Cluster Test Manager (CTM) for OpenVMS

Richard Stammers
Software Engineer, OpenVMS

Overview

This paper discusses CTM, the OpenVMS Cluster Test Manager. CTM is one of the principal tools used internally for testing OpenVMS in large and high-risk cluster configurations. CTM consists of two parts: the software that is used to manage and control the running of the tests, (known as the CTM test harness), and a variety of tests that are specifically designed and written to be run by CTM (known as CTM test modules).

This paper discusses the CTM test harness, and discusses in general the goals, strategy, and internal design of CTM. Later papers may discuss various CTM test modules.

Introduction

Testing OpenVMS can be very challenging. Not the least of these challenges is the sheer volume of work that is involved in thoroughly testing everything. This is particularly true in a cluster environment, where the possible permutations and combinations of hardware types, versions of the operating system software, and all the other software and their versions, can be gigantic. Providing the means to expedite such a large volume of testing work was one of the primary reasons why CTM was developed.

CTM is not a test as such. Rather, CTM provides a framework, or “test harness,” in which a large variety of tests can be efficiently controlled and managed in an OpenVMS Cluster environment. CTM provides a comprehensive means for managing such testing efforts on large and complex OpenVMS Clusters, where simply using DCL commands or BATCH to run the tests would be impractical. Using CTM hundreds of tests can be distributed across a cluster, and can be started and stopped with just a single command. CTM test runs can involve several thousand varied test processes when CTM is used on large test clusters.

Of course, testing involves more than merely running a lot of tests. The quality and nature of the tests are crucial. However, the way that the tests are sequenced and managed can be just as important to the quality of the testing as the tests themselves. With so much testing work to be done, the testing process can be very much a race against time, so it is essential to use the available testing time as efficiently as possible. In general, problems are much more likely to occur when a system or cluster under test is in a state of flux and is subjected to frequent changes, as against being in a “smooth” or steady state with respect to the tasks that are running. OpenVMS software has a natural tendency to smooth out the load on a system. This is very desirable for normal operations, but it might not give rise to the best conditions for testing. For example, if a group of tests are run continuously for, say, 10 hours and do not detect a problem, this does not in general provide 10 times the confidence level that running those same tests for just 1 hour would provide. In fact, the 10 hours of testing time can generally be used to better effect by constantly varying and modifying the tests that are run during the 10-hour test period. CTM provides the means to do this, “stirring the mix” of the testing, so to speak, by permitting repeated and continuous starting, stopping, and modification of the text mix throughout the test period.

Finding problems is only half the battle. The cause of a problem has to be isolated and identified so that it can be fixed. This can be very challenging in a large and complex cluster, especially particularly because the various components and subsystems of the OpenVMS operating system are so highly interrelated. It might seem rather a simplistic approach, but often the most viable initial testing strategy is to subject the systems under test to extreme stress testing. That is, subject the systems to a veritable barrage consisting of as large and complex and varied a sequence of tests that they can reasonably be expected to sustain. The idea is to “bring them to their knees,” as it were, just to “see if anything breaks.” When a problem is encountered, it can require considerable knowledge and skill to identify its cause. The test manager has to be able to meet this challenge, providing the means whereby the subsequent testing can be organized and sequenced to progressively home in on whatever is causing the problem and to provide the information required to fix it. This means that CTM not only has to provide the means to run a very large and complex mix of tests, but also has to provide the capability to be extremely precise in the selection and sequencing of tests in order to identify and fix problems. A corollary to this is that, with such an extreme degree of interrelationship between the components, if *anything* is changed or fixed, then *everything* needs to be retested.

Overview of CTM

CTM is designed to operate in a cluster environment. In general such clusters may comprise a mix of VAX and Alpha nodes, with CTM providing seamless functionality and testing capability across both the VAX and Alpha platforms. Similarly, CTM provides backwards compatibility for all the previous versions of OpenVMS that are supported and might be present in the cluster. At the time of this writing, a new version of CTM, providing similarly seamless capabilities for HP OpenVMS Industry Standard 64 Evaluation Release Version 8.1 (OpenVMS I64) is in development. CTM can run on a wide range of cluster sizes, varying from clusters comprising a single node through to clusters comprising 60 or more nodes with more than 600 storage spindles.

Perhaps the most fundamental aspect of the design of CTM is that it views the cluster as a whole as a collection of testing resources. In this context, a “testing resource” is any item of hardware within the cluster that can be used for testing. The current CTM implementation has two main categories of testing resources: nodes (single CPU systems or SMP systems) and storage devices (disks or magnetic tapes). This concept of testing resources is key to the whole design. CTM automatically maintains a dynamic database of testing resources that are available on the cluster on which it is running. A separation is made between the hardware resources on the cluster that CTM is permitted to use for testing, and those resources that are protected from CTM. All the resources on the cluster are potentially available for use by CTM. The test engineers are provided the means to dynamically control those resources that are to be used and those that are to be protected. For example the system disks are usually protected – in general, it is not a very good idea to do I/O testing on the system disks!

The tests that are run by CTM have to be specifically written to work with the CTM test harness, and are referred to as CTM test module. At present there are about 25 such test module in regular use, and these are constantly being updated and added to. Again, the concept of test resources is fundamental to a CTM test module. Within CTM, test modules are characterized as requiring certain test resources in order to run. For example, CTM_HIGH_IO is a disk I/O test that repeatedly writes, reads and verifies data to disk. It is characterized as requiring one CPU resource and one disk resource. Similarly, CTM_TCP is a TCP/IP test that causes TCP/IP packets to be exchanged between CPUs within the cluster. It requires two CPU test resources.

The interaction between the resources that are available to CTM in the cluster and the resources that are required by each of the tests is the basis of the CTM design. The cluster resources are described in two databases -- those that are available to CTM, and those that are protected from CTM. A weak pun is used for the naming of these databases that are called, respectively, the CARE (CTM Active Resource Entry) and DONT databases. The resources that each of the s module require are maintained in what is referred to as the Test Module Data Base (TMDB). An additional file, the load-numbers file, contains a numeric description of the performance characteristics of each all the different resource types that could be present in the cluster. Using the information in these files and databases, CTM can automatically start up, load balance, and manage large numbers of tests with very brief commands from the test engineer. The following command examples illustrate this.

CTM Command	Action
$ CTM START CTM_HIGH_IO /process=100	Starts 100 copies of the CTM_HIGH IO test, automatically balancing the load across each of the nodes and disks that are available in the cluster.
$ CTM START CTM_HIGH_IO /perdisk=4	Starts 4 copies of the CTM_HIGH IO, targeting each disk in the cluster, and automatically selecting the CPUs that will perform each of the tests.

In fact, the CARE and TMDB databases contain quite a detailed characterization of the test resources that are available on the cluster and the test resources that are required by a CTM test module. This information provides the test engineer with a considerable range of control for specifying how tests are to be run, ranging from the examples shown where CTM is used to automatically select and load balance the resources that are used, to detailed selection of the characteristics of the resources that the tests will use.

Test engineers interact with CTM through the CTM command center. The CTM command center provides an extensive command-line interface that lets test engineers control the starting and stopping of tests, monitor tests in progress, control test resources in the cluster, generate test reports, and perform all the other functions associated with controlling CTM and the running of CTM tests within the cluster. For convenience and redundancy, and because the component nodes within the cluster may be physically separated, CTM permits any node in the cluster to function as the command center. In fact, there may be multiple command s active on the cluster at any one time (one per node in the cluster). However, at any given time only one command center can act as the master command center, which is capable of issuing action commands that change the state of CTM, such as starting or stopping tests or modifying the characteristics of the test resources within the cluster. Other, nonmaster command s may be active but are confined to issuing interrogative commands, such as generating reports or getting information about active tests.

Organizing and identifying CTM tests

It is not uncommon for a CTM test run to involve running 3000 or more test processes. With so many tests being run it is essential that each tests is uniquely identified so that it can be properly managed and identified. CTM uses a job and process number scheme to identify the tests that it runs. In CTM nomenclature, each line to the command center that starts up test processes is referred to as a job. Job numbers revert to 1 each time CTM is restarted on the cluster. When a job involves starting multiple test processes, each test process within the job is allocated a sequential process number, starting at 1. Hence, the job and process number are always unique for all test processes that CTM is running at any point in time. These unique job/process numbers are referred to CTM as a DPID.

In the preceding example, if the command to start up 100 CTM_HIGH_IO processes were the first job, this would generate 100 test processes with DPIDs 00010001, 00010002, through 00010064 (hexadecimal), respectively. CTM uses the DPID for a variety of purpose. For instance, it is used as the OpenVMS process name on the nodes on which the test runs, is used to generate the names of whatever files the test process create, and is used as part of any data patterns that the test uses.

Viewing CTM test results

In general, every test process generates a test log. All test logs are placed in an area defined by the CTM$LOGS directory, which is accessible to every node on the cluster that is running CTM. The DPID is used to provide a unique name for each test log. The log usually contains header information that identifies the test, the test resources it was using, and the parameters that were used to invoke the test, followed by whatever other log information the test may generate during the test.

Once a CTM test is started, it runs until it is either explicitly stopped by the test engineer, or until a fatal error condition is detected. If a fatal error condition is detected (for example, a data corruption) the test usually generates an error or corruption report and then stops.

As they run, the CTM tests also generate performance and other types of information about their test run, which they periodically send via a mailbox to the TELogging process. This process exists on every participating CTM node in the cluster. It is responsible for adding the information that the test tasks send to it into the TEL data files, which are a common repository for information on all the tests that are running or have been run by CTM on the entire cluster

CTM provides a utility known as the TRU (TEL Reporting Utility) that can be run on any node in the cluster at the behest of the test engineers. The TRU generates reports based on the information in the TEL data files, and permits this information to be organized and presented by a variety of different criteria, such as time, or device or device type, or node or node type. This allows the test engineers to see what is happening on the cluster as a whole as well as providing the means to monitor the performance of individual systems under test, devices, or test processes.

Data patterns

There are no real constraints on the functionality of the tests that can be run within CTM, as long as the test complies with both the conventions required to communicate with the CTM test harness, and the needs and inventiveness of whoever is writing the test. However, a theme that is common to many of the tests is the transfer of data. For example, the CTM_WIDEST test involves transferring data between different areas of memory; the CTM_HIGH_IO test involves transferring data to and from disk files; and the CTM_TCP test involves transferring TCP/IP packets between different nodes on the cluster. The common theme is that a data pattern is generated, the data is transferred, and the results of the transfer are checked and validated. Data corruption is probably the most serious and most dangerous problem that can ever occur -- much worse, in fact than a process or system failure because of the potential to destroy user data without warning. Thus, great importance is attached to the data patterns that are used, and the requirements for the data patterns are quite demanding:

- The data pattern should provide the best possible chance of detecting corruptions.
- Wherever feasible, the data pattern should provide as much evidence and as many clues as possible as to what caused the corruption
- The data patterns should allow quick construction and verification, so that as many test iterations as possible can be performed within a given time.

Many techniques are employed within the CTM tests to meet these requirements. In general, each test uses its unique DPID within the data pattern so that if a crossover of data occurs between test tasks, the source of the offending data can be identified. For example, if multiple CTM_TCP tasks are exchanging packets, and a packet is erroneously sent to or received by the wrong recipient, the source of the bad data can be identified. Similarly, sequence numbers are often used within the data patterns so that dropped data can be identified. When writing to disks, the block number and position of the data fields within the blocks are usually incorporated into the patterns in order to provide more complete information about the nature of the corruption. Various schemes are used to vary the positions and alignment of the data buffers with respect to OpenVMS pages and disk blocks.

Speed is also of the essence, with respect to both creating and verifying the data patterns. The usual technique is to build all the data patterns or as many of them as possible at the start of the test so that time is not wasted on this during actual test iterations. Fields within the data patterns are usually organized and aligned on quadword or longword boundaries so that the verification can be done very briskly, and routines to do the compares and verification are frequently written in assembly language for the same reason.

When elements of the data patterns are common between successive transfers, the transfer may not actually take place properly, but because of the stale data in the receive area, a corruption might go undetected. Poisoning the receive areas before the transfer is one way of avoiding this, but this technique is avoided in the CTM tests because of the time it takes. Instead, a technique of alternating the data that is transferred with the 1s complemented form is used. In this way, literally every bit of the transferred data is changed on each successive transfer, and the performance hit is taken only once, up front, when two forms of the data pattern – the “true” data and the 1s complemented form are built.

Tracking down problems

Normally, a CTM test runs until it is explicitly stopped by the test engineer, or until a fatal error is detected. Almost all errors that the tests detect are considered to be fatal.

If a fatal error condition (such as, a data corruption) is detected, the test usually generate an error report and then stops. The nature of the error report depends on the nature of the test, but where the test involves a data transfer, a corruption report is always generated. To simplify the debugging process, the corruption reports usually show the expected data alongside the actual data, with the incorrect or corrupted fields flagged so that they can be readily identified. As described earlier, the data patterns often involve a 1s complemented form of the “true” data, and because humans are not very good at reading 1s complement, a translated hexadecimal form of the data is provided for convenience.

For some tests, a corruption report is not always sufficient to debug the problem. For example, with the CTM_TCP test or the CTM_LOCK_IT test (a CTM test that stress tests the Distributed Lock Manager), a “history of events” leading up to the error is often required to debug what is going wrong. In these instances, the test maintains a ring buffer that describes events leading up to the failure, which is output before the test is stopped.

In extreme cases, a crash dump may be needed to debug the problem, and for some tests a parameter is provided that is used to that specify that a bugcheck is to be performed when the problem is detected. If a bugcheck is needed, then the ensuing crash dump is most valuable if it is produced as soon as possible after the problem is detected. In these instances, no reports are generated.

In very extreme cases, a bugcheck might be required on multiple nodes on the cluster in order to debug a problem. For example, a CTM_HIGH_IO test might be running on one node in the cluster, targeting disk I/Os to a disk that is served by a different node on the cluster. When a problem is detected it might require a timely crash dump of both these nodes in order to figure out what is going wrong. CTM uses a technique known as “triggering” to accomplish this.

The way that triggering works is that a “trigger arming” routine is provided with CTM that permits a node to arm itself on a trigger. There are 256 such triggers available. In the preceding example of the CTM_HIGH_IO test, the node that was serving the disk would be armed on a trigger. When a node arms on a trigger, it uses DLM to take out shared access to a lock that is associated with the trigger, and specifies a blocking asynchronous system trap (AST) to fire when this lock is lost. The blocking AST routine performs the bugcheck. When the node that is conducting the CTM_HIGH_IO test detects the corruption, it takes out exclusive access to the lock in question, thereby causing both the blocking AST to fire on the node that is serving the disk and a timely crash dump to be generated.

Tracking down problems in a large and complex cluster can sometimes become very complicated indeed, particularly if the problem is intermittent and occurs very infrequently. The problem might be caused by any one of a number of components, or by the interaction between multiple components that are distributed across the cluster. Triggering permits multiple nodes in the cluster to be made aware very quickly when a test process on any given node detects a problem. In such instances multiple different nodes can each be armed on a number of triggers. Even a crash dump might not be sufficient to figure out what is going wrong. For this reason, the functionality that is invoked by triggering on the nodes can be changed to do things other than bugcheck by changing the functionality within the blocking AST that is fired when a node is triggered. For example, the blocking AST might be modified to output a data pattern that can be identified on a data scope or bus analyzer when a problem is detected and the node is triggered.

Varying test mixes and sequencing tests.

As suggested previously, the most efficient use of test time -- certainly in terms of surfacing particularly nasty problems -- is often achieved by varying the loads and changing the mix of tests that as much as possible on the systems under test. Although it's difficult to be very specific about what constitutes a "nasty problem," such problems leave little doubt as to their nature when they have to be fixed. They also seem to have certain common characteristics. For instance, they tend to happen intermittently and infrequently. Often the way they manifest themselves is not directly related to what caused them and provides few clues as to what is actually causing them. The problem may not be detected until some time after the original problem event occurred by which time the context that caused the problem has changed. Sometimes they happen as a result of an unusual combination of circumstances or interactions between components that are otherwise thought to be very solid and reliable. The effects can be very serious -- data corruptions, system failures, and so on.

The observation that these types of problems are most likely to be found by varying the testing mix as much as possible is borne out by both experience and empirical results. However, it is not hard to also think of theoretical reasons why this should be the case. A large OpenVMS Cluster is an amazingly complex system, and looking for problems in it can be likened to searching an immense combinatorial tree of all the possible states that the cluster can be in. The greater the variety of the tests and more the tests are mixed, the greater the area of the tree that will be searched. Hence, the increased probability that a problem will be found. On a more prosaic level, these types of problems often occur when unusual events occur, such as infrequent timers expiring, queues becoming full, or buffers overflowing. Continually stirring the test mix and spiking the loads tends to make these types of things happen more often than when the systems under test are left in a relatively constant state.

CTM provides the capability to mix tests, spike loads, and otherwise "torture" the systems under test by means of what are referred to as pulses. Logically, a CTM pulse can be thought of as a square wave that alternates indefinitely and with a certain specified periodicity between two states – blocking and free. A CTM pulse is shown schematically in Figure 1.

The pulses are implemented by means of the CTM pulse utility, which enables up to 256 such pulses to be defined and made available to every node in the cluster. Each pulse is completely independent as to when it starts and the amount of time it will subsequently block and free tests. When a CTM job is started, any of the defined pulses can be specified, with the effect that all the test processes associated with that job will stop and start in accordance with the specified pulse.

The pulse functionality is implemented by means of DLM. The pulse utility creates a lock that is associated with each pulse that is defined. It blocks by converting to exclusive access on the lock, and frees by converting to null access on the lock, in accordance with the timings specified in the definition of the pulse. Tests processes that have been instructed to synchronize to the pulse precede each test iteration with an attempt to convert to shared access to the lock, and hence run only when the pulse is free and are stopped when the pulse blocks.

Because each job can use any one of 256 different pulses, there is considerable flexibility for varying the mix of tests that are running at any one time. In addition, because the state of the locks is so rapidly communicated across the cluster, they provide a means to generate very sharp spikes in the load to which the systems under test and the cluster as a whole are subjected.

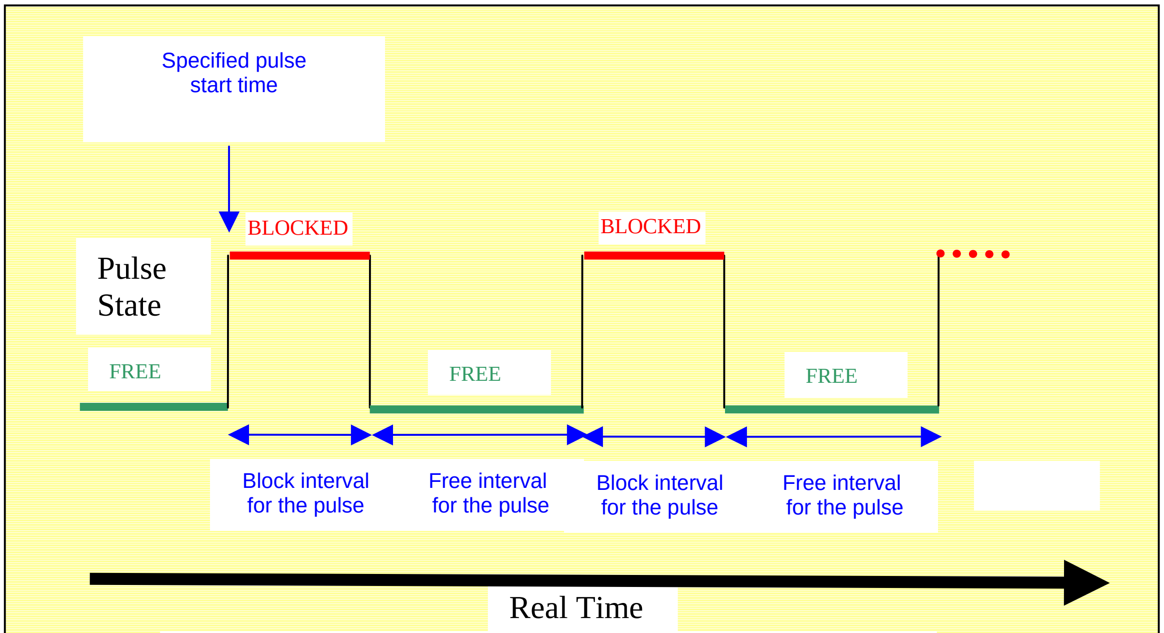
Specified pulse start time
BLOCKED
BLOCKED
Pulse State
FREE
FREE
FREE
Block interval for the pulse
Free interval for the pulse
Block interval for the pulse
Free interval for the pulse
Real Time

Figure 1. Schematic Representation of a CTM Pulse

Pulses can also be used as a diagnostic aid to isolate and identify components and combinations of components that cause problems. This is accomplished by successively “switching in” tests that use different components until the problem is detected, thereby allowing the culprits to be determined by the process of elimination.

Overall CTM design

Figure 2 provides a schematic view of the design and principal components of CTM.

In order to provide the highest degree of fault tolerance, the CTM harness software is based on a heterarchical rather than a hierarchical design. That is, there is no single “control” node or control process within the cluster that controls the running of the tests. Therefore, CTM as a whole is not vulnerable to failures on any given node in the cluster. Instead each node participates as a CTM peer within the cluster and each is capable of and required to perform all the tasks required by CTM. This redundancy is very important because CTM is often used to test high-risk clusters where some of the components are in early testing.

A participating CTM node always has the CTM server process, and the TELogger (Test Event Logger) process running. The CTM command center process is run whenever a participating node is explicitly called upon to act as the command center. The TRU (Test Report Utility) process is run whenever a participating node is explicitly called upon to generate a test report, and the Test Module Data Base (TMDB) utility is run whenever a participating node is called upon to modify or interrogate the TMDB.

The CTM related processes that run on participating CTM nodes communicate and share information by means of common databases and directories that are pointed to by logical names that are common to all participating CTM nodes in the cluster. Access to these databases and directories is controlled and synchronized, as required, by the Distributed Lock Manager (DLM). DLM is also used extensively for signaling and synchronization between the various processes that constitute CTM running on each of the participating nodes.

Note that the CTM “databases” are not databases in the usual sense, but rather are specially constructed RMS files. The OpenVMS File Definition Language (FDL) utility is used to create many of them.

CTM databases and related files

A number of directories, databases and files are involved in the implementation of CTM within a cluster, and are shared by and common to all the participating CTM nodes. The principal ones are as follows.

CTM$LIBRARY
This area contains the load-numbers file mentioned earlier, and all the executables, command files, and other files that constitute the CTM test harness. In general, there are separate versions of each of these files for the VAX and Alpha platforms. When any CTM-related process is to be run, the type of platform is identified and the appropriate version for that platform is used.

By convention, any corruption and error reports that a test generates are usually placed in this area as well.

The OpenVMS I64 version of CTM that is in development is very similar as regards the directories, databases and files that are used, but instead contains versions of the CTM test harness files that are appropriate for the OpenVMS I64 platform.

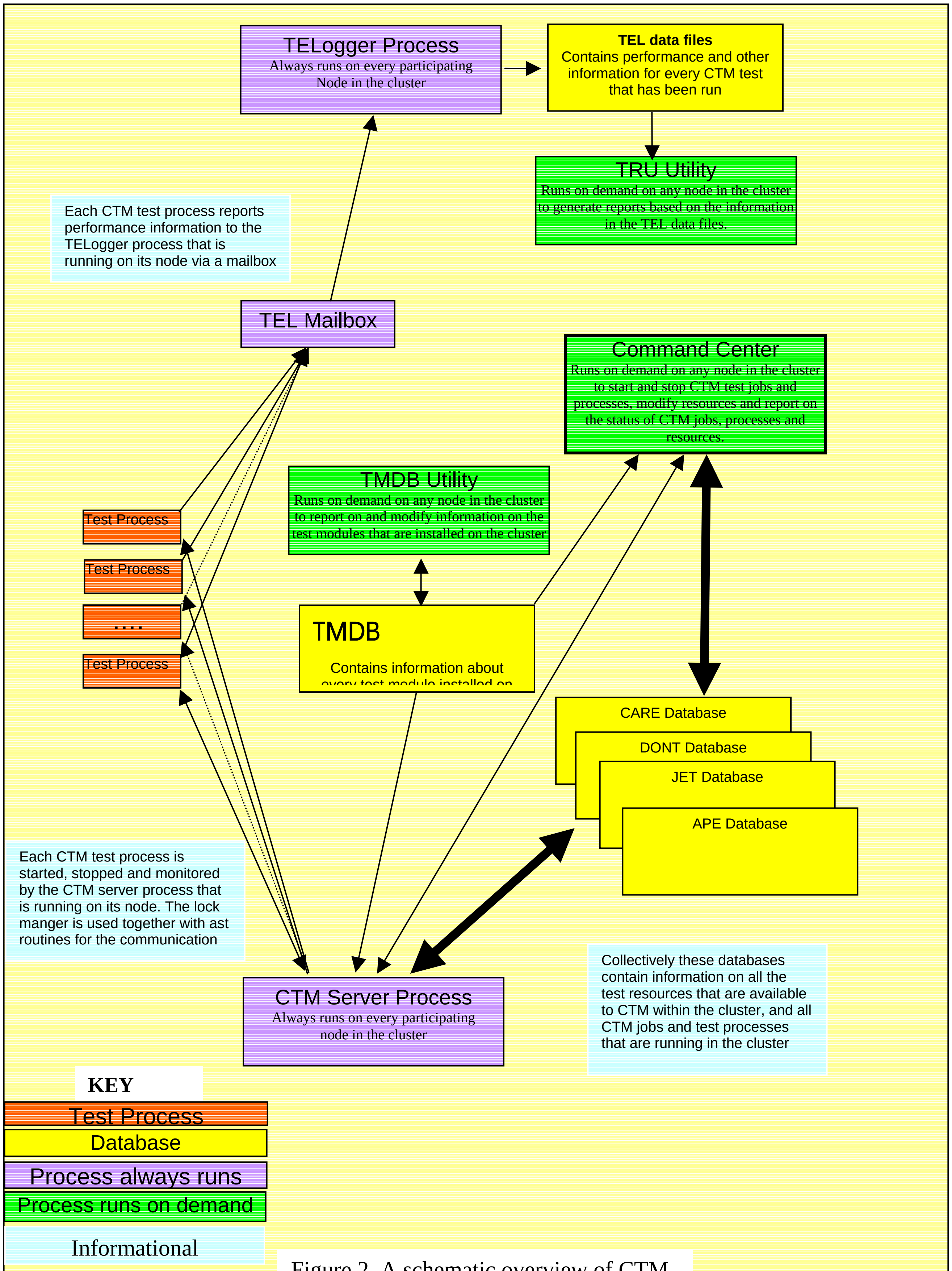
TELogger Process
Always runs on every participating Node in the cluster
TEL data files
Contains performance and other information for every CTM test that has been run
TRU Utility
Runs on demand on any node in the cluster to generate reports based on the information in the TEL data files.
Each CTM test process reports performance information to the TELogger process that is running on its node via a mailbox
TEL Mailbox
Command Center
Runs on demand on any node in the cluster to start and stop CTM test jobs and processes, modify resources and report on the status of CTM jobs, processes and resources.
TMDB Utility
Runs on demand on any node in the cluster to report on and modify information on the test modules that are installed on the cluster
Test Process
Test Process
. . . .
Test Process
TMDB
Contains information about
CARE Database
DONT Database
JET Database
APE Database
Each CTM test process is started, stopped and monitored by the CTM server process that is running on its node. The lock manger is used together with ast routines for the communication
Collectively these databases contain information on all the test resources that are available to CTM within the cluster, and all CTM jobs and test processes that are running in the cluster
CTM Server Process
Always runs on every participating node in the cluster
KEY
Test Process
Database
Process always runs
Process runs on demand
Informational

Figure 2. A schematic overview of CTM

CTM$TMROOT

This area contains all the executables, command files, and other files that are required for whatever CTM test module are installed on the system. Again, in general there are separate versions of each file for the VAX and Alpha platforms; when any CTM test module related process is to be run, the type of platform is identified and the appropriate version for that platform is used.

Similarly for the OpenVMS I64 version of CTM that is in development this area contains versions of the CTM test module that are appropriate for the OpenVMS I64 platforms.

CTM$DATABASES.

This area contains the major CTM databases that are used by the cooperating CTM processes within the cluster to record the test resources that are available to CTM, the test jobs and test process that are in progress, and the test resources that are being used by them. Access to these databases is protected and controlled by means of DLM.

CARE and DONT Databases

These databases are used by CTM to record the test resources that are available within the cluster and to record whatever cluster resources are protected from CTM. CTM automatically “probes for” and maintains a record of whatever test resources are available. Protected resources are explicitly controlled by the test engineers, either from the command center or by appropriately modifying the CTM startup sequence.

APE and JET Databases

The Job Entry Table (JET) database and the Active Process Entry (APE) database are used by CTM to record the tests that are in progress and the test resources they are using, and are maintained automatically by CTM. As described earlier, in CTM nomenclature each command issued at the command center is considered a job, and the various test processes that are started for that CTM job are considered processes within that job. The job/process number is always unique for all the test processes CTM is running at any point in time.

There is no correspondence between a job and a process as understood by OpenVMS, and a CTM job or process – other than each CTM job/process gives rise to a unique OpenVMS process on whatever node or nodes involved in that test. These tests take the DPID as the OpenVMS process name.

Loosely speaking, the JET database contains information about the jobs that CTM is running, and the APE database contains information about the processes that CTM is running and the test resources they are using. However, there is a lot of cross-pointing and shared information between the CARE, DONT, JET, and APE databases.

CTM$TMDB

This area contains the TMDB database that describes all the test module that are installed on the cluster, a characterization of the resources they require, and other information pertaining to how they are to be run.

TEL Data Files

As each test process runs, it may periodically report metrics on the performance of the test as well as other information about the progress of the test. All such information is collected in the TEL, or Test Event Logging, files. All TEL data from all the tests that have run since CTM was installed is kept in TEL data files. This data is cleared only when CTM is reinstalled or when the database is explicitly reinitialized.

CTM$LOGS

This area is a repository for the logs that each of the tests produces. The test logs are named simply by the unique DPID of the test process to which they relate.

In addition, each of the CTM server process that are running on all the participating nodes in the cluster maintain their own logs, which are also kept in this area.

CTM test harness software

Normally, on a cluster where CTM is to be used, a CTM$STARTUP command file is invoked as part of the system startup procedure for every node in the cluster that is to participate in the CTM testing. This command file defines the CTM related logicals as described earlier, and installs the CTM server and the TELogger images on each of these nodes.

Subsequent logging in to any of these nodes as CTM invokes a LOGIN.COM file that uses the Command Definition Utility (CDU) to define the command CTM to run the CTM command center process and to define the syntax of all CTM commands. The commands and their syntax are quite extensive, reflecting all the functionality that is provided by the CTM command center.

Similarly, the commands and the syntax for the TMDB utility (which is used to modify and update the Test Module Data Base) and for the TRU utility (which is used to generate test reports) are defined at this time.

A general mechanism and technique is employed by which all the cooperating CTM processes signal and communicate with each other across the cluster. DLM is used extensively, and each cooperating node creates a series of locks whose names are predicated on the node's name in the cluster. Each node prepares itself to be signaled by taking out shared access to its own locks, specifying blocking ASTs that are to fire when they are signaled. Other nodes that wish to signal that node then initiate communication by taking out exclusive access to the appropriate lock for the node they wish to signal, thus causing that nodes blocking AST to fire.

The CTM Command Center Process

This process can be run on any participating node in the cluster. It provides a command-line interface (CLI) that allows the test engineer to control and interrogate CTM. The first instance of this process being run becomes the CTM command center master. As such, the CLI provides a command set that provides the following functionality.

- Starting (booting) and stopping CTM as an active participant on each nodes in the cluster
- Making test resources available to CTM or protecting test resources from CTM
- Mounting and dismounting the storage devices that will be used as test resources by CTM
- Starting and stopping the CTM jobs and test processes
- Interrogating CTM about what test resources are available, the CTM jobs and test processes that are running, and the test resources they are using
- Generating reports from the TEL data files

Booting CTM

As the command center master, the CLI provides a command set that permits CTM to be stopped and started (booted) as an active participant on each node in the cluster that has successfully completed the CTM$STARTUP sequence. Normally, the first operation that is performed at the command center is to boot CTM on one or more nodes in the cluster.

Booting CTM involves several stages, one of which is to start up the CTM server and TELogging process on each node to be booted. The general mechanism by which CTM processes use DLM to signal each other across the cluster is used. Another task involved in booting CTM is to create or update the CARE database (the database of resources that are available to CTM). This is done by a sequence of wildcarded calls to SYS$GETSYI and SYS$GETDVI to identify the resources that are available on the cluster. Similarly, the JET and APE databases are also initialized to show that there are no jobs currently being run by CTM.

Controlling CTM resources

Requests to change the protection of test resources are handled by making the appropriate changes to the CARE and DONT databases. If the changed protection affects a resource that is in use by a test that is already running the status of that test can also be modified in the APE and JET databases. Commands to mount and dismount storage devices are handled in a similar fashion, and the node on which the device is being served is signaling to perform the mount or dismount in batch.

Starting and stopping CTM jobs and test processes

To start a job, the command center master is involved in varying amounts of work, depending on which test resources the test engineer has unambiguously specified in the command. The resource requirements for the specified test are first established from the TMDB. If all of these resources are unambiguously specified in the command, then details of the job are placed in the JET database, and the node that was specified is signaled to start the test processes. If the test resource is not specified clearly, the command center must determine what test resource to use. It attempts to load balance by referencing the tests that are already running (as described in the APE and JET databases), the load-numbers file, and various internal algorithms. Once the test resources to be used are identified, details of the job are then placed in the JET and the selected nodes are signaled to start the test processes. Handling commands to stop test processes is much easier; the nodes that are running the test processes are simply identified from the JET database, and then are signaled to stop the test processes.

Interrogating CTM and generating reports

When the command center is already running, any subsequent invocations of CTM give rise to a nonmaster version of the command center. As nonmaster the command CLI provides a reduced command set that is limited to generating reports and interrogating CTM about the resources that are available and about the jobs and test processes that are running. Requests for information about CTM resources and jobs and processes that are running are handled by examining the CARE, APE, and JET databases. Requests for reports are handled by calling the TEL Reporting Utility to generate the requested report based on the contents of the TEL database.

The CTM Server Process

This process runs on every participating node in the cluster on which CTM has been booted. Its primary purpose, based on commands signaled from the command center, is to stop and start CTM test processes on the node on which it is running, and to periodically monitor whether these tests are still alive.

Starting a CTM test process involves several stages. The TMDB for the test process is examined, and the required test resources are verified and are assigned to the test process. The OpenVMS context in which the test is to run is established and is described to the test by a series of process local symbol definitions. For example, the DPID that the test is to use is defined as a local process symbol. Similarly, if the test is required to create data files, a directory for these files is created on behalf of the test and is also defined as a process local symbol. Finally, the test process is started by creating an OpenVMS process to run a DCL command file that is specific to the test.

The CTM server process is responsible for maintaining the status in the APE of all the test processes that are running on the node. Because there can be so much work occurring on the test nodes, they are often extremely busy and there can be significant delays in the communication between the CTM server process and the test processes it is running. After each test process is started, it is "pinged" periodically by the CTM server process to ascertain that it is still alive. When the first correct response to the ping is received back from the test process it is flagged as RUNNING. If a test process fails to respond to the pings in a timely fashion, it is eventually flagged as MIA. When a test encounters a fatal error and stops, the CTM server process is notified and the test is flagged as DEAD. Stopping a test at the behest of the command center can also involve significant delays if the system is very busy. The test's ping is changed to a request to stop, and the test is flagged as FIP (finish in progress). When the test process eventually stops, the CTM server process is notified and the test is flagged as FINISHED.

As long as CTM is booted and running on a node, a secondary function of each CTM server process is to wake up periodically to verify that the contents of the CARE, APE, and JET databases are accurate and up to date. Access to each of these databases is controlled by the acquisition of DLM locks. If the state of the locks indicates that a CTM server process on another node is verifying the databases, no action is taken, and CTM relies on the functionality of the peer server process to maintain their accuracy. If not, the databases are "walked" and verified by issuing a series of SYS$GETSYI, SYS$GETDVI and SYS$GETJPI system service calls to verify that the contents of the databases are accurate and up to date.

The TELogger Process

This process runs on every participating node in the cluster on which CTM has been booted. Its function is to accept messages, such as performance metric information from the test tools, and to write that information to the TEL data files.

The Test Module Data Base Utility

The TMDB utility is used to create, maintain, and report information about the CTM test module that are installed on the cluster. It is run on demand, and it can be run on any participating node in the cluster. It handles such information as the name of test module, a characterization of the test resources used by each of the test module, and other information pertaining to how the tests are to be run.

The TRU Reporting Utility

This utility is used to create generate reports from the data that is contained in the TEL database. It is run on demand, and can be run on any participating node within the cluster.

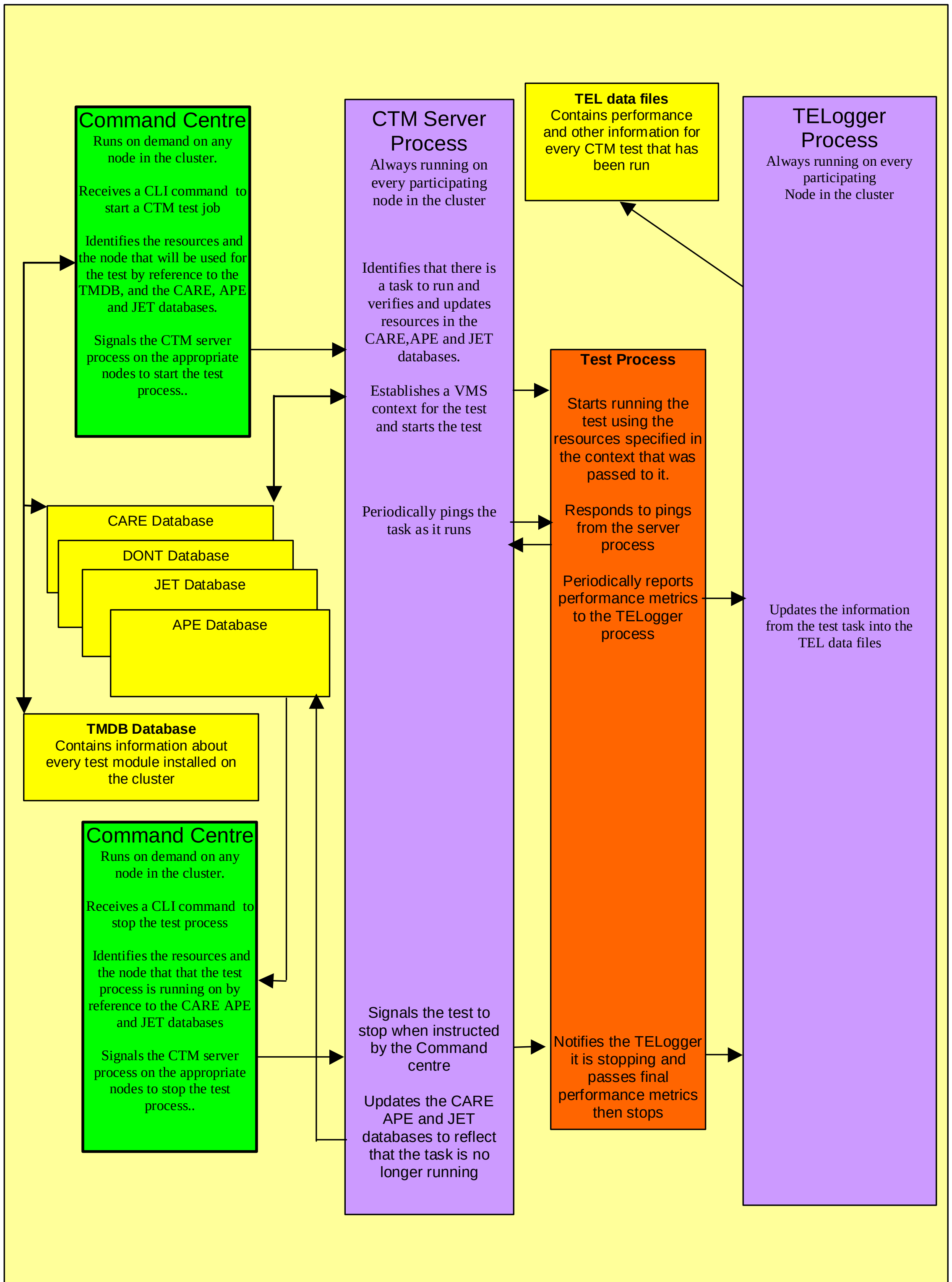
Command Centre
Runs on demand on any node in the cluster.
Receives a CLI command to start a CTM test job
Identifies the resources and the node that will be used for the test by reference to the TMDB, and the CARE, APE and JET databases.
Signals the CTM server process on the appropriate nodes to start the test process..
CTM Server Process
Always running on every participating node in the cluster
Identifies that there is a task to run and verifies and updates resources in the CARE,APE and JET databases.
Establishes a VMS context for the test and starts the test
Periodically pings the task as it runs
Signals the test to stop when instructed by the Command centre
Updates the CARE APE and JET databases to reflect that the task is no longer running
TEL data files
Contains performance and other information for every CTM test that has been run
TELogger Process
Always running on every participating Node in the cluster
Updates the information from the test task into the TEL data files
Test Process
Starts running the test using the resources specified in the context that was passed to it.
Responds to pings from the server process
Periodically reports performance metrics to the TELogger process
Notifies the TELogger it is stopping and passes final performance metrics then stops
CARE Database
DONT Database
JET Database
APE Database
TMDB Database
Contains information about every test module installed on the cluster
Command Centre
Runs on demand on any node in the cluster.
Receives a CLI command to stop the test process
Identifies the resources and the node that that the test process is running on by reference to the CARE APE and JET databases
Signals the CTM server process on the appropriate nodes to stop the test process..

Figure 3. A schematic representation of how a CTM test process is run

CTM Test Modules (Test Processes)

Figure 3 provides a schematic representation of how a test process is run by CTM.

The primary function of a CTM test module is, of course, to perform whatever actual testing functionality is required. Several such test modules have been mentioned in the foregoing description of CTM. For example, the CTM_HIGH_IO test performs disk I/O testing by means of repeatedly writing, reading back, and then verifying data patterns to and from disk. Because CTM test modules have to be written specifically to work within the CTM test harness, they also must perform certain additional tasks.

All test processes are initiated from the CTM command center, which resolves any ambiguity concerning the resources that the test is to use, including the node on which the test is to run. The CTM command center then signals the CTM server process on the selected node to start the test. The CTM server process on the signaled node establishes an OpenVMS context for the test process and then starts it.

In order to interface with CTM, the test module must use the test context provided by the CTM server process by using parameters that are passed to the test by the CTM server process. These parameters establish the process logical names that describe them. By convention, the DPID that is established by the CTM server process and that is passed as a process logical name is used in building whatever the data patterns the test may use. Similarly, the directories and file names that the test module uses must comply with what the CTM server process tells the test to use.

The test is also required check in periodically with the CTM server process and to respond periodically to pings from the CTM server.

As the test process runs it periodically sends the TELogging process that is running on that node performance metric information for the test. The TELogging process places this information in the TEL data files, where it can be accessed and reported on in the context of other activity taking place on the cluster. This confers a great deal of standardization to the reports that are generated and results in a considerable saving of effort by eliminating the onerous task of writing lengthy report-generating programs for each test.

Assuming that the test does not encounter a fatal error, it is eventually stopped by an explicit command from the CTM command center. In turn, the CTM server process signals the test task to stop. Upon receiving this signal, the test task makes one final report on the overall test metrics to the TELogger and then stops. Once the test process has stopped, the CTM server process updates the various CTM databases and files to reflect that the test is no longer running.

Summary

After many years of use, there can be no doubt that CTM is an invaluable tool for testing OpenVMS on large clusters, and that it provides power and flexibility that goes beyond what could be provided by any single test. The TEL Reporting Utility (TRU) provides reports and clusterwide overviews of the testing and test history that are invaluable and that would not otherwise be possible.

Although this paper began by talking about the challenges involved in testing on OpenVMS, the greatest challenge of all was left unstated. That challenge is to continue to maintain the highest levels of quality and reliability for which OpenVMS is renowned. The hope and expectation is that CTM will continue to help meet this challenge in the years to come.