorReservation, tempUseFence }
       str DiskGroup
       str StartVolumes = 1
       str StopVolumes = 1
       static int NumThreads = 1
       boolean MonitorReservation = 0
       temp str tempUseFence = "INVALID"

)
```

Notice the definitions for the StartVolumes and StopVolumes attributes. These are type dependant attributes, as they are really only applicable to the DiskGroup resource type definition. As they are assigned a value within the types definition, these values will affect all resources defined of the resource type 'DiskGroup'.
These values can be modified from within the resource type definition, such as:

```
type DiskGroup (
       static int OnlineRetryLimit = 1
       static str ArgList[] = { DiskGroup, StartVolumes, StopVolumes, MonitorOnly,
MonitorReservation, tempUseFence }
       str DiskGroup
       str StartVolumes = 0
       str StopVolumes = 1
       static int NumThreads = 1
       boolean MonitorReservation = 0
       temp str tempUseFence = "INVALID"

)
```

From this example, one can see that the StartVolumes attribute's value was modified from a '1' as shown in the previous example, to a '0' in this example resource type definition.

Some attributes can have their values overridden on a per resource level. To do this, the attribute must be defined in the VCS cluster configuration file (main.cf) under the resource where the value is to be overridden. Continuing with using the DiskGroup resource type examples previously presented, the following example demonstrates a sample resource definition within a main.cf file. In this example, a resource of the DiskGroup resource type is defined:

```
…
DiskGroup shared_dg1 (
             DiskGroup = shared_dg1
             )
…
```

By not providing a value for the attributes StartVolumes or StopVolumes, the value assigned to these variables within the resource type definition are used as the default value for the attribute. However, if values are assigned as demonstrated below:

```
…
DiskGroup shared_dg1 (
             DiskGroup = shared_dg1
             StartVolumes = 1
             StopVolumes = 1
             )
…
```

The values defined in the resource type definition are overridden, just for the resource (shared_dg1) specified. Attributes that are defined as 'static' attributes may not be overridden. Static attributes are identified within the resource types definition by the keyword 'static'. Observe the lines for the StartVolumes and StopVolumes attributes in the following example:

```
type DiskGroup (
      static int OnlineRetryLimit = 1
      static str ArgList[] = { DiskGroup, StartVolumes, StopVolumes, MonitorOnly,
MonitorReservation, tempUseFence }
      str DiskGroup
      str StartVolumes = 0
      static str StopVolumes = 1
      static int NumThreads = 1
      boolean MonitorReservation = 0
      temp str tempUseFence = "INVALID"
)
```

A difference between the two attributes can be identified, as one (StopVolumes) is defined as a static attribute, which can not be overridden, while the other (StartVolumes) is not.

Resource Specific Attributes

Resource specific attributes are attributes which pertain to a given resource only. As mentioned previously, the main deference between resource specific attributes and type dependant attributes is that resource specific attributes are defined for each resource within the cluster configuration file (main.cf). The following example demonstrates the use of a resource specific attribute (PathName) from a main.cf except:

```
FileOnOff tmp_file01 (
      PathName = "/tmp/file01"
)
```

Local and Global Scope of Attributes

Attributes can be defined as local or global in scope. *Global* attributes are defined as attributes whose assigned value applies to all systems. On the other hand, attributes whose value applies on a per-system basis is defined as *local* in scope*.* By default, attributes are global in nature. To define an attribute as local, the "at" operator (@) is used to indicate the system to which a local value applies. An example of local and global attributes can be found in a typical MultiNICA resource definition.

```
MultiNICA mnic (
     Device@sysa = { le0 = "166.98.16.103", qfe3 = "166.98.16.103" }
     Device@sysb = { le0 = "166.98.16.104", qfe3 = "166.98.16.104" }
     NetMask = "255.255.255.0"
     )
```

In this resource definition, the values for the 'Device' attribute are localized to a particular system (sysa and sysb respectively). The 'NetMask' attribute however, is a global attribute, and its value is constant between all of the systems.

Attribute Data Types and Dimensions

As the type of data that is desired to be contained within a attribute value may vary greatly. To accommodate this, VCS allows the use of numerous different attribute data types and dimensions. Data types can be defined as a string, an integer, or a Boolean value. Attribute dimensions permitted include scalar, vector, keylist, and association. Please see the VCS Users Guide for a complete description of these data types and dimensions in the "Attributes" section.

ATTRIBUTE PASSING

The VCS engine passes the values for all attributes specified by a particular resource definition to the respective agent for that resource. When script based agents are used, the agent typically passes an identical set of attributes to the most entry points when they are called by the agent. This typically consists of the resource name, followed by the contents of the ArgList, although there are some exceptions that will be mentioned later. Each entry point receives the same variables set in the same order. The entry point can then utilize the variables as necessary. In the following series of examples, the following resource definition file would be used:

```
type Apache (
        str ServerRoot
        str ApacheDaemon
        str IPAddr
        int Port
        str TestFile
        int DebugLevel
        static str ArgList[] = { ServerRoot, ApacheDaemon, IPAddr, Port, TestFile, DebugLevel
}
)
```

Online, Offline, and Monitor Entry Points

The online, offline and monitor entry points will each receive the resource name and the contents of the ArgList when called. Each entry point may or may not need all the variables provided, but each will receive all of the variables in the same order regardless.

The following Perl code example represents the beginning of a script based online, offline or monitor entry point where variables are defined and assigned the values of the passed arguments. It is important to understand that the values are passed in a specific order, and not by name. So the order used to assign values to the variables in the various entry point scripts must be the same.

```
my ($ResName, $ServerRoot, $ApacheDaemon, $IPAddr, $Port, $TestFile, debugLevel) = @ARGV;
```

To use a shell script as an example instead of Perl, the assignment of variables would look similar to the example provided below.

```
ResName=$1
ServerRoot=$2
ApacheDaemon=$3
IPAddr=$4
Port=$5
TestFile=$6
DebugLevel=$7
```

Note the fact that ResName is referred to as $1, as $0 represents the name of the script executed

Clean Entry Point

The clean entry point is the only exception to the argument passing rule, as it is passed a “clean reason” as the second variable, right after the resource name and before the contents of the ArgList. This argument consists of an integer within the range of 0 to 5. Each value has a specific meaning, as described below:

Integer	Meaning
0	The offline entry point did not complete within the expected time
1	The offline entry point was ineffective.
2	The online entry point did not complete within the expected time.
3	The online entry point was ineffective
4	The resource was taken offline unexpectedly
5	The monitor entry point consistently failed to complete within the expected time

Refer to the following code example for a demonstration of how this affects the assignment of passed variable within a clean entry point.

```
my ($ResName, $CleanReason, $ServerRoot, $ApacheDaemon, $IPAddr, $Port, $TestFile,
DebugLevel) = @ARGV;
```

Notice the addition of the assignment of a value to the $CleanReason variable. This value is not commonly used in most agents. It does allow the agent developer to utilize the integer to perform different functions based on the reason why the clean entry point is being called.

SCRIPT BASED AGENT RETURN VALUES

When utilizing script based entry points, the exit values used are not arbitrary values. Depending on the entry point, they provide a very specific function as described in the following section.

Online and Offline

The online and offline scripts each accept the same ArgList from the ScriptAgent and carry out the proper tasks to online or offline the resource. When the online or offline entry point completes, the monitor entry point will be called to verify the resource state change has occurred.

Some applications may take a period of time between when the online or offline procedures complete and the application is able to be monitored. To facilitate this, the exit code for the online and offline scripts define a number of seconds to wait after the completion of the entry point before running the monitor entry point.

This is somewhat different than standard script programming, where an exit code of 0 typically represents success and exit code of 1 indicates a failure of the script.

A key point is here is that the monitor entry point is what determines if a resource is offline or online, not any of the other entry points. When developing the online and offline entry points, you may provide testing functions for debug purposes, but remember that the agent will invoke the monitor entry point to actually validate the resource has gone online or offline.

In the example below, the online script for a very basic Apache agent is shown. It attempts to start the Apache server daemon. If the start passes, the script exits with return code of 5. This will cause the monitor entry point to be called in five seconds after the exit of the script. If the start fails, the script exits with exit code of 0 which specifies that the monitor entry point should be called immediately after the completion of the script.

```
$ApacheDeamon = “/opt/site/bin/httpd";
$Command      = $ApacheDeamon." -d $ServerRoot";

$status = system($Command);

if ($status == 0) {
  # Command executed without problems
  exit 5;

} else {
  # Command executed with problems
  exit 0;
}
```

The wait incorporated when the online procedure is successful allows the Apache web server plenty of time to come fully online before monitoring of the resource occurs. When the online procedure is not successful, the monitor is called as quickly as possible to initiate any further action that may be required such as calling the clean entry point, or restarting the resource.

In either case, the ApacheAgent will run the monitor to determine if the resource is online or not. The only difference is the length of delay before it is called.

Monitor

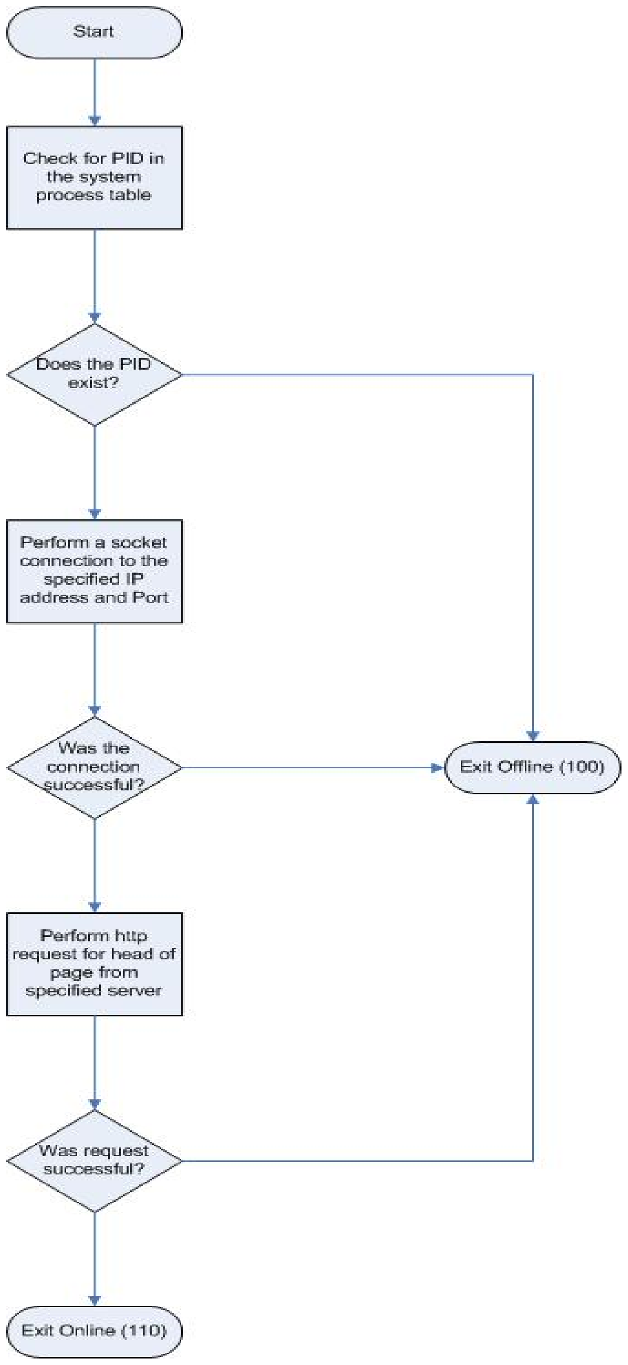
Monitor entry points can return a host of different values. Any values returned that are less than 100 are used to indicate a script or configuration error. For instance, if a required attribute was not provided with a value, the monitor entry point could exit with a value of 99. This would signify that the resource was unable to be monitored. The use of logging messages before the entry point exit would be used to assist the user in identifying and correcting the problem. The return value of 100 specifies that the resource is in an offline state. Values returned between the ranges of 101 to 110 represent that the resource is in an online state.

The reason for the wide range of possible return codes is because these values are associated with the confidence level associated with the resource. The last two positions represent the confidence level of the monitor. This is representative of a percentage. Given this, a return value of 101 represents an online resource with a 10% confidence level. A value of 102 represents an online resource with a 20% confidence level. This continues in sequence until the exit value of 110. The exit value of 110 represents an online resource with a 100% confidence level.

Further information on the ConfidenceLevel attribute and how it can be utilized by an agent developer is covered under the Advanced Topics section in Appendix A of this paper. For now, the only monitor entry point exit values will be 99 (representing an unknown state), 100 (representing a offline state), and 110 (representing a online state).

From a high level, it is best to construct a monitor script to test at the most basic level first, followed by increasing depth monitor. In this fashion, if a resource is offline completely, it will be caught early with a minimal of system resource utilization.

For example, the Apache monitor script that will be used later in this document as an example will use varying levels of monitoring to determine the health of an Apache server. The overall program flow that will be used is as follows:


Start
Check for PID in the system process table
Does the PID exist?
Perform a socket connection to the specified IP address and Port
Was the connection successful?
Exit Offline (100)
Perform http request for head of page from specified server
Was request successful?
Exit Online (110)

If a procedure is not successful, the script will exit immediately with an offline value and will not continue on to the next level of monitoring.

Clean

The clean entry point is called under a number of different circumstances as described earlier. The exit codes for the clean entry point are either 0 (successful clean operation) or 1 (unsuccessful clean operation).

AGENT DEVELOPMENT CONCEPTS

This section focuses on concepts that are used for the actual construction of a VCS agent.

USING THE SCRIPTAGENT BINARY

In order to simplify agent development using scripts, VERITAS ships the ScriptAgent binary with VCS. The ScriptAgent binary has all the necessary agent framework capability needed compiled into it. The ScriptAgent is designed to look for scripts named offline, online, monitor and clean which represent the respective entry points in the directory where it was started.

To use the ScriptAgent, create a new directory under $VCS_HOME/bin directory named the same as the agent name. On a UNIX system, $VCS_HOME is typically set to /opt/VRTSvcs for example. Assuming that an agent for an Apache web server is desired to be created, a new directory called /opt/VRTSvcs/bin/Apache should be created. The ScriptAgent binary (/opt/VRTSvcs/bin/ScriptAgent) can then be copied or symbolically linked into this directory and called ApacheAgent. Naming conventions will be covered in detail later in this paper.

The use of symbolic links to the ScriptAgent binary is highly recommended. In the event of a VCS upgrade or update it may be possible that the ScriptAgent binary is replaced. By using symbolic links the agents will automatically use the new binary without further effort on the part of the administrator.

At this point a fully functional ApacheAgent is available for use. It is completely capable of connecting to the VCS engine to receive commands, and being controlled with all the standard attributes. The ApacheAgent will look within the /opt/VRTSvcs/bin/Apache directory for scripts called online, offline, monitor and clean which represent the various agent entry points.

When VCS starts, the ScriptAgent (now called the ApacheAgent) starts and registers with the cluster engine. It receives a complete list of all resources that will be under its control. Known as the Agent startup function, this is completely transparent to the script developer and just described here for clarity.

MESSAGE LOGGING

The messages that an agent will display provide the first indication that a problem may exist. As a result, the messages that an agent generates become very important in providing support for agents. Most customers also see proper logging as a key quality determination factor.

VERITAS expects all agents developed for use with VCS to provide complete logging of all error conditions and resource state changes. Further, the agent will not generate unnecessary logging, such as continuous messages that a resource is offline, when in fact it is supposed to be offline.

The following section will document how to use the VCS logging facility, how to enable internationalization and how to enable various levels of logging.

Message Format

In VCS 4.0, the format for all messages is:

<Timestamp><Mnemonic><Severity><UMI><MessageText>

Where Timestamp specifies the time of the log message generation, Mnemonic represents the product being used ('VCS' in all capital letters for all agents), Severity indicating the severity of the message, a Unique Message ID (UMI), and the text of the message.

An example of a log message with a WARNING severity level may resemble the following:

```
2004/11/03 22:19:11 VCS WARNING V-16-10001-5526 (vscpdb06)
Mount:mnt_prbma_control03:offline:attempting Unmount of mountpoint:/oradata/prbma/control03,
blockdevice:/dev/vx/dsk/prbmadg/control03, type:vxfs
```

Unique Message IDs (UMIs)

A UMI provides multiple amounts of information to the user. The format for a UMI is as follows:

V-<*Originator ID*>-<*Category ID*>-<*Message ID*>

The Originator ID is a decimal number that is assigned by VERITAS to an agent developer.

The Category ID is a binary number in the range of 0 to 65536. This ID number is also assigned by VERITAS to an agent developer. When a debug message is used, the category ID defaults to the value of '50' and need not be defined by the logging functions.

Message ID numbers range in value from 0 to 65536 for a given category. The Message ID is assigned by the agent developer and is used to uniquely identify every non-debug (re: normal) log message generated by the agent. Within debug messages the Message ID field is not used. As a result, it need not be defined for debug messages.

Message Text

The text of a log message consists of a formatted message string preceded by a dynamically generated header. The header consists of three colon separated fields; the name of the agent, the resource name, and the name of the entry point.

Previously, an example of a log message was provided. Review this log message example again:

```
2004/11/03 22:19:11 VCS WARNING V-16-10001-5526 (vscpdb06)
Mount:mnt_prbma_control03:offline:attempting Unmount of mountpoint:/oradata/prbma/control03,
blockdevice:/dev/vx/dsk/prbmadg/control03, type:vxfs
```

In this example, the message header consists of:

```
Mount:mnt_prbma_control03:offline:
```

In this example, 'Mount' is the name of the agent type that generated the message. The next field is the name of the resource (mnt_prba_control03) which is followed by the last field that specifies the entry point that was used (offline). Following the header is the message text to be logged.

When specifying the text of a message, adhere to the following requirements:

- Begin messages with a capitalized first letter.
- Do not end messages with a period.
- Do not end messages with a new line.
- Do not use the "\" line continuation character.

Severity Levels

Normal messages may be assigned a severity level. The use of severity levels with log messages facilitates the filtering of messages and notification of problems to the administrator. The severity levels that can be used with normal messages are described in the following table:

Severity Level	Description
Critical	Represents a failure of a script function
Error	Typically used to denote a negative condition within a function
Warning	Informational messages that are used to signify a potential issue
Notice	Messages that signify that a event (such as a resource state change) has occurred
Information	Messages that are purely informational in nature.

Logging Functions

VCS 4.0 signified a change in the manner by which a agent developer would log messages. Prior to VCS 4.0, the halog command was used almost exclusively for all agent logging functions. With the release of VCS 4.0, a variety of functions or *wrappers* are provided to ease the development of log messages for script based agents. The halog command can still be utilized to create log messages, however as the provided wrappers are easier to use and less error prone, only the use of the wrappers will be discussed in this paper.

To utilize the logging functions, a line in the entry points must be given that specifies the location of the file defining the logging functions. This line must be provided prior to the invocation of any logging functions. With shell based entry points, this line would resemble:

```
. ${VCS_HOME:-/opt/VRTSvcs}/bin/ag_i18n_inc.sh
```

And for Perl based entry points, the line would resemble:

```
use ag_i18n_inc;
```

Once this line has been defined, the various logging functions can be used.

The VCSAG_SET_ENVS Function

This function is used within every script based agent entry point. Its primary function is to set environmental variables required for the other functions. Lists of the environmental variables that are set include:

Environmental Variable	Description
VCSAG_LOG_CATEGORY	Sets the category ID
VCS_LOG_AGENT_NAME	Sets the absolute path to the agent.
VCSAG_LOG_SCRIPT_NAME	Sets the absolute path to the entry point script
VCSAG_LOG_RESOURCE_NAME	Sets the name of the resource

This function must be called before any other logging function is used. The method by which the function is called varies only slightly depending on what scripting language the entry point is programmed in. If the entry point is programmed in Shell, an example of its use would be:

```
VCSAG_SET_ENVS ${resource_name} ${script_name} ${category_id}
```

Whereas the same invocation from within a Perl based entry point would resemble:

```
VCSAG_SET_ENVS ($resource_name,$script_name,$category_id);
```

The VCSAG_LOG_MSG Function

This function is used to log normal messages. At a minimum, this function must include the severity level of the message enclosed in quotes, the text of the log message enclosed within quotes, and an integer representing the message ID.

The severity levels are specified through the use of a letter. The following table shows the letter that represents each of the severity levels:

Identifying Letter	Severity Level
C	Critical
E	Error
W	Warning
N	Notice
I	Information

When used within a shell based entry point, an example of this function being used would be as follows:

```
VCSAG_LOG_MSG "C" "Two files found" 140
```

And the same usage as shown above with a Perl based entry point would look like:

```
VCSAG_LOG_MSG ("C", "Two files found", 140);
```

In both of these examples, a message of a Critical severity level consisting of the text “Two Files Found” and assigned the message ID of 140 would be logged.

The VCSAG_LOGDBG_MSG Function

This function is used to generate log messages that are specifically designed to only be used for debugging or troubleshooting. These messages must be specifically enabled to be logged by the user, typically at the request of customer support personnel, by the use of the halog –addtag command. Unless enabled, these messages are not outputted to the log files.

At a minimum, they consist of the severity level of the message followed by the text of the message. A shell based example would look like:

```
VCSAG_LOGDBG_MSG "1" "Two files found"
```

While a Perl example would look like:

```
VCSAG_LOGDBG_MSG ("1", "Two files found");
```

As is evident in the examples provided above, the severity level is different for debug messages than it is for normal log messages. With the debug messages, the severity level is represented by an integer with a value in the rage of 1 to 21.

Perl Script Messaging Example

The following example demonstrates the use of the logging functions from within a Perl based entry point:

```
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
       if 0;
```

```
($res_name) = @ARGV;

use ag_i18n_inc;

VCSAG_SET_ENVS ($res_name,online,10061);

# Perform online function here

VCSAG_LOG_MSG ("N", "online completed", 1);

exit 0;
```

Shell Script Messaging Example

The following example demonstrates the use of the logging functions from within a shell based entry point:

```
!#/bin/ksh

ResName=$1

VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"

. $VCSHOME/bin/ag_i18n_inc.sh

VCSAG_SET_ENVS $ResName 10061

# Perform online function here

VCSAG_LOG_MSG "N" "online completed" 1

exit 0
```

MESSAGE CATALOGS

Message catalogs start with the creation of a Source Message Catalog (SMC). SMC files are standard text files that follow a very simple format:

```
#!language = language_ID
#!module = module_name
#!version = version
#!category = category_ID

# Start message list
message_ID1 {%s:msg}
message_ID2 {%s:msg}
message_ID3 {%s:msg}
```

The top portion of the SMC file defines a header of information. The header information required is described in more detail below.

Header Information	Description of
language	Defines the language id of the file. An example of this would be 'en' which stands for English
module	The module name. This is set to 'HAD' for all agents
version	The version of the module. This should be set to '4.0' for all VCS 4.0 agents
category	The category ID as assigned by VERITAS for the agent

An example SMC file would resemble the following:

```
#!language = en
#!module = HAD
#!version = 4.0
#!category = 203

100 {"%s:Invalid message for agent"}
101 {"%s:Process has been restarted"}
102 {"%s:Error opening /proc directory"}
103 {"%s:No start program defined"}
```

In the presented example, the messages that can be outputted by the agent are listed in ascending order starting at message ID 100. The %s character sequence at the beginning of each message specified the three part message header. This message header is automatically inserted by the agent framework.

Once a SMC file has been created, it can be converted to a Binary Message Catalog (BMC) through the use of the bmcgen command. The resulting BMC file is then stored under %VCS_HOME%/messages/<language code>.

Take for example a BMC file that was created with support for Japanese. It would be stored in %VCS_HOME%/messages/ja.

Providing support for multiple languages can be a complex task. Given the low frequency of use, and the complexity involved with providing support for multiple languages, further discussion of this will not be presented in this paper. The information is presented here simply to inform the agent developer that this feature is available to them if desired. For more information, please refer to the VCS Agent Developer's Guide.

THE AGENT INFORMATION FILE

As previously discussed, an Agent Information File (AIF) is a XML formatted file. The AIF consists of two functional parts; the header information and the attribute argument details.

The header information part precedes the attribute argument details and consists of several lines of information. Header tags that must be specified are listed in the following table:

Header Tag	Description
name	Specifies the name of the agent
version	Specifies the version of the agent
agent_description	Describes the agent
platform	Defines the platform that the agent is created for
vendor	Specifies the agent vendor
info_implemented	Indicates if the info entry point is implemented or not. Can be assigned a value of 'Yes' or 'No'
agenttype	Specifies the type of the agent. May be either Binary, Script, or Mixed. Script based agents should have this set to 'Script'
minvcsversion	Defines the minimal VCS version required to support the agent

The header information is followed immediately by the attribute argument details section. Each attribute that is used by the agent is defined in this section along with any specific values. The following table describes the possible attributes that can be defined for each attribute.

Argument	Description

type	Possible values for the attribute type, such as “str” for strings
dimension	Possible values for the attribute dimension, such as “Scalar"
editable	Indicates if the attribute is editable or not. In most cases, the resource attributes are editable. Possible Values = “True” or “False”
important	Indicates whether or not the attribute is important enough to display. In most cases, the value is True. Possible Values = “True” or “False”
mustconfigure	Indicates if the attribute must be configured to make the resource online. The GUI displays such attributes with special indication. If the value of the `mustconfigure` attribute is not specified, the resource state should become “UNKNOWN” in the first monitor cycle. Example of such attributes include `Address` for the IP agent, `Device` for the NIC agent, and `FsckOpt` for the Mount agent). Possible Values = “True” or “False”
unique	Indicates if the attribute value must be unique in the VCS configuration, that is, whether or not two resources of same resource type may have the same value for this attribute. Example of such an attribute is `Address` for the IP agent. Possible Values = “True” or “False”
persistent	This argument should always be set to “True”; it is reserved for future use. Possible Values = “True”.
range	Defines the acceptable range of the attribute value. The VCS GUI can use this value for attribute value validation. Value Format: The range is specified in the form {a,b} or [a,b]. Square brackets indicate that the adjacent value is included in the range. The curly brackets indicate that the adjacent value is not included in the range. For example, {a,b] indicates that the range is from a to b, contains b, and excludes a. In cases where the range is greater than “a” and does not have an upper limit, it can be represented as {a,] and, similarly, as {,b] when there is no minimum value.
default	Defines a default value for the attribute if desired.
displayname	Defines a user friendly name to be displayed by the GUI for the attribute. For example, if the attribute was FsckOpt, this value could be set to the more user friendly value of ‘fsck option’.

Below is an example of a AIF for a FileOnOff agent. This particular agent only has one attribute defined as ‘PathName’.

```
<?xml version="1.0">
<agent name="FileOnOff" version="4.0">
    <agent_description>Creates, removes, and monitors files.
    </agent_description>
    <platform>Solaris</platform>
    <agenttype>Binary</agenttype>
    <info_implemented>No</info_implemented>
    <minvcsversion>4.0</minvcsversion>
    <vendor>VERITAS</vendor>

    <attributes>
        <PathName type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="True"
        persistent="True" range="" default=""
        displayname="PathName">
        <attr_description>Specifies the absolute pathname.
        </attr_description>
        </PathName>
    </attributes>

    <agentfiles>
        <file name="$VCS_HOME/bin/FileOnOff/FileOnOffAgent" />
    </agentfiles>
</agent>
```

MULTILEVEL MONITORING

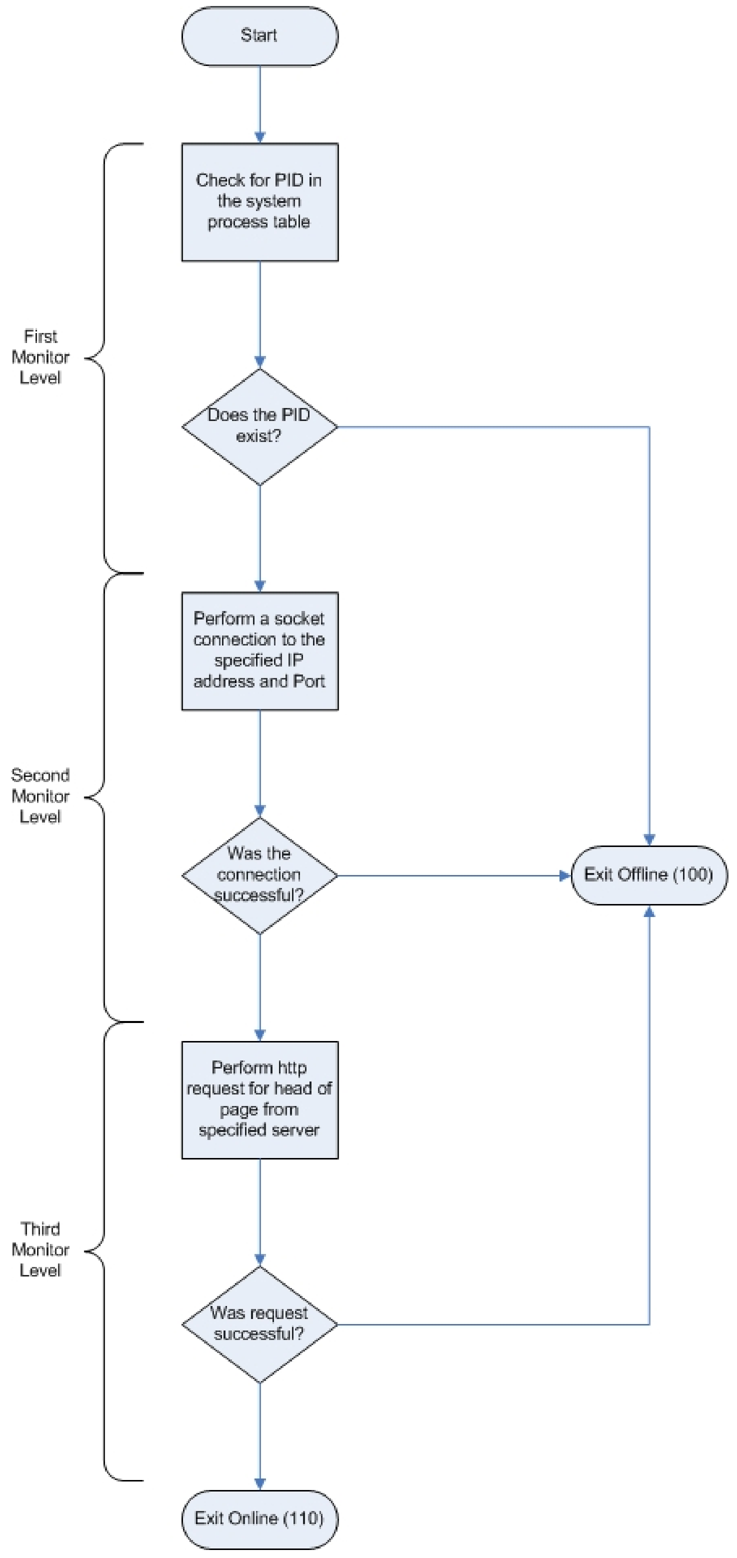
An agent must be able to monitor a resource even when the resource and its service group are offline. This means the agent developer must use care to ensure all operations carried out by the monitor are available when the group is offline.

For example, consider an application that is installed on shared storage that fails between a series of servers. If an agent was developed to monitor this application through the use of application binaries located on the shared storage the monitoring may be problematic. This is due to the fact that the application can not be properly monitored if the shared storage is only available to another host. For this reason, it is better to use operating system capabilities to monitor applications when possible.

This said however, often the best method to monitor an application requires the use of an application binary or access to the application. Take an Apache web server for an example. The best manner to monitor a web site served by an Apache instance is to connect to the server and attempt to retrieve a valid page from the server. However if the web server is not online on the local host when it was specified to be monitored, and this was the only method of monitoring, the monitor would hang awaiting a socket timeout.

To prevent this from occurring, an agent must be developed to perform a series of tests in a successive manner so that the status can be returned as quickly and with as few system resources used as possible. Each of these tests is referred to as a *level* of monitoring.

The following flowchart demonstrates the use of multiple levels of monitoring. Notice that there are three levels of monitoring, with each successive monitoring level building on the functionality of the previous level.



Start
First Monitor Level
Check for PID in the system process table
Does the PID exist?
Second Monitor Level
Perform a socket connection to the specified IP address and Port
Was the connection successful?
Exit Offline (100)
Third Monitor Level
Perform http request for head of page from specified server
Was request successful?
Exit Online (110)

ERROR HANDLING

Dealing with error conditions is a standard practice when creating a program in any programming language. As the goal for any agent developer should be to provide the ability for the developed agent to be used on as many systems as possible, this increases the amount of risk associated with using the agent. System configurations, untrained personnel, and improperly configured applications can all generate error conditions that may not exist on the systems used to develop the agent.

Although it is practically impossible to predict all the possible errors that could possibly be generated, there are general steps that the agent developer can take to minimize the possible effects. This section will describe some of those procedures.

Error Checking

Error checking routines should be utilized to ensure that the agent can perform its task properly in the user's environment, and if not, to provide adequate diagnostic messages back to VCS. Examples of items that should be checked include:

- Ensure proper script interpreter is installed on a system. If coding in Perl, verify VCS Perl is installed in the expected location.

- The existence of non-standard commands

Commands that are not considered to be a standard component of the operating system should be checked to ensure that they exist on the system prior to executing the command. This also includes commands external to the script that are assumed to be available.

- The number/value of arguments passed to the agent.

An agent may require a certain number of arguments to be passed to it to allow it to complete its intended function. The agent may also want to verify that the number of values passed to the agent is acceptable.

- The ability of the agent to execute a particular function.

Although it is a general practice not to use commands that may 'hang', there are some situations where these commands are required. To prevent the agent from hanging as a result of executing the command, it may be possible to perform a check to ensure that the command can be properly executed.

An example of this is when the agent includes the capabilities of performing multiple levels of monitoring. The additional monitoring level may be optional, and require a value to be passed to the agent to activate the check. As part of the error checking, the existence of this optional value can be checked, and if present, a flag set to identify that the optional monitor check is to be performed.

- The validity of the passed values.

The values that are passed to an agent and likewise the agent's entry points are normally derived from user input. Typographical and such errors are common errors that must be dealt with. An agent developer may not be able to predict all possible permutations of user errors, however the developer should take steps to ensure that the data presented is valid.

Take for example an agent that expects to have a IP address passed to it as a attribute value. The developer can verify that it conforms to the dotted quad standard of IP address notation before continuing with the agent operations.

The following example demonstrates some of the types of error checking:

```
# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
    if (-x $ApacheDeamon) {
        VCSAG_LOGDBG_MSG (1, "$ApacheDeamon is executable");
    } else {
        VCSAG_LOG_MSG ("C", "$ApacheDeamon is not executable", ($msg_id+1));
        VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+2));
        exit 0;
    }
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+3));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 0;
}
```

REQUIRED DESIGN CRITERIA

When developing agents for use with VCS, the following minimal criteria must be met by the agent developer:

Offline Monitor Requirements

The agent must be capable of monitoring if a resource is online or offline at all times, even if the service group for this resource is offline. This means the agent must be capable of running when the underlying resources for that resource are no available, such as file systems.

Clean Capability

All agents for on-only and on-off resources will provide a clean capability. Clean is required for proper resource restart and failover.

Cluster Configuration Changes Prohibited

At no time is it permissible for an agent to make calls to open the VCS cluster configuration and make changes. For example, the agent changing the status of a status attribute in the main.cf. Opening the configuration from inside an agent leaves the cluster potentially vulnerable if a crash occurs while the configuration is open. A complete cluster crash with the configuration open causes a stale configuration and prevents the cluster from starting. VCS 4.0 provides the concept of “temporary attributes” which may be set without opening the configuration file.

Debug Messages

To assist in agent troubleshooting, it is often desirable to have a method that will allow the logging of additional informational messages. These messages, referred to as “Debugging Messages”, should not occur during the normal operation of the agent. The developer must provide a method to enable and disable debug messages within the agent.

Agent Permissions

The executable components of an agent need to be set properly so as not to induce additional security risks to the system.

For all Unix-based platforms, executable components of custom agents should be given the permission mode of 744. For Windows-based platforms, the executable components of the agent should have the Local Administrators given Full Control, and Everyone Read and Read and Execute permissions.

Agent Packaging

To provide for ease of installation and it is recommended that the agent be packaged in a format suitable for use with the operating system used.

Each operating system has a preferred packaging method. The following table lists these preferred packaging methods

Operating System	Packaging Method
Solaris	Pkg, install with pkgadd
HP-UX	Shar, install with swinstall
AIX	Bff, install with installp
Linux	Rpm, install with rpm
Windows	InstallShield for Microsoft Windows Installer v. 2.01 or later

SECOND GENERATION AGENT EXAMPLE

The following section is provided as a walkthrough of the development process of a VCS agent, using the concepts previously outlined in this paper. Upon the completion of this section, a completely functional agent is the result. More importantly, the examples presented here will demonstrate and reinforce previously discussed concepts.

DEFINING FUNCTIONAL REQUIREMENTS

For this example, we will be constructing a VCS agent that will control an Apache web server. Before a single line of code is written, it is important to define the function requirements of the agent to be developed. By performing a little bit of prior planning, the development of agent will be significantly easier.

At a high level, we want the agent to support an Apache web server that is version 1.3. The agent should run in a Solaris environment. It should support multiple instances of Apache running on the same physical host.

To accomplish these high level requirements, it is necessary to implement four entry points. Each of which will have a specific function that when combined will allow us to manage the Apache web server. The entry points to be used and the desired functions of each are described below:

Online

The online entry point will be responsible for bringing an instance of an Apache web server online on the local host. Accomplishment of this will require the calling the Apache daemon with a –d option and then specifying the server root location. Very simply, the following code can be used to online an Apache web server instance:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my($cmd_output);

# Execute the command
$cmd_output = `/usr/local/apache/bin/httpd –d /webdocs`;

exit 0;
```

Offline

The function of the offline entry point will be to bring the Apache web server instance down, or offline, on the local host. This is done by obtaining the PID from the PID file created when the Apache web server is brought online.

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
$kill = “/usr/bin/kill”;
my ($cmd_output, $pid);

# Obtain the PID from the PID file
open(PIDFILE,”/usr/local/apache/logs/httpd.pid”);
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Kill PID
```

```
$cmd_output = `$kill –TERM $pid`;

exit 0;
```

Notice that the system commands, such as ‘kill’, have been defined with the full pathname. This is in line with the best practices that are discussed later in this document.

Monitor

The entry point needs to validate the state that the Apache web server is currently in. This means that the entry point needs to be able to determine if the web server is in an online or offline state. We can do this by again retrieving the PID from the PID file that Apache creates when it brings an instance online. Once the PID has been obtained, we can then verify that the PID specified exists in the system process table.

The basic code for doing this would resemble:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my ($pid);

# Obtain the PID from the PID file
open(PIDFILE,”/usr/local/apache/logs/httpd.pid”);
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Check to see if there is a entry under /proc for $pid
if (-r “/proc/$pid”) {
    # PID exists, exit with a ONLINE status
    exit 110;
}

# Return with a OFFLINE status
exit 100;
```

Clean

The clean entry point is designed to ensure that the resource is offline and that it is able to be restarted. As mentioned previously, it may be called for a variety of reasons. For example, if the resource goes offline unexpectedly, it fails to go online, or it fails to go offline the clean entry point will be called.

With Apache, we will want the clean entry point to kill a specific web server instance, as well as any child processes that may have been spawned by that process. A basic script to do this follows:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
$ps = “/usr/bin/ps”;
$kill = “/usr/bin/kill”;
$grep = “/usr/bin/grep”;

my ($cmd_output, $pid, $args);

#  Obtain data from the process table
```

```
$data=`$ps –ef –o pid –o args | $grep \"/usr/local/apache/bin/httpd –d /webdocs\"`;
@data = split(/\n/,$data);

#  Go through collected data and look for any PIDs to kill
foreach $line (@data) {
    ($pid, $args) = split (" ",$line);
    if ($line eq "/usr/local/apache/bin/httpd –d /webdocs") {
        $cmd_output = `$kill -9 $pid`;
    }
}

exit 0;
```

DETERMINING ATTRIBUTES NEEDED

In the examples given above for the entry points required, the code is not very flexible. As you can tell, the examples were all hard coded to only be able start up an instance of Apache in one location. This does not coincide with the functional requirements that were defined.

In order to make the agent code more flexible, attributes need to be used. As attributes can be dynamically assigned values by the user, this can enable a high degree of flexibility.

The attributes that an agent can use are defined within the resource type definition file for that agent. However, before we can start creating the resource type definition, we need to reexamine the code that we have and identify what attributes are needed.

Online

First, let us reexamine the online entry point code:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my($cmd_output);

# Execute the command
$cmd_output = `/usr/local/apache/bin/httpd –d /webdocs`;

exit 0;
```

To allow for additional flexibility here, we may want to be able to dynamically define the location of the httpd daemon process. Currently this is hard coded to '/usr/local/apache/bin/httpd'. Assigning an attribute to define this value will allow the agent to support many different methods of installing the Apache web server.

Again, we will define the attribute within the resource type definition file, and the value for the attribute in the main VCS configuration file. The VCS engine will then pass this value to the agent, which in turn, passes it to the entry point. Once we have determined that we are going to use an attribute for a particular purpose, we need to add a method to capture the value for that attribute. The code will also need to be modified to take into account the new values.

As an example, to reconfigure the online entry point to use an attribute named ApacheDeamon the following changes would be required:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
```

```
my($cmd_output);

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon) = @ARGV;

# Execute the command
$cmd_output = `$ApacheDeamon –d /webdocs`;

exit 0;
```

Notice that the line that captures the passed values contains a variable named ‘resource_name’. This is because the VCS engine will automatically insert the name of the resource as the first argument to all of the entry points.

Further attributes could also be used. As an example, an attribute that defines the Server Root could also be defined. This can allow a system administrator to install one copy of the Apache binaries on the local host at one location, but then use a different location for the specific web server instances. For example:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my($cmd_output);

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot) = @ARGV;

# Execute the command
$cmd_output = `$ApacheDeamon –d $ServerRoot`;

exit 0;
```

As you can see, this vastly increases the flexibility of the online entry point.

Offline

Assuming that we incorporate the previously defined attributes into the offline entry point script, the offline entry point would now resemble the following:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
$kill = “/usr/bin/kill”;
my ($cmd_output, $pid, $pid_file);

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot) = @ARGV;

$pid_file = $ServerRoot.”/logs/httpd.pid”;

# Obtain the PID from the PID file
open(PIDFILE,”$pid_file”);
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Kill PID
$cmd_output = `$kill –TERM $pid`;
```

```
exit 0;
```

At this point in time, there are no additional attributes that can be used in the offline entry point.

Monitor

Incorporating the attributes used with the monitor entry point results in the following:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my ($pid);

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot) = @ARGV;

$pid_file = $ServerRoot.”/logs/httpd.pid”;

# Obtain the PID from the PID file
open(PIDFILE,”$pid_file”);
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Check to see if there is a entry under /proc for $pid
if (-r “/proc/$pid”) {
    # PID exists, exit with a ONLINE status
    exit 110;
}

# Return with a OFFLINE status
exit 100;
```

Again, at this time there are no additional attributes that can be defined.

Clean

The modified clean entry point would resemble:

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
$ps = “/usr/bin/ps”;
$kill = “/usr/bin/kill”;
$grep = “/usr/bin/grep”;

my ($cmd_output, $pid, $args);

#  Obtain values passed to the agent
my ($resource_name, $clean_reason, $ApacheDeamon, $ServerRoot) = @ARGV;

#  Obtain data from the process table
$data=`$ps –ef –o pid –o args | $grep \”$ApacheDeamon –d $ServerRoot\”`;
@data = split(/\n/,$data);

#  Go through collected data and look for any PIDs to kill
foreach $line (@data) {
    ($pid, $args) = split (“ “,$line);
    if ($line eq “$ApacheDeamon –d $ServerRoot”) {
        $cmd_output = `$kill -9 $pid`;
    }
}

exit 0;
```

Although there are no additional attributes that are needed with the clean entry point, there is a change that needs to be noted. With the clean entry point the VCS engine will automatically add an integer that represents why the clean entry point is being called. We do not need to use this value now, but as it shifts the arguments over by one, a variable named ‘clean_reason’ is defined as a place holder.

CREATING THE TYPE DEFINITION FILE

After the reexamination of the code for attribute possibilities, only two attributes were found to be needed at this time. These are the attributes ApacheDeamon and ServerRoot. Armed with this information, it is now possible to create a resource type definition file for the agent.

```
type Apache (
        str ServerRoot
        str ApacheDeamon
        static str ArgList[] = { ApacheDaemon, ServerRoot }
      )
```

The type definition file will define the name of the agent, the attributes that can be used, and the type and dimension of the values that may be assigned to those attributes. In this example, the agent that we are creating is called ‘Apache’. This agent will use two attributes that are both capable of holding string values.

Of great importance is the ArgList line. This line defines the order which the values of the attributes are passed to the agent. This must match what we have used in the entry points, otherwise unpredictable and often undesirable results will occur.

INCREASING THE AGENT FUNCTIONALITY

At this point, we have the basic functionality of an agent. But the agent can still be improved upon. Most importantly, the monitor entry point needs to be enhanced.

The current method of monitoring the Apache web server simply looks for the Apache PID file, examines the contents, and ensures that the PID exists under /proc. This is prone to error. Take for example if the web server crashed, but the same PID was reused by the system for another process. Continuing to use solely this monitor method would result in a false positive result being returned to the VCS engine. In a highly available environment, this is unacceptable.

To prevent against this from happening, additional checking should occur to ensure that the PID not only exists, but that it also matches the specified command line.

```
#!/opt/VRTSvcs/bin/perl5

# Initialize variables used
my ($pid, $output, $startup_command);

# Initialize variables used
$ps = “/usr/bin/ps”;
$grep = “/usr/bin/grep”;
```

```
#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot) = @ARGV;

$pid_file = $ServerRoot."/logs/httpd.pid";
$startup_command = "$ApacheDeamon –d $ServerRoot";

# Obtain the PID from the PID file
open(PIDFILE,"$pid_file");
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Check to see if there is a entry under /proc for $pid
if (-r "/proc/$pid") {

    # PID exists under /proc
    # Now verify that the PID has not been reused by the system
    # This is done by verifying that the PID matches the startup
    # command used to online the resource

    $output = `$ps –p $pid –o args | $grep –v COMMAND`;

    if ($output !~ $startup_command {
        # The PID does not match the startup command
        # exit offline
        exit 100;
    }

    # PID exists and the arguments match the startup command
    # Exit with a ONLINE status
    exit 110;
}

# Return with a OFFLINE status
exit 100;
```

Furthermore, additional levels of monitoring can be added to increase the accuracy of the monitor reporting. As an example, assume that the PID was found as shown above. This will not however take into account if the web server is actually able to serve web pages, and is not in a 'hung' state.

To check for this, we can do two things. First we can attempt to open a socket connection to the web server. This will tell us if the server is listening on the correct port. Second, we can attempt to obtain a page from the server to ensure that the web server is actually responding.

Failure of being able to establish a socket connection would preclude the need to attempt to obtain a page, as if no connection is possible to the server, no pages can be obtained. Likewise, if the PID doesn't exist, then there is no need to continue to attempt to perform the socket connection.

This results in three levels of monitoring. The first level is the monitoring for the PID of the web server. The second is the attempt to establish a socket connection with the web server on the correct port. And the third is to attempt to retrieve a web page.

The second and third levels of monitoring are often referred to as 'deep level' monitoring. Not all users of the agent may wish to perform these additional levels of monitoring. For this reason, we want to provide the user with the ability to choose if these additional levels of monitoring should be used or not. We can accomplish this through the use of an attribute whose only function is to specify if the monitor entry point is to continue with the second and third levels of monitoring upon a successful monitor at the first level. We could do this by simply checking to see if all the required attributes needed to perform the deep level monitoring, such as the IP address and port to connect to, were specified. But this does not allow the user to selectively turn on and turn off the deep level monitoring without performing additional steps.

The following code demonstrates the implementation of the additional checks as described above.

```
#!/opt/VRTSvcs/bin/perl5

# Specify Perl Modules to use
use IO::Socket;

# Initialize variables used
my ($pid, $output, $startup_command, $ret, $eol, $space, $sock, $ver);

# Initialize variables used
$ps = “/usr/bin/ps”;
$grep = “/usr/bin/grep”;
$ret = 0;
$eol = "\015\012";
$space = $eol x 2;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

$pid_file = $ServerRoot.”/logs/httpd.pid”;
$startup_command = “$ApacheDeamon –d $ServerRoot”;

# Obtain the PID from the PID file
open(PIDFILE,”$pid_file”);
$pid = <PIDFILE>;
close(PIDFILE);

# Remove trailing newline from $pid
chomp($pid);

# Check to see if there is a entry under /proc for $pid
if (-r “/proc/$pid”) {

    # PID exists under /proc
    # Now verify that the PID has not been reused by the system
    # This is done by verifying that the PID matches the startup
    # command used to online the resource

    $output = `$ps –p $pid –o args | $grep –v COMMAND`;

    if ($output !~ $startup_command {
        # The PID does not match the startup command
        # exit offline
        exit 100;
    }

    # PID exists and the arguments match the startup command

    # Now check to see if deep level monitoring is enabled.
    # If it is, then continue on to performing the deep level
    # monitoring checks

    if ($DoDeep == 1) {
        $sock = IO::Socket::INET->new( Proto    => "tcp",
                                       PeerAddr => "$IPAddr",
                                       PeerPort => "$Port",
                                       );

        unless ($sock) {
            # Could not establish a socket connection
            # exit offline
            exit 100;
        }

        # Attempt to perform a HTTP GET request

        $sock->autoflush(1);

        # request head of TestFile (usually index.html)
        print $sock "HEAD $TestFile HTTP/1.0" . $space;

        while ( <$sock> ) {
            if (/^HTTP/) {
                ($ver, $ret, $str) = split(/\s+/o);
            }
            last if ($ret gt 0);
        }

        close $sock;

        # Tests if server responds with valid HTTP response
        if ($ret > 0 && $ret <  408) {
            exit 110;
        }

    }

    # Exit with a ONLINE status
    exit 110;
}

# Return with a OFFLINE status
exit 100;
```

Notice that some additional uses for attributes have been determined. As we have added additional attributes, this will necessitate the modification of the other entry points, and more importantly, the modification of the resource type definition file as shown below:

```
type Apache (
        str ServerRoot
        str ApacheDeamon
        str IPAddr
        boolean DoDeep
        str Port
        str TestFile
        static str ArgList[] = { ApacheDaemon, ServerRoot, DoDeep, IPAddr, Port, TestFile }
      )
```

POLISHING THE CODE

The term ‘polishing the code’ refers to cleaning up the code and adding code that may not be required for the functionality of the script, but is required for another reason. At this point, we have a functional agent that has a high degree of flexibility and monitoring ability. This is not to mean that there is no work left to perform however.

Logging Messages

Currently there is no logging messages incorporated into the agent. This places an administrator at a huge disadvantage in the event that the agent needs to be troubleshot. It also tends to make the user wary of the cluster, as it appears that nothing is happening when the log files are examined.

As a result we will add two different types of log messages. One that is targeted to the end user, and the other with is really only used for debugging or troubleshooting purposes by support personnel. The debug messages will not be displayed in the log files unless they are enabled through the use of the halog –addtag command.

Error Checking

There is also no error checking present in the current routines. As discussed earlier, error checking is critical to stem errors that may occur before a more serious problem is created.

Using Subroutines

Subroutines are also not present within the current agent code. This is not to mean that it is a requirement to have a subroutine, however we have a good use for one. Examine a small snippet of the current offline entry point code:

```
…
# Remove trailing newline from $pid
chomp($pid);
…
```

The chomp function here only removes the tailing new line character from the pid variable. This just happens to work correctly in our current example, but a better method would be to have a subroutine that would be used to clean up or strip the content of a variable. This would ensure that any undesirable characters would be removed.

Following is a review of the current files for the agent that has been polished.

Online

```
# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $command, $cat_id, $msg_id, $ret_code);

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 0;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,online,$cat_id);

$ServerRoot   = _clean($ServerRoot);
$command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# ==============
# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
    if (-x $ApacheDeamon) {
        VCSAG_LOGDBG_MSG (1, "$ApacheDeamon is executable");
    } else {
        VCSAG_LOG_MSG ("C", "$ApacheDeamon is not executable", ($msg_id+1));
        VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+2));
        exit 0;
    }
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+3));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 0;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+6));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+7));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+8));
    exit 0;
}

# Execute the command
VCSAG_LOG_MSG ("N", "Executing the startup command: $command", ($msg_id+9));

$cmd_output = `$command`;

# In the event that troubleshooting of the online procedure is required,
# The following code may assist in troubleshooting when debugging is enabled
$ret_code = $? >> 8;
VCSAG_LOGDBG_MSG (1, "The return code of the startup command was: $ret_code");
VCSAG_LOGDBG_MSG (1, "The output from the startup command was: $cmd_output");

VCSAG_LOG_MSG ("N", "Online Entry point completed", ($msg_id+10));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
```

```
# Accepts:
#   ($str)
#   $str – string to clean
#---------------------------------------
# Returns:
#    $str – cleaned string
#=======================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Offline

```
# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $pid_file);
$kill = “/usr/bin/kill”;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 50;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,offline,$cat_id);

$ServerRoot = _clean($ServerRoot);
$pid_file = $ServerRoot.”/logs/httpd.pid”;
```

```
# Error Checking
# =============

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 0;
}

# Execute the command

# Obtain the PID from the PID file
if (-e $pid_file) {
    open(PIDFILE,”$pid_file”);
    $pid = <PIDFILE>;
    close(PIDFILE);
) else {
    VCSAG_LOG_MSG ("C", "PID File $pid_file does not exist", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 0;
}

# Remove trailing newline from $pid
$pid = _clean($pid);

# Kill PID
VCSAG_LOG_MSG ("N", "Killing PID: $pid", ($msg_id+6));
$cmd_output = `$kill –TERM $pid`;

# In the event that troubleshooting of the online procedure is required,
# The following code may assist in troubleshooting when debugging is enabled
$ret_code = $? >> 8;
VCSAG_LOGDBG_MSG (1, "The return code of the kill command was: $ret_code");
VCSAG_LOGDBG_MSG (1, "The output from the kill command was: $cmd_output");

VCSAG_LOG_MSG ("N", "Offline Entry point completed", ($msg_id+7));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];
    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Monitor

```
# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;
use IO::Socket;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $pid_file, $ret, $eol, $space, $sock,
$ver, $startup_command, $output);

$ps = “/usr/bin/ps”;
$grep = “/usr/bin/grep”;
$ret = 0;
$eol = "\015\012";
$space = $eol x 2;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 100;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,monitor,$cat_id);

$ServerRoot = _clean($ServerRoot);
$pid_file = $ServerRoot.”/logs/httpd.pid”;
$startup_command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# =============
# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+3));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 99;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 99;
}

# Execute the command

# Obtain the PID from the PID file
if (-e $pid_file) {
    open(PIDFILE,”$pid_file”);
    $pid = <PIDFILE>;
    close(PIDFILE);
) else {
    VCSAG_LOGDBG_MSG (1, "PID File $pid_file does not exist");
    VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
    exit 100;
}

# Check to see if there is a entry under /proc for $pid
if (-r “/proc/$pid”) {

    VCSAG_LOGDBG_MSG (1, "Found PID $pid in /proc");

    # PID exists under /proc
    # Now verify that the PID has not been reused by the system
    # This is done by verifying that the PID matches the startup
    # command used to online the resource

    $output = `$ps –p $pid –o args | $grep –v COMMAND`;

    if ($output !~ $startup_command {
        # The PID does not match the startup command
        # exit offline
        VCSAG_LOGDBG_MSG (1, "PID $pid does not match output");
        VCSAG_LOGDBG_MSG (1, "Output is: $output");
        VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
        exit 100;
    }

    VCSAG_LOGDBG_MSG (1, "PID $pid matches output");
    VCSAG_LOGDBG_MSG (1, "Output is: $output");

    # PID exists and the arguments match the startup command
    # Now check to see if deep level monitoring is enabled.
    # If it is, then continue on to performing the deep level
    # monitoring checks

    if ($DoDeep == 1) {

        VCSAG_LOGDBG_MSG (1, "Deep Level monitoring enabled");
        VCSAG_LOGDBG_MSG (1, "Performing socket connection test");

        # Ensure the IPAddr and Port attributes have a value assigned to them
        if ($IPAddr eq “” || $Port eq “”) {
            VCSAG_LOGDBG_MSG (1, "A required attribute was not defined");
            VCSAG_LOGDBG_MSG (1, "Can not continue performing deep level monitoring checks");
            VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
            exit 110;
        }

        $sock = IO::Socket::INET->new( Proto    => "tcp",
                                       PeerAddr => "$IPAddr",
                                       PeerPort => "$Port",
                                       );

        unless ($sock) {
            # Could not establish a socket connection
            # exit offline
            VCSAG_LOGDBG_MSG (1, "Could not establish a socket connection");
            VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
            exit 100;
        }

        VCSAG_LOGDBG_MSG (1, "Performing HTTP GET test");

        # Attempt to perform a HTTP GET request

        $sock->autoflush(1);

        # request head of TestFile (usually index.html)
        print $sock "HEAD $TestFile HTTP/1.0" . $space;

        while ( <$sock> ) {
            if (/^HTTP/) {
                VCSAG_LOGDBG_MSG (1, "Received HTTP header");
                ($ver, $ret, $str) = split(/\s+/o);
                VCSAG_LOGDBG_MSG (1, "Return = $ret");
            }
            last if ($ret gt 0);
        }

        close $sock;

        # Tests if server responds with valid HTTP response
        if ($ret > 0 && $ret <  408) {
            VCSAG_LOGDBG_MSG (1, "Return is within tolerance range");
            VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
            exit 110;
        }

    }

    # Exit with a ONLINE status
    VCSAG_LOGDBG_MSG (1, "Deep Level Monitoring was not enabled");
    VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
    exit 110;
}

# Return with a OFFLINE status
VCSAG_LOGDBG_MSG (1, "PID not found under /proc");
VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
exit 100;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Clean

```
# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
my ($resource_name, $clean_reason, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port,
$TestFile) = @ARGV;

# Initialize variables used
```

```
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $args, $command, $data, $line);
$kill = “/usr/bin/kill”;
$ps = “/usr/bin/ps”;
$grep = “/usr/bin/grep”;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 150;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,clean,$cat_id);

$ServerRoot = _clean($ServerRoot);
$command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# ==============

# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 0;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+5));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+6));
    exit 0;
}

# Execute the command

#  Obtain data from the process table
$data=`$ps –ef –o pid –o args | $grep \”$command\”`;
@data = split(/\n/,$data);

#  Go through collected data and look for any PIDs to kill
foreach $line (@data) {
    ($pid, $args) = split (“ “,$line);
    if ($line eq $command) {
        VCSAG_LOG_MSG ("N", "Found matching line: $line", ($msg_id+7));
        VCSAG_LOG_MSG ("N", "Killing PID: $pid", ($msg_id+8));
        $cmd_output = `$kill -9 $pid`;

        # In the event that troubleshooting of the online procedure is required,
        # The following code may assist in troubleshooting when debugging is enabled
        $ret_code = $? >> 8;
        VCSAG_LOGDBG_MSG (1, "The return code of the kill command was: $ret_code");
        VCSAG_LOGDBG_MSG (1, "The output from the kill command was: $cmd_output");
    }
}

VCSAG_LOG_MSG ("N", "Clean Entry point completed", ($msg_id+9));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Resource Type Definition File

```
type Apache (
        str ServerRoot
        str ApacheDeamon
        str IPAddr
        boolean DoDeep = 0
        str Port
        str TestFile
        static str ArgList[] = { ApacheDaemon, ServerRoot, DoDeep, IPAddr, Port, TestFile }
      )
```

CREATING THE AGENT INFORMATION FILE

Now that we have a polished, functional agent, we can create the Agent Information File for the agent. Remember that the Agent Information File is a XML formatted text file. For this agent this file would resemble the following:

```
<?xml version="1.0">
<agent name="Apache" version="4.0">
    <agent_description>Manages an Apache v1.3 Web Server
    </agent_description>
    <platform>Solaris</platform>
    <agenttype>Script</agenttype>
    <info_implemented>No</info_implemented>
    <minvcsversion>4.0</minvcsversion>
    <vendor>VERITAS</vendor>
    <attributes>
        <ApacheDaemon type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="False"
        persistent="True" range="" default=""
        displayname="Apache Daemon">
        <attr_description>Specifies the full path and name of the Apache Daemon binary
        </attr_description>
        </ApacheDaemon>
        <ServerRoot type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="True"
        persistent="True" range="" default=""
        displayname="Apache Server Root">
        <attr_description>Specifies the full path and name of the Apache Server Root
        </attr_description>
        </ServerRoot>
        <DoDeep type="str" dimension="Boolean" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="[0,1]" default="0"
        displayname="Do Deep Monitoring">
        <attr_description>Specifies if deep level monitoring is enabled
        </attr_description>
        </DoDeep>
        <IPAddr type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="IP Address">
        <attr_description>Specifies the IP address of the Apache web server
        </attr_description>
        </IPAddr>
        <Port type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="Port">
        <attr_description>Specifies the port to connect to the apache web server
        </attr_description>
        </Port>
        <TestFile type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="Test File">
        <attr_description>Specifies the file to attempt to retrieve from the web server
        </attr_description>
        </TestFile>
    </attributes>
    <agentfiles>
        <file name="$VCS_HOME/bin/Apache/ApacheAgent" />
        <file name="$VCS_HOME/bin/Apache/online" />
        <file name="$VCS_HOME/bin/Apache/offline" />
        <file name="$VCS_HOME/bin/Apache/monitor" />
        <file name="$VCS_HOME/bin/Apache/clean" />
    </agentfiles>
</agent>
```

RECOMMENDED AGENT DEVELOPMENT PRACTICES

In this section, recommendations for agent development are presented to the agent developer. Although these are simply recommendations, and not requirements, the use of them tends to decrease upkeep, fault isolation, and transportability of a created agent.

PROGRAM FLOW RECOMMENDATIONS

Program flow refers to how agent scripts are laid out. In general, all agents developed for all platforms can follow the same flow. Following a consistent model reduces the learning curve for support personnel. The recommended program flow for all agent scripts is to provide all recommended header information, followed by the main program loop, and any necessary subroutines.

Main Program Loop

All overall program logic for agent scripts should reside in the main program loop. This is the section of the agent that performs the actual tasks required. It should be designed in such a way that a person can easily understand the tasks performed.

Each function that is performed should be clearly commented to identify the purpose of the function.

Subroutines

Subroutines used should be named in a manner that reflects their purpose. For example, a subroutine that is used to log error messages would be better named “Log_err” verses “a12”. Subroutines in Perl should be named with the first letter of the subroutine name capitalized to differentiate between variable or module usages.

As in the main program loop, all functions should be commented to quickly identify their purpose.

Subroutines should only be used to handle repetitive tasks that may need to be performed within the entry point. The use of subroutines just for the sake of having a subroutine should be avoided.

Modules

With Perl, the use of Perl modules is a common programming practice. These modules normally consist of functions that can perform an action in a consistent manner across a range of scripts. When used with VCS agent development, these modules can perform tasks that are common to all the entry points used.

The use of modules for this purpose provides a great number of advantages. For developers who develop a large number of agents for a company, these modules can standardize the mythology used to perform a particular task. With this standardization comes an increased ease in troubleshooting problems and performing upgrades to the agent.

A common developer problem is that the developer tends to attempt to place all functions in a module. Modules should only be used for repetitive tasks for a number of scripts, and not as a substitute for a subroutine.

RECOMMENDED SECTIONS

Copyright

Although not a requirement of current United States copyright laws, a copyright notice is always preferred to be present in each agent. If an agent consists of multiple entry points written as scripts or separate programs, then the copyright notice should be present in each script.

Header Information

Each program developed should include a series of comments that provide, at a minimum, the following information:

- The name of the application that the program was developed for.
- The version of the application that the program was developed for.
- The version of VCS that the agent was developed and tested with.
- The version of the agent.
- The date that the program was developed.
- A brief description of the purpose of the agent.
- Additional information as required. (Special points of interest, the name of the developer, ect)

This information is a crucial aid in troubleshooting an agent. Additional information can also be provided in the header as needed. Examples of additional information that a developer may include could be a revision history of the agent, amplifying notes on the agent's requirements or function, or notes on the environment that the agent was tested in.

An example of a header section for a custom agent is as follows:

```
#===========================
# Apache version 1.3 monitor
# Agent revision 2.0
# Tested with Apache 1.3 only
# Developed on: VCS 4.0
# Created:  Nov 15 2003
#===========================
# Developed to monitor the Apache web server at process, network
# and httpd service levels
#===========================
#  12/13/03 – Modified network connection model to use standard sockets
#===========================
```

Message ID List

Every message that is outputted from an agent will have a Unique Message ID (UMI) associated with it. These message ID numbers become very important when converting an agent for use internationally. VERITAS will provide a range of message Ids for use with a vendor-developed agent.

A list of the message ID's used within a given entry point, and the message syntax associated with them should be listed under the header information. This will help to create the necessary SMC files required for the internationalism of the agent, as well as providing a quick reference for troubleshooting purposes.
If an agent is modified after it has been created and additional messages have been added, the message IDs should not be renumbered merely to keep the message ID's in numerical order. Simply use the next available message ID number and continue. This prevents problems that can occur when dealing with multiple revisions of the agent.

A sample of the message ID list for an agent entry point is demonstrated below:

```
# Messages Used:
# MSGID         Message
#---------------------------------------------
# 2120006    Unable to open Process ID file %s
# 2120007    Command is %s
# 2120009    Process ID file %s contains an incorrect value
```

Variable Definition

All variables that are used in the program should be defined after the header information. Exceptions to this rule are largely based on the programming language used. In some cases, defining a variable outside of the main program loop would constitute that variable as a global variable. This may need to be avoided to facilitate multithreaded operations. In this scenario, the variables should be defined immediately when possible.

Variables used should always be initialized to an initial value.

Any command that is not native to the programming language used should be defined as a variable. These variables should include the full path to the command.

Variables that are used to hold information passed from the HAD engine should be named in the same manner as in the Types Definition file.

When defining variables within an agent entry point, it is always recommended to include comments as to the purpose of the variable.

A sample of the variable declaration portion of an agent is as follows:

```
# Initialize variables used
my($cmd_output, $command, $cat_id, $msg_id, $ret_code);

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 0;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,online,$cat_id);

$ServerRoot   = _clean($ServerRoot);
$command = “$ApacheDeamon –d $ServerRoot”;
```

The reader will notice that the ‘my’ executable operator is used to initialize the variables. Although for the purpose of this script, this is acceptable, it may be desired to use the ‘local’ operator in some cases. It is assumed that the reader understands the Perl programming language well enough to make the distinction.

NAMING CONVENTIONS

As with any set of standards, a standard naming convention is one of the first issues to consider. In regards to VCS, there are two naming conventions are involved. These are the name of the agent or agent package, and the name of the resource type definition file.

Agents

Agents should be named in a manner that allows their purpose to be easily recognized. The name of an agent should reflect the name of the application the agent is designed for. In the event that the application has several components, then it is desirable for a common base name to be utilized that reflects the main application name followed by an identifier for the component.

The name used should always begin with a capital letter. If the name consists of multiple 'parts', then each part should begin with a capital letter. This allows for easy recognition when listing the contents of the %VCS_HOME/bin directory.

The use of underscore characters and other separators should be avoided.

The following table demonstrates several examples of both proper and improper naming conventions:

Application	Proper Agent Name	Improper Agent Name	Explanation
Apache Web Server	Apache	apache	Capitalize
IPlanet Directory Server	IPlanetDS	Dir_Serv	Do not use the underscore character as a separator.
Netscape Web Server	NSWeb	Webserver	The term 'webserver' does not uniquely identify the application the agent is designed for.

Type Definition Files

The Type Definition files utilized for a given custom agent should be named with the name of the agent followed by 'Types.cf'. For example, if the agent was given the name "ApacheAgent", then the Type Definition file for that agent will be called "ApacheTypes.cf".

Some applications may require several agents. Although it is possible to create one Type Definition file that encompasses all the agents utilized, it is advisable that this not be done as it can affect the portability of the agents used.

It is recommended that the developer always create new Types.cf files and not use the default Types.cf shipped with VCS. This will prevent any future patches to VCS overwriting the developer's work.

GENERAL BEST PRACTICE RECOMMENDATIONS

The following are some best practices not previously mentioned within this document.

- Use caution when using the 'ps' command. Limitations on its output length can cause problems.
- Perl scripts must invoke the correct version of PERL or exit. (eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}' if 0;)
- If a password must be specified in the VCS configuration, it is recommended that the stored password be in an encrypted format.
- Agents should be tested for multithreaded operation as required.
- Agents written in C++ must be tested for memory leaks.
- Ensure that the agent is running with the correct effective UID. This may be performed within the agent entry point as needed.

THIRD GENERATION AGENT EXAMPLE

This section takes the agent developed as part of the second generation agent example and incorporates some of the recommended best practices outlined in the previous section. This final agent development example is indicative of a complete script based VCS agent for the Apache web server.

ENTRY POINTS

Online Entry Point

```
# Copyright(C) 2004 VERITAS Software Corporation.  ALL RIGHTS RESERVED.
# UNPUBLISHED -- RIGHTS RESERVED UNDER THE COPYRIGHT
# LAWS OF THE UNITED STATES.  USE OF A COPYRIGHT NOTICE
# IS PRECAUTIONARY ONLY AND DOES NOT IMPLY PUBLICATION
# OR DISCLOSURE.
#
# THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND
# TRADE SECRETS OF VERITAS SOFTWARE.  USE, DISCLOSURE,
# OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR
# EXPRESS WRITTEN PERMISSION OF VERITAS SOFTWARE.
#
#              RESTRICTED RIGHTS LEGEND
# USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS
# SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH
# (C) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND
# COMPUTER SOFTWARE CLAUSE AT DFARS 252.227-7013.
#              VERITAS SOFTWARE
# 350 Ellis Street, MOUNTAIN VIEW, CA
#===========================
# Apache version 1.3 online EP
# Agent revision 2.0
# Tested with Apache 1.3 only
# Developed on: VCS 4.0
# Created:  Nov 15 2004
#===========================
# Developed to online the Apache web server
#===========================
# Messages Used:
# MSGID        Message
#---------------------------------------------
#1   $ApacheDeamon is not executable
#2   Can not continue operation, Exiting
#3   The required ApacheDeamon attribute is not set
#4   Check the main.cf file for errors
#5   Can not continue operation, Exiting
#6   The required ServerRoot attribute is not set
#7   Check the main.cf file for errors
#8   Can not continue operation, Exiting
#9   Executing the startup command: $command
#10  Online Entry point completed

# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
```

```
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $command, $cat_id, $msg_id, $ret_code);

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 0;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,online,$cat_id);

$ServerRoot   = _clean($ServerRoot);
$command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# ==============
# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
    if (-x $ApacheDeamon) {
        VCSAG_LOGDBG_MSG (1, "$ApacheDeamon is executable");
    } else {
        VCSAG_LOG_MSG ("C", "$ApacheDeamon is not executable", ($msg_id+1));
        VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+2));
        exit 0;
    }
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+3));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 0;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+6));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+7));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+8));
    exit 0;
}

# Execute the command
VCSAG_LOG_MSG ("N", "Executing the startup command: $command", ($msg_id+9));

$cmd_output = `$command`;

# In the event that troubleshooting of the online procedure is required,
# The following code may assist in troubleshooting when debugging is enabled
$ret_code = $? >> 8;
VCSAG_LOGDBG_MSG (1, "The return code of the startup command was: $ret_code");
VCSAG_LOGDBG_MSG (1, "The output from the startup command was: $cmd_output");
```

```
VCSAG_LOG_MSG ("N", "Online Entry point completed", ($msg_id+10));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Offline Entry Point

```
# Copyright(C) 2004 VERITAS Software Corporation.  ALL RIGHTS RESERVED.
# UNPUBLISHED -- RIGHTS RESERVED UNDER THE COPYRIGHT
# LAWS OF THE UNITED STATES.  USE OF A COPYRIGHT NOTICE
# IS PRECAUTIONARY ONLY AND DOES NOT IMPLY PUBLICATION
# OR DISCLOSURE.
#
# THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND
# TRADE SECRETS OF VERITAS SOFTWARE.  USE, DISCLOSURE,
# OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR
# EXPRESS WRITTEN PERMISSION OF VERITAS SOFTWARE.
#
#               RESTRICTED RIGHTS LEGEND
# USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS
# SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH
# (C) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND
# COMPUTER SOFTWARE CLAUSE AT DFARS 252.227-7013.
#               VERITAS SOFTWARE
# 350 Ellis Street, MOUNTAIN VIEW, CA
```

```
#===========================
# Apache version 1.3 offline EP
# Agent revision 2.0
# Tested with Apache 1.3 only
# Developed on: VCS 4.0
# Created:  Nov 15 2004
#===========================
# Developed to offline the Apache web server
#===========================
# Messages Used:
# MSGID         Message
#--------------------------------------------
#51    The required ServerRoot attribute is not set
#52    Check the main.cf file for errors
#53    Can not continue operation, Exiting
#54    PID File $pid_file does not exist
#55    Can not continue operation, Exiting
#56    Killing PID: $pid
#57    Offline Entry point completed

# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $pid_file);
$kill = “/usr/bin/kill”;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 50;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,offline,$cat_id);

$ServerRoot = _clean($ServerRoot);
$pid_file = $ServerRoot.”/logs/httpd.pid”;

# Error Checking
# ==============

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 0;
}
```

```
# Execute the command

# Obtain the PID from the PID file
if (-e $pid_file) {
    open(PIDFILE,”$pid_file”);
    $pid = <PIDFILE>;
    close(PIDFILE);
) else {
    VCSAG_LOG_MSG ("C", "PID File $pid_file does not exist", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+5));
    exit 0;
}

# Remove trailing newline from $pid
$pid = _clean($pid);

# Kill PID
VCSAG_LOG_MSG ("N", "Killing PID: $pid", ($msg_id+6));
$cmd_output = `$kill –TERM $pid`;

# In the event that troubleshooting of the online procedure is required,
# The following code may assist in troubleshooting when debugging is enabled
$ret_code = $? >> 8;
VCSAG_LOGDBG_MSG (1, "The return code of the kill command was: $ret_code");
VCSAG_LOGDBG_MSG (1, "The output from the kill command was: $cmd_output");

VCSAG_LOG_MSG ("N", "Offline Entry point completed", ($msg_id+7));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Monitor Entry Point

```
# Copyright(C) 2004 VERITAS Software Corporation.  ALL RIGHTS RESERVED.
# UNPUBLISHED -- RIGHTS RESERVED UNDER THE COPYRIGHT
# LAWS OF THE UNITED STATES.  USE OF A COPYRIGHT NOTICE
# IS PRECAUTIONARY ONLY AND DOES NOT IMPLY PUBLICATION
# OR DISCLOSURE.
#
# THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND
# TRADE SECRETS OF VERITAS SOFTWARE.  USE, DISCLOSURE,
# OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR
# EXPRESS WRITTEN PERMISSION OF VERITAS SOFTWARE.
#
#               RESTRICTED RIGHTS LEGEND
# USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS
# SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH
# (C) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND
# COMPUTER SOFTWARE CLAUSE AT DFARS 252.227-7013.
#               VERITAS SOFTWARE
# 350 Ellis Street, MOUNTAIN VIEW, CA
#===========================
# Apache version 1.3 monitor EP
# Agent revision 2.0
# Tested with Apache 1.3 only
# Developed on: VCS 4.0
# Created:  Nov 15 2004
#===========================
# Developed to monitor the Apache web server
#===========================
# Messages Used:
# MSGID         Message
#---------------------------------------------
#101    The required ApacheDeamon attribute is not set
#102    Check the main.cf file for errors
#103    Can not continue operation, Exiting
#104    The required ServerRoot attribute is not set
#105    Check the main.cf file for errors
#106    Can not continue operation, Exiting

# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;
use IO::Socket;

#  Obtain values passed to the agent
my ($resource_name, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port, $TestFile) = @ARGV;

# Initialize variables used
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $pid_file, $ret, $eol, $space, $sock,
$ver, $startup_command, $output);

$ps = “/usr/bin/ps”;
```

```
$grep = “/usr/bin/grep”;
$ret = 0;
$eol = "\015\012";
$space = $eol x 2;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 100;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,monitor,$cat_id);

$ServerRoot = _clean($ServerRoot);
$pid_file = $ServerRoot.”/logs/httpd.pid”;
$startup_command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# ==============

# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 99;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+5));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+6));
    exit 99;
}

# Execute the command

# Obtain the PID from the PID file
if (-e $pid_file) {
    open(PIDFILE,”$pid_file”);
    $pid = <PIDFILE>;
    close(PIDFILE);
) else {
    VCSAG_LOGDBG_MSG (1, "PID File $pid_file does not exist");
    VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
    exit 100;
}

# Check to see if there is a entry under /proc for $pid
if (-r “/proc/$pid”) {
    VCSAG_LOGDBG_MSG (1, "Found PID $pid in /proc");

    # PID exists under /proc
    # Now verify that the PID has not been reused by the system
    # This is done by verifying that the PID matches the startup
    # command used to online the resource

    $output = `$ps –p $pid –o args | $grep –v COMMAND`;

    if ($output !~ $startup_command {
        # The PID does not match the startup command
        # exit offline
        VCSAG_LOGDBG_MSG (1, "PID $pid does not match output");
        VCSAG_LOGDBG_MSG (1, "Output is: $output");
        VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
        exit 100;
    }

    VCSAG_LOGDBG_MSG (1, "PID $pid matches output");
    VCSAG_LOGDBG_MSG (1, "Output is: $output");

    # PID exists and the arguments match the startup command

    # Now check to see if deep level monitoring is enabled.
    # If it is, then continue on to performing the deep level
    # monitoring checks

    if ($DoDeep == 1) {

        VCSAG_LOGDBG_MSG (1, "Deep Level monitoring enabled");
        VCSAG_LOGDBG_MSG (1, "Performing socket connection test");

        # Ensure the IPAddr and Port attributes have a value assigned to them
        if ($IPAddr eq “” || $Port eq “”) {
            VCSAG_LOGDBG_MSG (1, "A required attribute was not defined");
            VCSAG_LOGDBG_MSG (1, "Can not continue performing deep level monitoring checks");
            VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
            exit 110;
        }

        $sock = IO::Socket::INET->new( Proto    => "tcp",
                                       PeerAddr => "$IPAddr",
                                       PeerPort => "$Port",
                                       );

        unless ($sock) {
            # Could not establish a socket connection
            # exit offline
            VCSAG_LOGDBG_MSG (1, "Could not establish a socket connection");
            VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
            exit 100;
        }

        VCSAG_LOGDBG_MSG (1, "Performing HTTP GET test");

        # Attempt to perform a HTTP GET request

        $sock->autoflush(1);

        # request head of TestFile (usually index.html)
        print $sock "HEAD $TestFile HTTP/1.0" . $space;
        while ( <$sock> ) {
            if (/^HTTP/) {
                VCSAG_LOGDBG_MSG (1, "Received HTTP header");
                ($ver, $ret, $str) = split(/\s+/o);
                VCSAG_LOGDBG_MSG (1, "Return = $ret");
            }
            last if ($ret gt 0);
        }

        close $sock;

        # Tests if server responds with valid HTTP response
        if ($ret > 0 && $ret <  408) {
            VCSAG_LOGDBG_MSG (1, "Return is within tolerance range");
            VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
            exit 110;
        }

    }

    # Exit with a ONLINE status
    VCSAG_LOGDBG_MSG (1, "Deep Level Monitoring was not enabled");
    VCSAG_LOGDBG_MSG (1, "Returning ONLINE");
    exit 110;
}

# Return with a OFFLINE status
VCSAG_LOGDBG_MSG (1, "PID not found under /proc");
VCSAG_LOGDBG_MSG (1, "Returning OFFLINE");
exit 100;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

Clean Entry Point

```
# Copyright(C) 2004 VERITAS Software Corporation.  ALL RIGHTS RESERVED.
# UNPUBLISHED -- RIGHTS RESERVED UNDER THE COPYRIGHT
# LAWS OF THE UNITED STATES.  USE OF A COPYRIGHT NOTICE
# IS PRECAUTIONARY ONLY AND DOES NOT IMPLY PUBLICATION
# OR DISCLOSURE.
#
# THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND
# TRADE SECRETS OF VERITAS SOFTWARE.  USE, DISCLOSURE,
# OR REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR
# EXPRESS WRITTEN PERMISSION OF VERITAS SOFTWARE.
#
#              RESTRICTED RIGHTS LEGEND
# USE, DUPLICATION, OR DISCLOSURE BY THE GOVERNMENT IS
# SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH
# (C) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND
# COMPUTER SOFTWARE CLAUSE AT DFARS 252.227-7013.
#              VERITAS SOFTWARE
# 350 Ellis Street, MOUNTAIN VIEW, CA
#============================
# Apache version 1.3 clean EP
# Agent revision 2.0
# Tested with Apache 1.3 only
# Developed on: VCS 4.0
# Created:  Nov 15 2004
#============================
# Developed to clean up after the Apache web server
#============================
# Messages Used:
# MSGID        Message
#---------------------------------------------
#151    The required ApacheDeamon attribute is not set
#152    Check the main.cf file for errors
#153    Can not continue operation, Exiting
#154    The required ServerRoot attribute is not set
#155    Check the main.cf file for errors
#156    Can not continue operation, Exiting
#157    Found matching line: $line
#158    Killing PID: $pid
#159    Clean Entry point completed

# Verify Perl interpreter is available
eval 'exec $VCS_HOME/bin/perl5 -S $0 ${1+"$@"}'
        if 0;

# Specify Perl Modules to use
use ag_i18n_inc;

#  Obtain values passed to the agent
my ($resource_name, $clean_reason, $ApacheDeamon, $ServerRoot, $DoDeep, $IPAddr, $Port,
$TestFile) = @ARGV;
```

```
# Initialize variables used
my($cmd_output, $cat_id, $msg_id, $ret_code, $pid, $args, $command, $data, $line);
$kill = “/usr/bin/kill”;
$ps = “/usr/bin/ps”;
$grep = “/usr/bin/grep”;

# Assign the Category ID as directed by VERITAS
$cat_id = 10061;

# Specify the beginning Message ID to use for this entry point
# This is done in case the message id range has to change.
# If so, it only needs to be changed here
$msg_id = 150;

# Set Env variables for logging
VCSAG_SET_ENVS ($resource_name,clean,$cat_id);

$ServerRoot = _clean($ServerRoot);
$command = “$ApacheDeamon –d $ServerRoot”;

# Error Checking
# ==============

# Ensure that the ApacheDeamon attribute has been assigned a value,
# and that the binary exists and is executable

if ($ApacheDeamon ne “”) {
    VCSAG_LOGDBG_MSG (1, "ApacheDeamon assigned $ApacheDeamon value");
} else {
    VCSAG_LOG_MSG ("C", "The required ApacheDeamon attribute is not set", ($msg_id+1));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+2));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+3));
    exit 0;
}

# Ensure that the ServerRoot attribute has been assigned a value,
if ($ServerRoot ne “”) {
    VCSAG_LOGDBG_MSG (1, "ServerRoot assigned $ServerRoot value");
} else {
    VCSAG_LOG_MSG ("C", "The required ServerRoot attribute is not set", ($msg_id+4));
    VCSAG_LOG_MSG ("C", "Check the main.cf file for errors", ($msg_id+5));
    VCSAG_LOG_MSG ("C", "Can not continue operation, Exiting", ($msg_id+6));
    exit 0;
}

# Execute the command

#  Obtain data from the process table
$data=`$ps –ef –o pid –o args | $grep \”$command\”`;
@data = split(/\n/,$data);

#  Go through collected data and look for any PIDs to kill
foreach $line (@data) {
    ($pid, $args) = split (“ “,$line);
    if ($line eq $command) {
        VCSAG_LOG_MSG ("N", "Found matching line: $line", ($msg_id+7));
        VCSAG_LOG_MSG ("N", "Killing PID: $pid", ($msg_id+8));
        $cmd_output = `$kill -9 $pid`;

        # In the event that troubleshooting of the online procedure is required,
        # The following code may assist in troubleshooting when debugging is enabled
        $ret_code = $? >> 8;
        VCSAG_LOGDBG_MSG (1, "The return code of the kill command was: $ret_code");
        VCSAG_LOGDBG_MSG (1, "The output from the kill command was: $cmd_output");
    }
}

VCSAG_LOG_MSG ("N", "Clean Entry point completed", ($msg_id+9));
exit 0;

#=========================================
# Subroutine Name:
#    _clean
#-----------------------------------------
# Description:
# Clean a variable of undesired chars
#-----------------------------------------
# Accepts:
#   ($str)
#   $str – string to clean
#-----------------------------------------
# Returns:
#    $str – cleaned string
#=========================================
sub _clean {
    my ($str,$q);
    $str = $_[0];

    # Remove leading and trailing white space
    $str =~ s/^\s+//;
    $str =~ s/\s+$//;

    # Remove leading and trailing double and single quotes
    $q = '"';
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;
    $q = "'";
    $str =~ s/^$q+//;
    $str =~ s/$q+$//;

    # Remove trailing backslash
    $q = '/';
    $str =~ s/$q+$//;

    return $str;
}
```

TYPE DEFINITION

```
type Apache (
        str ServerRoot
        str ApacheDeamon
        str IPAddr
        boolean DoDeep = 0
        str Port
        str TestFile
        static str ArgList[] = { ApacheDaemon, ServerRoot, DoDeep, IPAddr, Port, TestFile }
      )
```

RESOURCE DEFINITION

```
…
Apache corp_web (
   ServerRoot = “/web/corpweb
   ApacheDeamon = “/opt/Apache/http”
   DoDeep = 1
   IPAddr = “192.168.1.40”
   Port = 80
   TestFile = “/index.html”
   )
…
```

AGENT INFORMATION FILE

```
<?xml version="1.0">
<agent name="Apache" version="4.0">
    <agent_description>Manages an Apache v1.3 Web Server
    </agent_description>
    <platform>Solaris</platform>
    <agenttype>Script</agenttype>
    <info_implemented>No</info_implemented>
    <minvcsversion>4.0</minvcsversion>
    <vendor>VERITAS</vendor>
    <attributes>
        <ApacheDaemon type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="False"
        persistent="True" range="" default=""
        displayname="Apache Daemon">
        <attr_description>Specifies the full path and name of the Apache Daemon binary
        </attr_description>
        </ApacheDaemon>
        <ServerRoot type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="True"
        persistent="True" range="" default=""
        displayname="Apache Server Root">
        <attr_description>Specifies the full path and name of the Apache Server Root
        </attr_description>
        </ServerRoot>
        <DoDeep type="str" dimension="Boolean" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="[0,1]" default="0"
        displayname="Do Deep Monitoring">
        <attr_description>Specifies if deep level monitoring is enabled
        </attr_description>
        </DoDeep>
        <IPAddr type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="IP Address">
        <attr_description>Specifies the IP address of the Apache web server
        </attr_description>
        </IPAddr>
        <Port type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="Port">
        <attr_description>Specifies the port to connect to the apache web server
        </attr_description>
        </Port>
        <TestFile type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="False" unique="False"
        persistent="True" range="" default=""
        displayname="Test File">
        <attr_description>Specifies the file to attempt to retrieve from the web server
        </attr_description>
        </TestFile>
    </attributes>
    <agentfiles>
        <file name="$VCS_HOME/bin/Apache/ApacheAgent" />
        <file name="$VCS_HOME/bin/Apache/online" />
        <file name="$VCS_HOME/bin/Apache/offline" />
        <file name="$VCS_HOME/bin/Apache/monitor" />
        <file name="$VCS_HOME/bin/Apache/clean" />
    </agentfiles>
</agent>
```

APPENDIX A – ADVANCED TOPICS

This section will provide the reader with an introduction to more advanced agent development topics. These advanced topics may not be utilized frequently; however they can demonstrate methods to deal with select corner cases.

USE OF THE CONFIDENCELEVEL ATTRIBUTE

Previously in this paper, the exit values for the monitor entry point were discussed. During this discussion, the ConfidenceLevel attribute was explained briefly. The following will describe in further detail the use of this attribute.

This confidence level is not natively used by the VCS engine at all; however it is stored by the VCS engine and available to be queried. It is dependant upon the agent developer to query the VCS engine for this value for appropriate action as required.

The use of the confidence level within a monitor entry point is not a common occurrence for most VCS agents. But it can be useful when a series of steps are desired to be taken prior to having the monitor entry point return an offline value which, in turn, may cause a resource to be faulted.

Examine the following code example:

```
if ($connector_status{$connector} eq "PAUSED") {
       if ($conf_level == 110) {
              system("$su - $owner -c \"$vtadmin_cmd start connector $folder/$connector\"");
              exit 105;
       } elsif ($conf_level == 105) {
              system("$su - $owner -c \"$vtadmin_cmd stop connector $folder/$connector\"");
              sleep 5;
              do_container_check("$connector");
              system("$su - $owner -c \"$vtadmin_cmd start connector $folder/$connector\"");
              exit 103;
       } else {
              exit 100;
       }
}
```

This snippet of code is from a monitor entry point. Earlier in the script the confidence level value stored by the VCS engine was queried and stored in the $conf_level variable as shown below:

```
# Query VCS for the ConfidenceLevel value for this resource.
$sysname = `uname -n`;
$conf_level = `hares -value $res_name ConfidenceLevel $sysname`;
```

The monitor entry point then uses this value to determine what it should do in certain situations. In this example, if the last return value was 110 (or 100% confidence level) then the monitor simply attempts to restart the resource. If the confidence level is equal to 105, this means to the monitor entry point that the resource had a issue during the last monitor cycle. If there still appears to be a problem, then the monitor attempts to perform a more drastic method of restarting the resource by stopping and then restarting it. On the third time the monitor entry point is called, and there is still a problem, then the monitor entry point returns a value of 100 to signify that the resource is offline. This triggers VCS to perform whatever configured action is required, normally resulting in a faulting of the resource and failing over the resource to another system.

This is a bit of a departure from the normal best practices of developing an agent. The normal train of thought is that the monitor entry point should only perform actions directly related to the actual monitoring of an application or service. If the monitoring should detect a failure, then actions such as restarting the resource are left to the VCS engine to perform. In this particular case however, the resource was only desired to be marked as being faulted after multiple successive monitor failures.

At this point, the VCS engine could still handle the restarting of the resource. The departure from the norm is that for each restart attempt, the resource had to be restarted in a different manner. Using the ConfidenceLevel attribute in conjunction with the monitor entry point allowed for this requirement to be met by the agent developer.

VERITAS Software Corporation
Corporate Headquarters
350 Ellis Street
Mountain View, CA 94043
650-527-8000 or 866-837-4827

For additional information about VERITAS Software, its products, or the location of an office near you, please call our corporate headquarters or visit our Web site at www.veritas.com.