Understanding VMScluster State Transitions

Keith Parris

VMS Engineering

May 23, 1992; 2nd draft

Revision/Update Information: Version X0.1, 23–May–1992, Keith Parris, STAR::PARRIS

NOTE

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Copyright 1992 Digital Equipment Corporation, Maynard Massachusetts

Digital Equipment Corporation
Maynard, Massachusetts

Introduction

Notes to presenter:

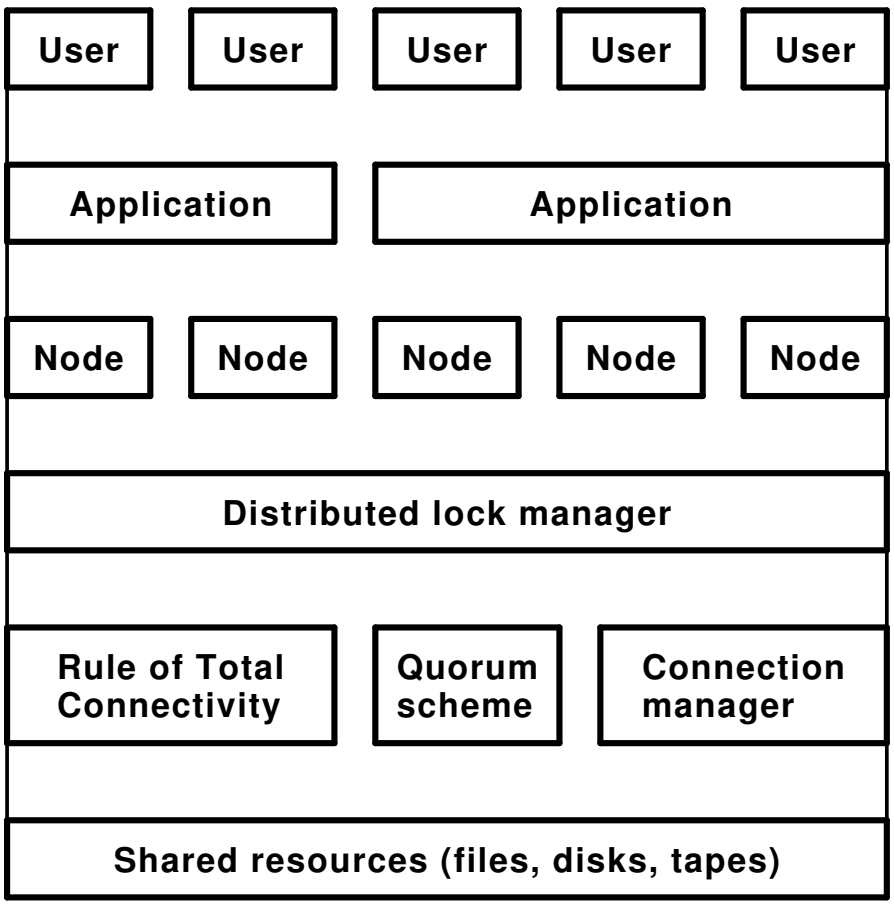
This session is VA037, Understanding VMScluster State Transitions, scheduled for Thursday at 16:00 to 17:00.

The messages we need to get across are:

- State transitions are not just an annoying and useless pause; important things happen during them.
- With due care, one can minimize the duration and depth of impact of state transitions on user work.

Why have state transitions? Foundation for shared access to resources

Figure 1: Foundation for shared access



ART_FOUNDATION

VMScluster configurations provide a way for

- Computer users

  to share computer resources in a coordinated manner.

- Applications

  use VMS services to access these shared resources. These applications may run on various

- Nodes

  in the cluster. Coordination is required among all the nodes in the cluster to do this. The various VMS services use the

- Distributed lock manager

  to control access to shared resources such as disks, files and records. The

- Rule of Total Connectivity

for configurations specifies that each node must be able to talk directly with all the other nodes in the cluster. The

- Quorum scheme

  ensures that a majority of the possible cluster members are present and operational before access to shared resources is attempted. The

- Connection manager

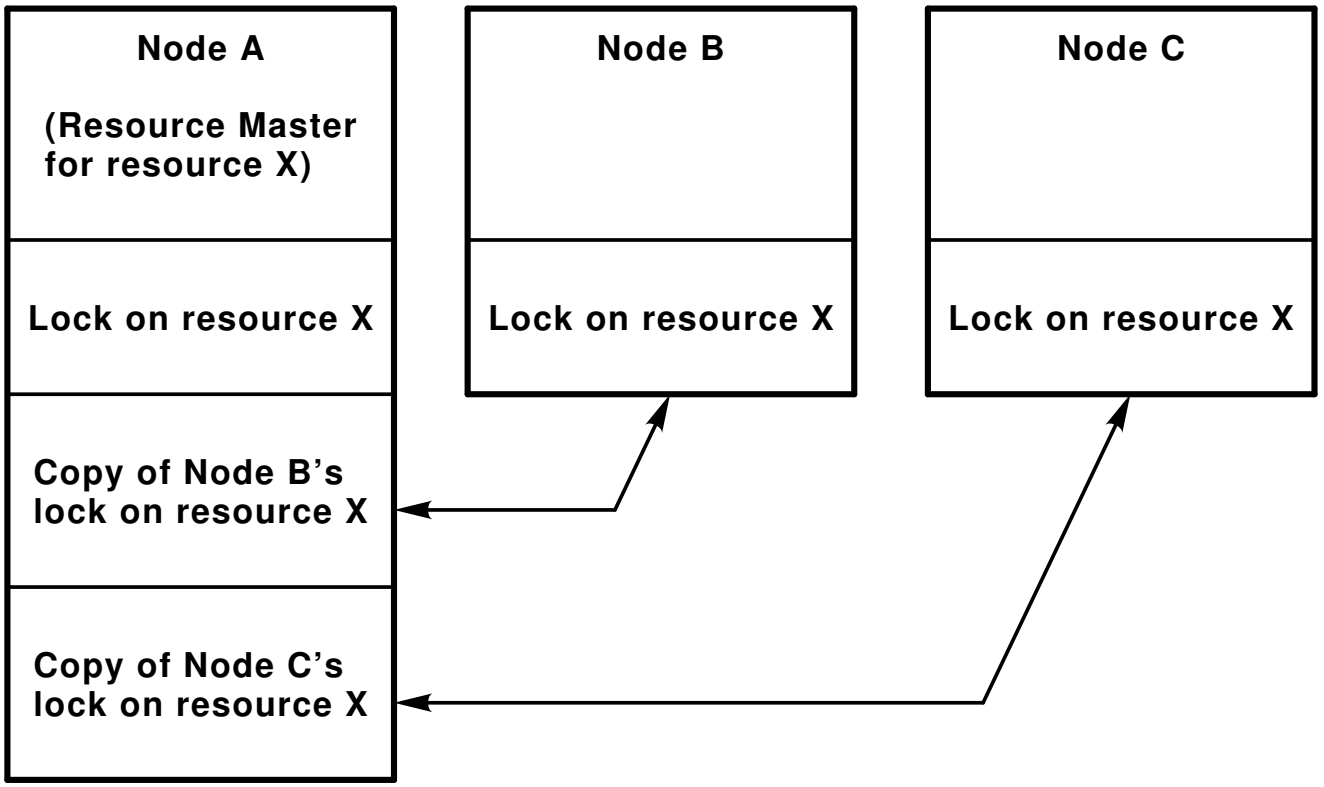
  handles communications between cluster nodes, enforces the rule of total connectivity, and administers the quorum scheme.

If the number of nodes in a cluster remained fixed and constant, there'd be no need for a state transition.

But systems join and leave the cluster over time as nodes boot up and shut down or fail. Since we want the cluster to be able to continue working despite these changes, we have to accommodate them.

Distributed lock manager

Figure 2: Distributed locks



The design of the Lock Manager needed to be robust, able to survive node failures, yet perform well under steady-state conditions.

The method chosen has a single node, called the *resource master* to arbitrate lock requests on a given resource.

One needs to be able to deal with a failure of the resource master. Possible ways would be to:

1. Replicate all the information on all nodes
2. Store the information on non-volatile storage (i.e. on disk or in NVRAM)
3. Maintain enough information to reconstruct what a resource master needs to know

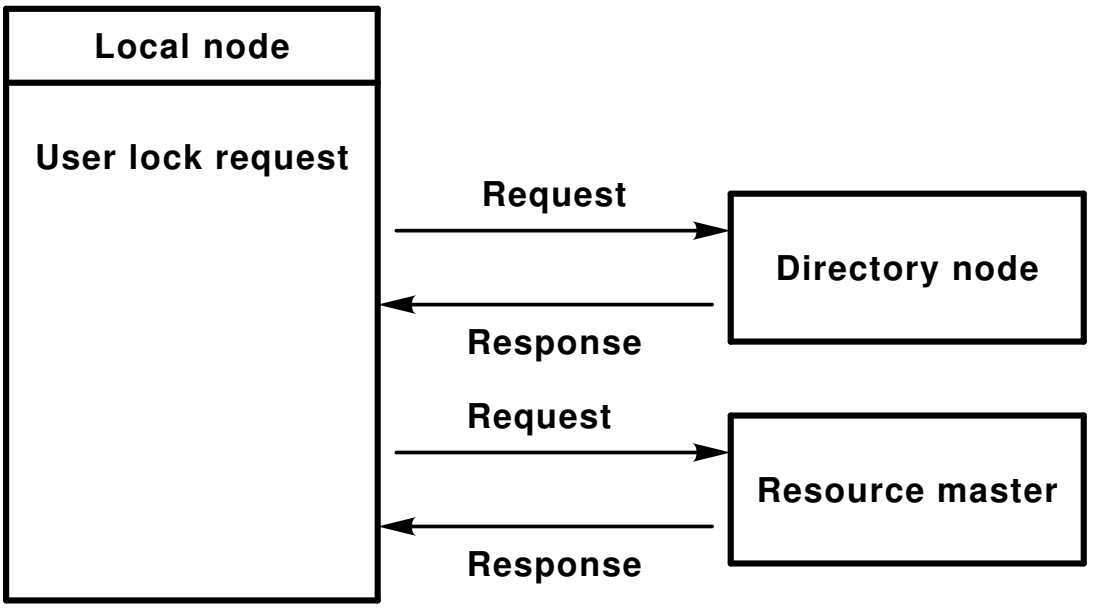
The last option was chosen. Each node keeps track of the locks it holds. If the resource master fails, the nodes left select a new master, and give it the information they have.

Having one node designated as the resource master is efficient, because we only have to make the lock request to that one node when we need a lock, since it knows about all the locks on a given resource in the whole cluster.

Processes on the resource master node will get quick responses to lock requests, since no other node is involved. Because of this, the VMScluster software (as of version 5.5) will switch the resource mastership around over time as needed to ensure that the node with the most need for locks on a given resource is made the resource master.

Distributed lock manager: Directory lookup

Figure 3: Directory lookup



If we're requesting a lock on a resource for the first time, we have to find out which node is the resource master. If we asked every node, this would take much too long. If we designated a single node to keep track of this, that node might get overloaded. Instead, so that the distributed lock manager will scale well, we use a distributed directory lookup scheme to spread this work over a number of designated nodes.

The node requesting the lock asks the lock directory node about the resource. Three possible answers come back:

Possible responses to directory lookup:

1. Request granted (directory node is also resource master)

   I happen to be the resource master for that lock: here's the answer to your request.

2. Become master node (no other locks on this resource)

   No other node has any interest in that resource right now, so it says, in effect, "Tag, you're it. You're now the resource master."

3. Node *xxx* is the resource master

   Node such-and-such is the resource master; go ask that node for the lock.

Only in the last of these three cases must we go make an additional lock request to the resource master node.

Also, once we know which node is the resource master, we remember that, so we don't have to go ask the directory node every time we need another lock.

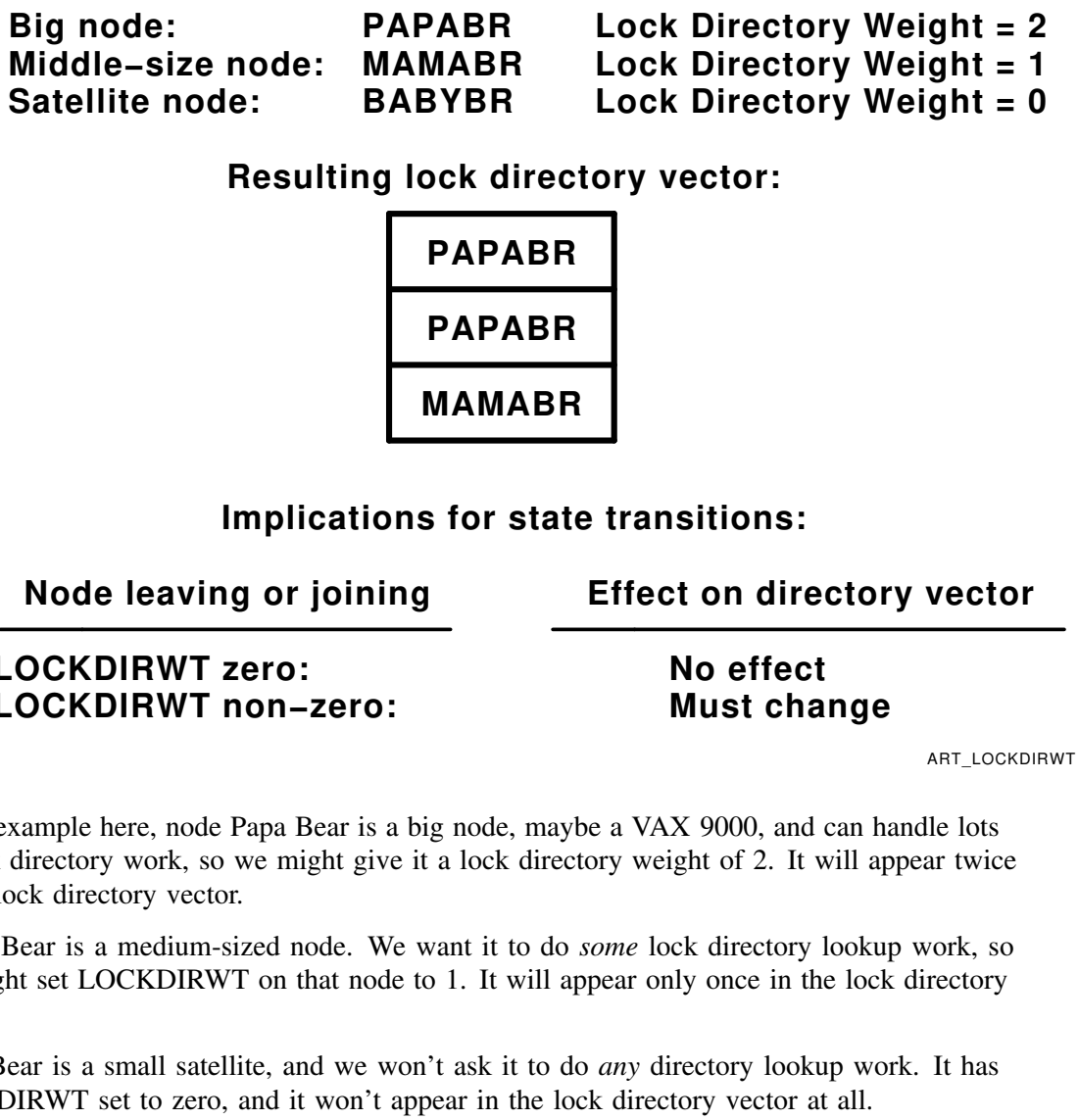
Distributed lock manager: Lock directory vector

Designation of which nodes will do directory lookup work is controlled by the value of the SYSGEN parameter LOCKDIRWT.

The least powerful nodes are given a value of zero for their lock directory weight, and don't have to handle directory lookup work. The more powerful nodes are given non-zero values. Larger values may be given to faster CPUs.

A table is kept on each node in the cluster. It has each node's identification in it, repeated a number of times equal to that node's LOCKDIRWT value.

Figure 4: Lock directory weights and resulting vector



In the example here, node Papa Bear is a big node, maybe a VAX 9000, and can handle lots of lock directory work, so we might give it a lock directory weight of 2. It will appear twice in the lock directory vector.

Mama Bear is a medium-sized node. We want it to do *some* lock directory lookup work, so we might set LOCKDIRWT on that node to 1. It will appear only once in the lock directory vector.

Baby Bear is a small satellite, and we won't ask it to do *any* directory lookup work. It has LOCKDIRWT set to zero, and it won't appear in the lock directory vector at all.

When we first need a lock on a resource, we run the resource name through a hashing algorithm to get a small index value. This index is used to pick out the identification of one of the nodes in the lock directory vector. Since more-powerful nodes typically appear more times in the table, they have a greater chance of handling lock directory lookups.

Since an identical copy of this table is kept on all nodes, and they all use the same hashing algorithm, they will always agree on which node will be the directory node for a given resource.

Now the implications of this on state transitions are:

- Nodes with zero LOCKDIRWT leaving or joining don't affect the lock directory vector
- Nodes with non-zero LOCKDIRWT leaving or joining force us to change the lock directory vector

Distributed lock manager: Lock rebuild types

When nodes leave or join the cluster, the lock database must be modified to reflect the new configuration. This is called a lock rebuild.

When a resource master leaves the cluster, we must choose a new master and tell it about all the existing locks on that resource. Finally, we must update the directory information to point to the new resource master.

In addition, resource masters that remain in the cluster need to clean up after the node which left by removing any locks it held.

We've seen that when a node which joins or leaves the cluster has a non-zero value for LOCKDIRWT, we have to rebuild the directory vector on all nodes. Rebuilding the directory vector means we have to update the directory information on all the directory nodes.

While we rebuild the directory and lock information, we may have to temporarily stall lock requests, until we can straighten things out. This can cause a short, yet often noticeable pause at the user application level. (Note that I/O isn't stalled; but since locks are used to coordinate most I/O operations, the effect will be to eventually hold off on I/O requests.)

The amount of lock directory rebuild work required depends on the circumstances. We classify lock rebuilds into four types based on their severity.

1. Merge:

   Enter already-held lock tree into directory on directory node. (Lock requests not stalled)

   A node joining the cluster only holds one lock tree, a node-specific one associated with its own system ID. All we really need to do is notify the directory node that we're the master for this tree. Since there are no possible access conflicts, we don't stall lock requests while we do this operation, and thus we avoid a pause at user level.

2. Partial:

   Release locks held by departed node, and remaster any locks it was resource master for.

3. Directory:

   Do what a Partial rebuild does, and also rebuild the directory information.

4. Full:

   Throw away all master copies of locks, select new resource masters, and then re-acquire locks, rebuilding the directory information in the process.

   The full rebuild is not of much interest here, as it now only occurs when a cluster is first formed.

When do we use these different rebuild types?

Node's LOCKDIRWT	State change	Type of rebuild
zero	Node joining	Merge
zero	Node leaving	Partial
non-zero	Node joining or leaving	Directory

n/a	Cluster formation	Full

Distributed Lock Manager: Recent changes

Recent changes to lock manager state transition behavior:

- V5.0
  - — Partial and merge rebuilds
- V5.2
  - — Dynamic remastering

    not just in conjunction with a state transition. The algorithm for doing the shift was relatively simple at that point.
- V5.4
  - — Remastering during shutdown

    Instead of waiting until after a node leaves to find another resource master node to take its place, we now take care of this work before the node leaves, if it is shut down normally.

    This leaves less work to be done in conjunction with the state transition.
  - — Multi-threaded lock rebuild

    Some of the lock rebuild work can now be done in parallel, so it can finish faster.
- V5.5
  - — Load-based remastering

    At this point, the full-fledged algorithm for load balancing is implemented.

    The node with the most need for a lock will now tend to be chosen as the resource master, and which node is the resource master node can change over time as the workload within the cluster changes.

State transition triggers

Now that we know what needs to be done with locks in conjunction with a state transition, how does the system know when to initiate one?

A state transition is needed when:

Cluster membership changes, that is, when:

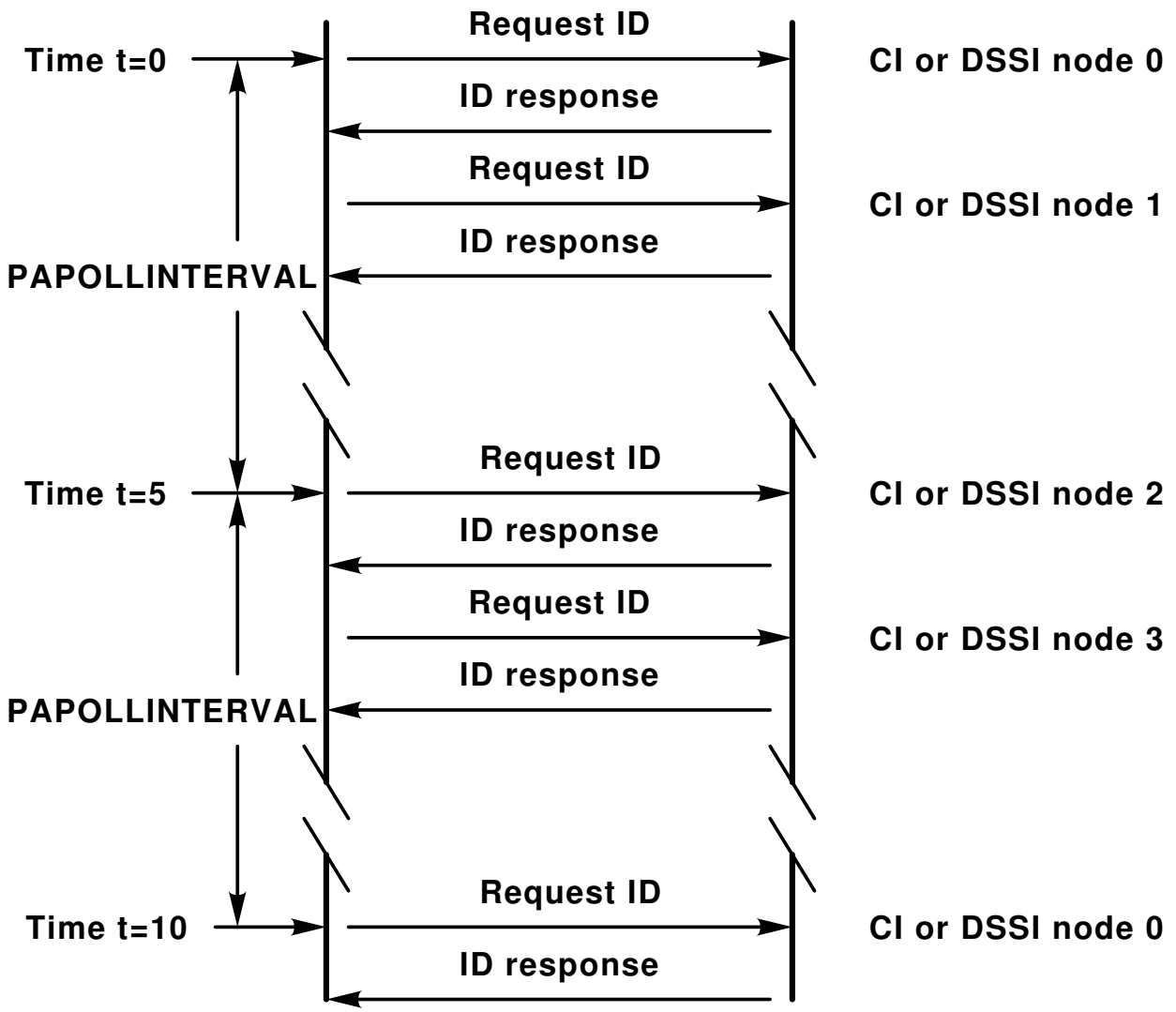
- A new member is added to the cluster
- An existing member is removed from the cluster

How do we recognize these events?

New member detection: CI/DSSI polling

```
PANUMPOLL = 2
PAMAXPORT = 3
PAPOLLINTERVAL = 5 seconds
```

Figure 5: Polling on CI and DSSI



On CI and DSSI, periodic polling of each possible node number is used to detect a new node on the interconnect.

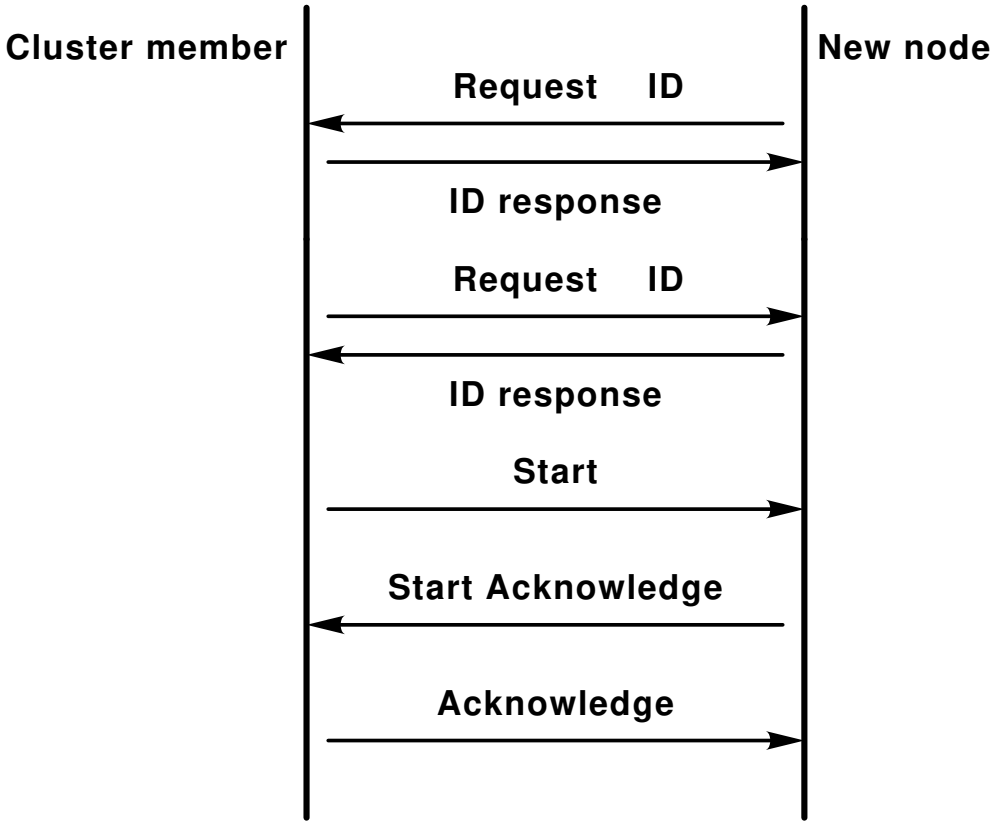
Every PAPOLLINTERVAL seconds, we poll PANUMPOLL nodes, starting at zero and going up as high as PAMAXPORT or the actual maximum node number the interconnect supports (7 for DSSI, 15 or 31 for CI).

In this example, PANUMPOLL is 2, so 2 nodes will be polled each interval. PAMAXPORT is 3, so only nodes 0 through 3 are polled. PAPOLLINTERVAL is 5 seconds.

At the first interval, nodes 0 and 1 are polled. At the next interval, nodes 2 and 3 are polled. The following interval, we start over again with node 0.

New member detection: CI/DSSI virtual circuit formation

Figure 6: Virtual circuit formation on CI and DSSI



When a node comes up on CI or DSSI, the polling mechanism allows it and the existing nodes to detect each other.

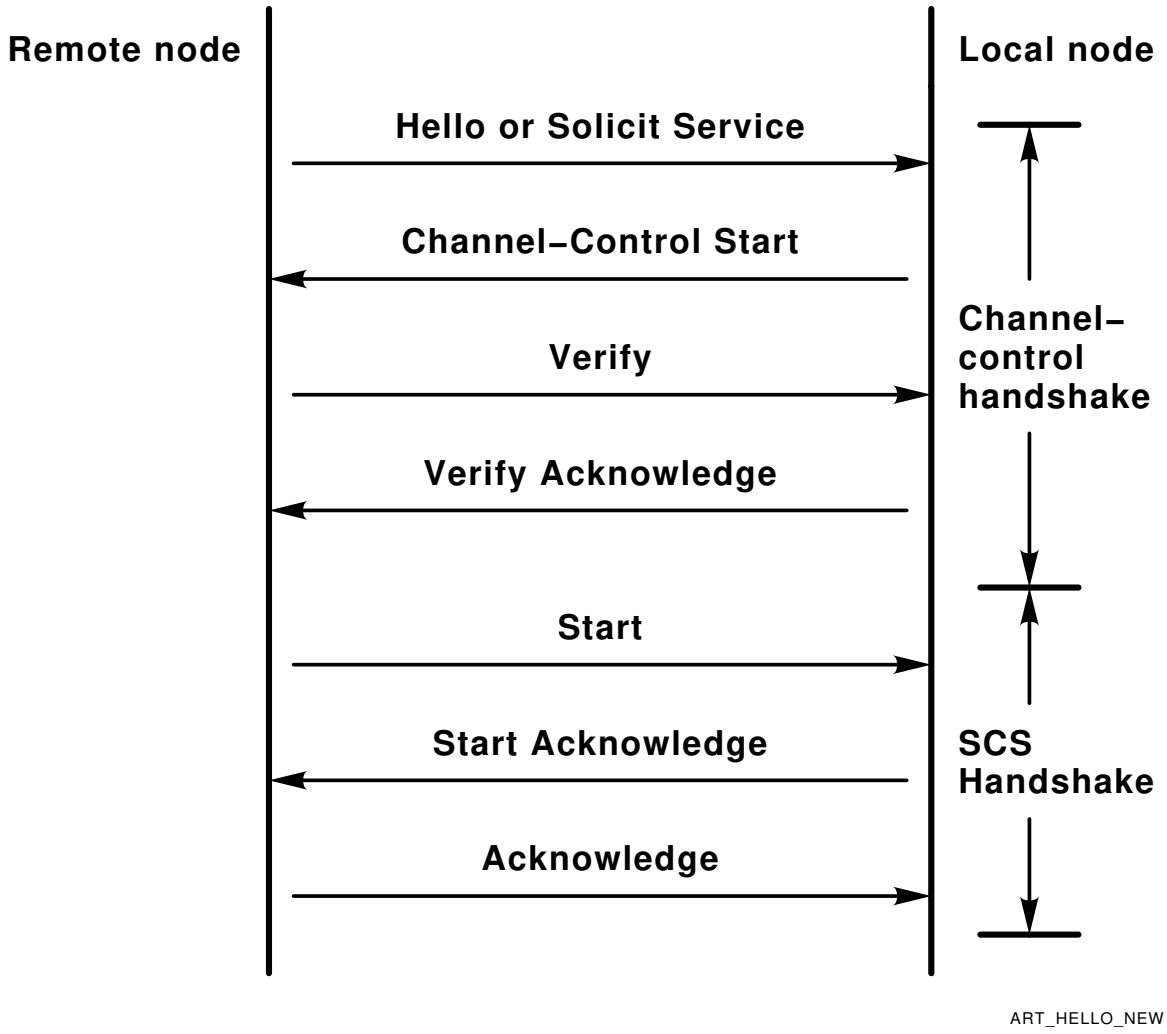
When an ID response is received from a new node, we send a START datagram, the node responds with a START ACKNOWLEDGE datagram, and we ACKNOWLEDGE receipt of that. At this point, we have opened a virtual circuit with the new node.

Next, SCS software polling allows the connection manager to discover the connection manager on the new node.

Once a connection manager is aware of a connection manager on a new node, it forms a connection and begins the process of adding the node to the cluster.

New member detection: Ethernet/FDDI virtual circuit formation

Figure 7: Virtual circuit formation on Ethernet and FDDI



On Ethernet and FDDI, each node sends out a multicast Hello packet periodically to all other nodes with the same cluster group number.

A booting satellite sends out a Solicit Service packet, looking for a system which can MSCP-serve its system disk to it.

When we get a Hello packet or a Solicit Service packet from a new node, we initiate a channel-control handshake. We send a Channel-Control START packet, the node responds with a VERIFY packet, and we send back a VERIFY ACKNOWLEDGE packet. At this point, we have an open channel between the nodes, and each node knows that the other exists.

The SCS software then performs its handshake, just like that which occurs when CI polling discovers a new node. A virtual circuit is formed between the nodes as a result of the SCS handshake, and, as before, the connection manager can now begin its work.

Failures

In addition to new member detection, loss of contact with an existing cluster member can also trigger a state transition.

Types of failures:

- Node failure
- Adapter failure
- Interconnect failure

From the fact that we can't communicate with a node, we can't really determine whether the node went down, the interconnect had a failure, or the local adapter simply stopped working. From the viewpoint of sitting inside the machine, it's hard to tell exactly what's going on.

As a system manager, it's much easier. When one of the systems has smoke pouring out the back, you probably have a node failure. When you see a blue flash and sparks jumping from the Ethernet cable, followed by a loud clap of thunder, you've just had an interconnect failure, and probably a number of adapter failures as well.

When you're talking on the telephone, and in the middle of the conversation you don't get a response, you don't know if the person on the other end is just distracted for a moment, or if you got disconnected. You may say "Hello? Hello?" a few times, before giving up, hanging up, and redialing.

Here are some specific examples of more common types of failures.

Failure examples:

- Thinwire station insertion (loss of a segment temporarily)

  Somebody may decide to add a PC to the thinwire Ethernet segment in their office—a segment which may have 20 cluster satellites on it, too.

- Bridge loss and spanning-tree reconfiguration time

  If a bridge has a failure, the other bridges take a period of time to figure out who's left and how they will form a new spanning tree to cover the LAN.

  {clarify timings}

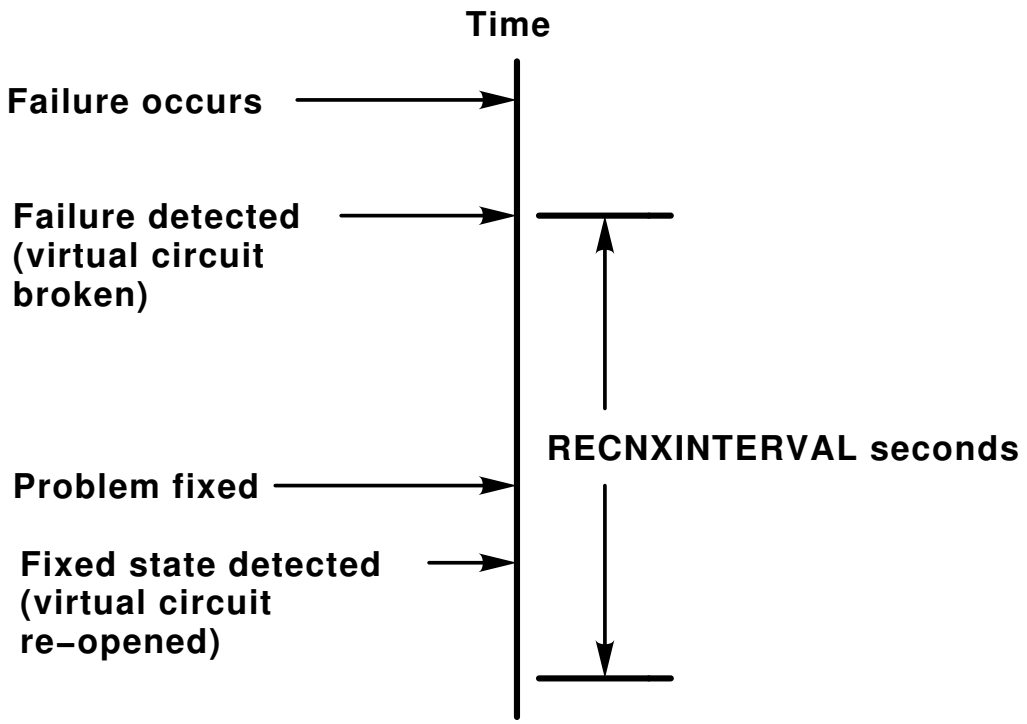
- Power failure for network equipment

  In a computer room I worked in recently, some equipment was moved around and an area was cleared out. Sometime in the middle of the night, the cleaning crew came in to clean up the area. In the process a floor buffer got plugged into the same circuit as all the FDDI concentrators and bridges, and tripped a circuit breaker. Given just the location of a socket in the wall, have you ever tried to find out where the electrical panel is located? I guess they just found another socket to finish the buffing, but that didn't help the network.

  The systems were back up by, oh, no later than noon the next day, I'd say.

Repair period: Recovery within repair period

Figure 8: Recovery within RECNXINTERVAL period



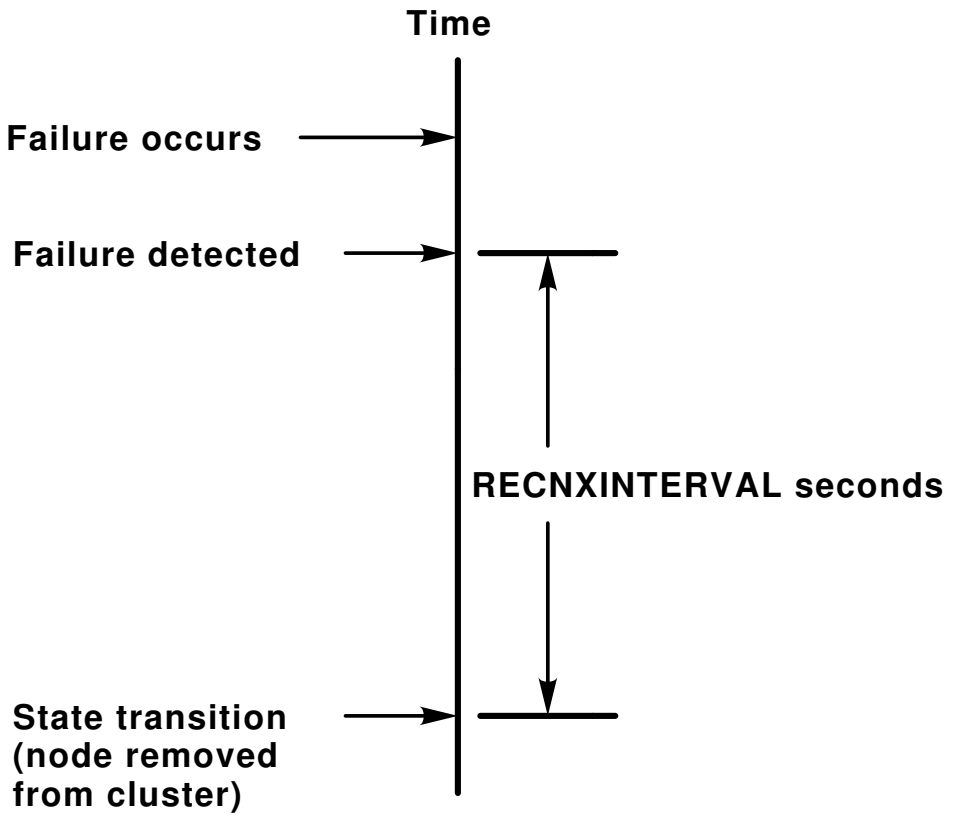
Because outages can be intermittent and temporary, what we commonly call "soft" failures rather than "hard" or long-term failures, and because the effect of removing a node from the cluster is fairly drastic, we wait for a short period after a failure is detected in hopes the problem condition will clear up.

The SYSGEN parameter RECNXINTERVAL provides some control over the degree of resilience the system has to temporary failures, providing a grace period or ride-through period.

If the problem gets cleared up before RECNXINTERVAL seconds passes, we continue as usual.

Repair period: Hard failure

Figure 9: Recovery period



If the problem doesn't go away within RECNXINTERVAL seconds after we detect it, we remove the unreachable node from the cluster.

In choosing a value for this parameter, one has to make a trade-off between

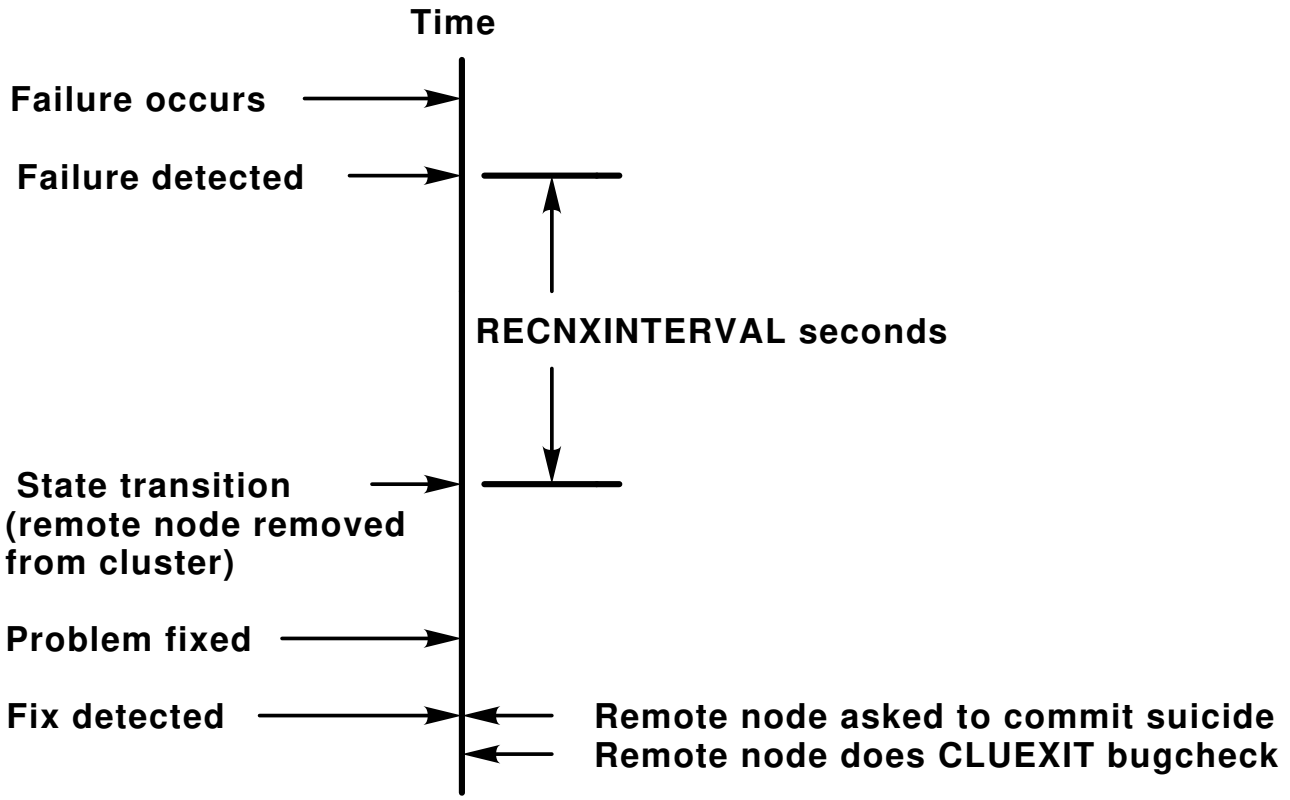
- stability

  (or resilience to problems of short duration), and
- quick reaction time to failures

Note that if you're an unlucky user who happened to request a lock on some shared resource, and a lock on that resource had just been granted to a user on the doomed node just before it crashed, you will have to wait until RECNXINTERVAL seconds passes and the state transition frees the locks from the dead node before you'll get the lock you needed and your work can continue.

Repair period: Late recovery

Figure 10: Recovery after RECNXINTERVAL



Sometimes the problem gets fixed, but too late.

In this case, the node has already been kicked out of the cluster, and any locks it held are now long gone. Folks may have changed files or records that *it* still thinks it has locked. There's no way it can continue from where it left off.

The users shortly get, uh, *notified* of this problem. :-)

When the node attempts to reconnect, it is told to "bug-off", so to speak, so it bugchecks. After it reboots, it may rejoin the cluster as a new member.

Failure detection mechanisms

How do we detect a failure?

Mechanisms to detect a node or communications failure:

- Last-gasp datagram

  When we receive a last-gasp datagram, we can bypass the repair period and its associated delay.

  This is generated during a normal shutdown, so this is the normal case for failure detection.

- Periodic checking
  - — TIMVCFAIL check

    TIMVCFAIL is a new parameter in VMS 5.5.

    It controls how often we check a virtual circuit with another node to make sure it is still working.

  - — Multicast Hello messages for Ethernet and FDDI

    The same Hello packets we use to detect new nodes on Ethernet and FDDI can be used to detect a failure, when we notice that they stop coming.

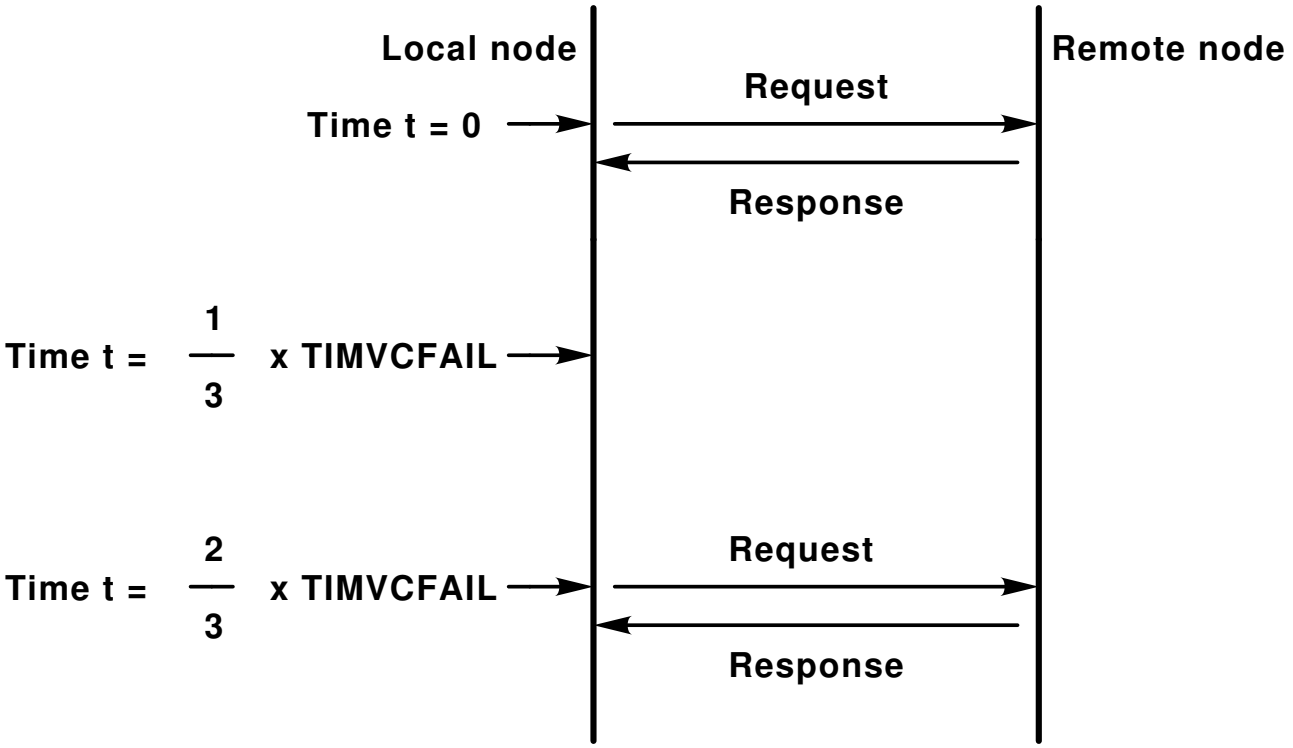
  - — Polling for CI and DSSI

    Depending on the timing of events, the regular polling on CI and DSSI which we discussed earlier in connection with new-member detection sometimes happens to be the first to detect a failure.

First, let's take a look at the new TIMVCFAIL mechanism.

Failure detection mechanisms: TIMVCFAIL check

Figure 11: Virtual-circuit check using TIMVCFAIL



The units for the TIMVCFAIL parameter are hundredths of a second, unlike many other parameters which are in units of whole seconds. The default value is 1600, or 16 seconds.

The parameter specifies the worst-case failure detection time. Three times during the TIMVCFAIL period, we do a check to help us determine if we can talk on each virtual circuit we have established with another node.

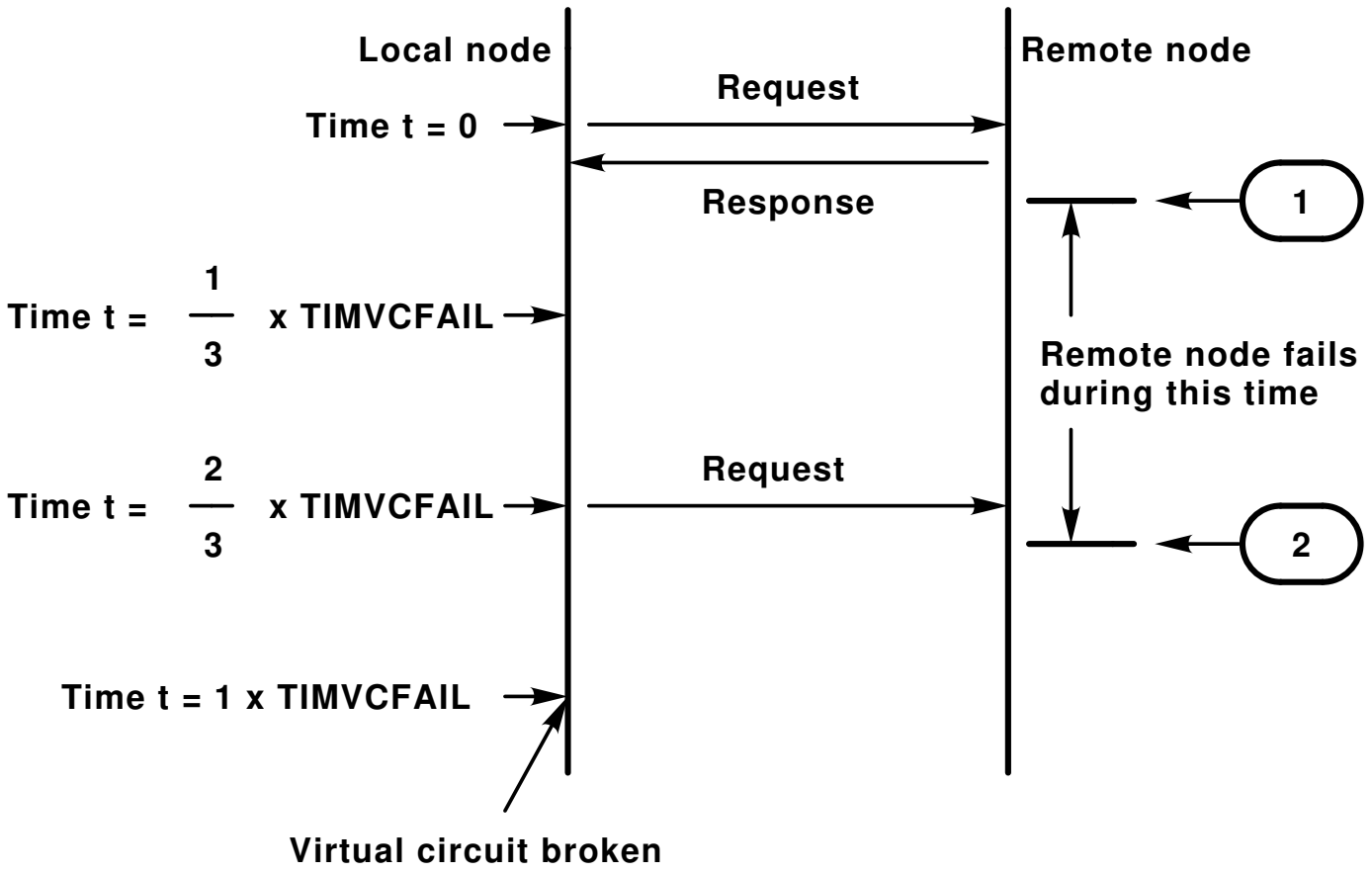
Assume here that we haven't received any System Communications Services (SCS) control messages from a node for a while. (An SCS control message would be something like a request to connect to a specific System Application (SYSAP) such as the VMS MSCP disk or tape server, or a message to adjust SCS credits for flow control.)

Anyway, if we haven't heard from the SCS software on the node for a while, we send out a connection request to a dummy SYSAP, and expect an immediate response back from the SCS software on the other node indicating that the SYSAP we specified wasn't found.

The next time subperiod, we note that we have received an SCS control message (the response to our connection request, in this case) from the other node during the last time subperiod, so we do nothing. The next time period, if we have not received anything we will do a dummy connection request again.

Failure detection mechanisms: TIMVCFAIL failure detection

Figure 12: Failure detection using TIMVCFAIL



ART_TIMVC2

Here we see an example where the remote node fails. The first two time periods proceed as before, but the node fails before it can respond to the next connection request.

At the end of the third subperiod, we notice that we sent out a request at the end of the second subperiod, and have not gotten a response or any other SCS control message yet, so we consider the virtual circuit to be broken.

Now, if we can't recover before RECNXINTERVAL seconds passes, we'll have to kick that node out of the cluster.

Note the best-case and worst-case detection times here:

- If the remote node fails at point (1), just after the response to the first request, it takes us nearly the full period of TIMVCFAIL to report the failure.

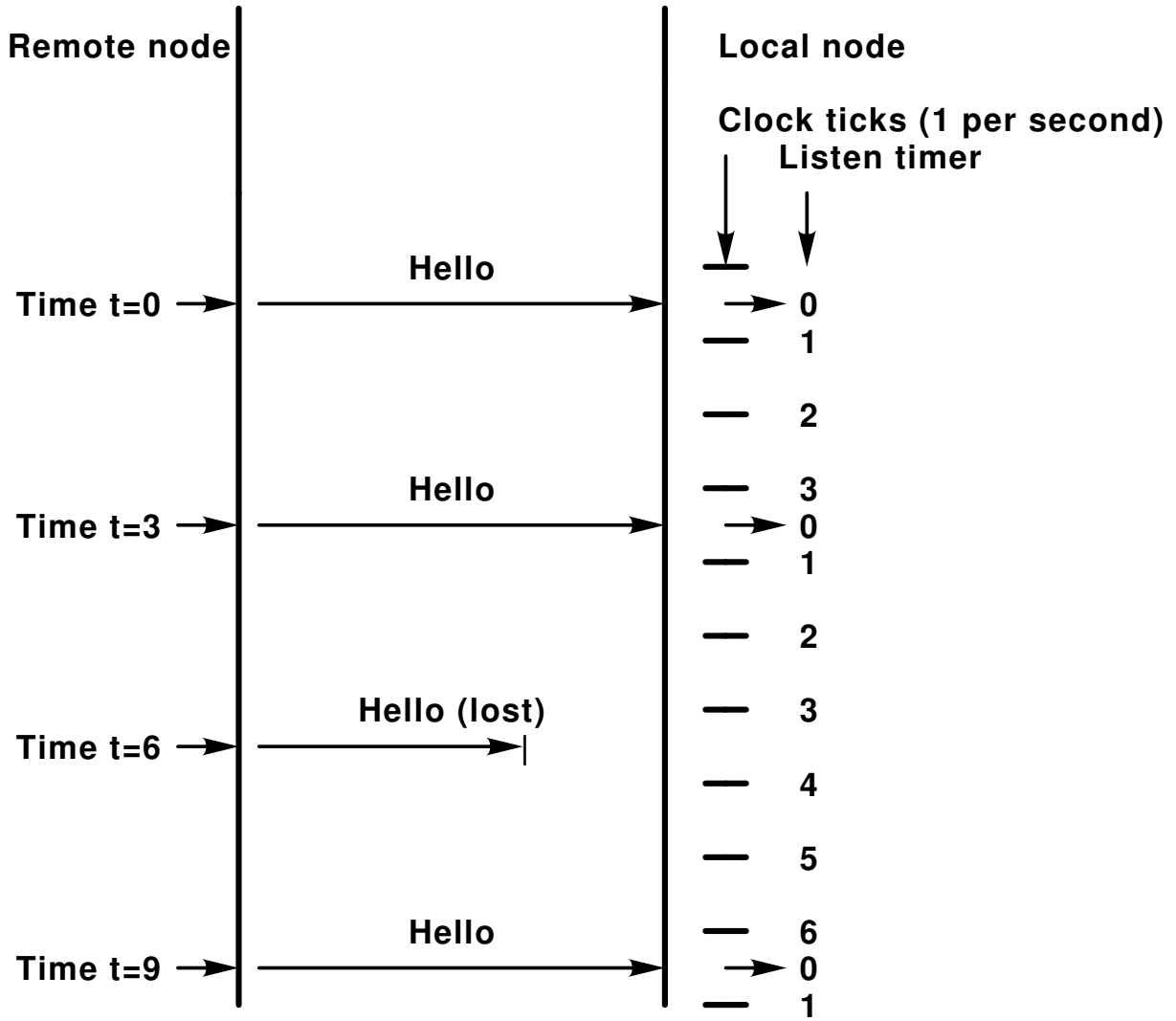
- If the failure occurs at point (2), just before we can send back a response to the second request, it takes about 1/3 of TIMVCFAIL to report the failure.

In any case, the mechanism ensures that we can always detect the failure in no more than one full TIMVCFAIL period.

Next, let's look at the Hello packet mechanism and how it is used to detect failures.

Failure detection mechanisms: Hello packets on LAN

Figure 13: Hello packets on LAN



On Ethernet and FDDI local area networks (LANs), each cluster member sends out a multicast packet every three seconds to all other nodes with the same cluster group number.

Each node keeps a listen timer for each of the other nodes. The listen timer is used to keep track of how long it has been since cluster traffic has been received from that node.

Conceptually, you can think of the listen timer for each remote node as being incremented once a second. The actual implementation uses a ring of queues to minimize the amount of work actually being done each second.

But continuing with our analogy, when a Hello packet or any other SCS-protocol traffic is received from a node, the listen timer is reset to zero.

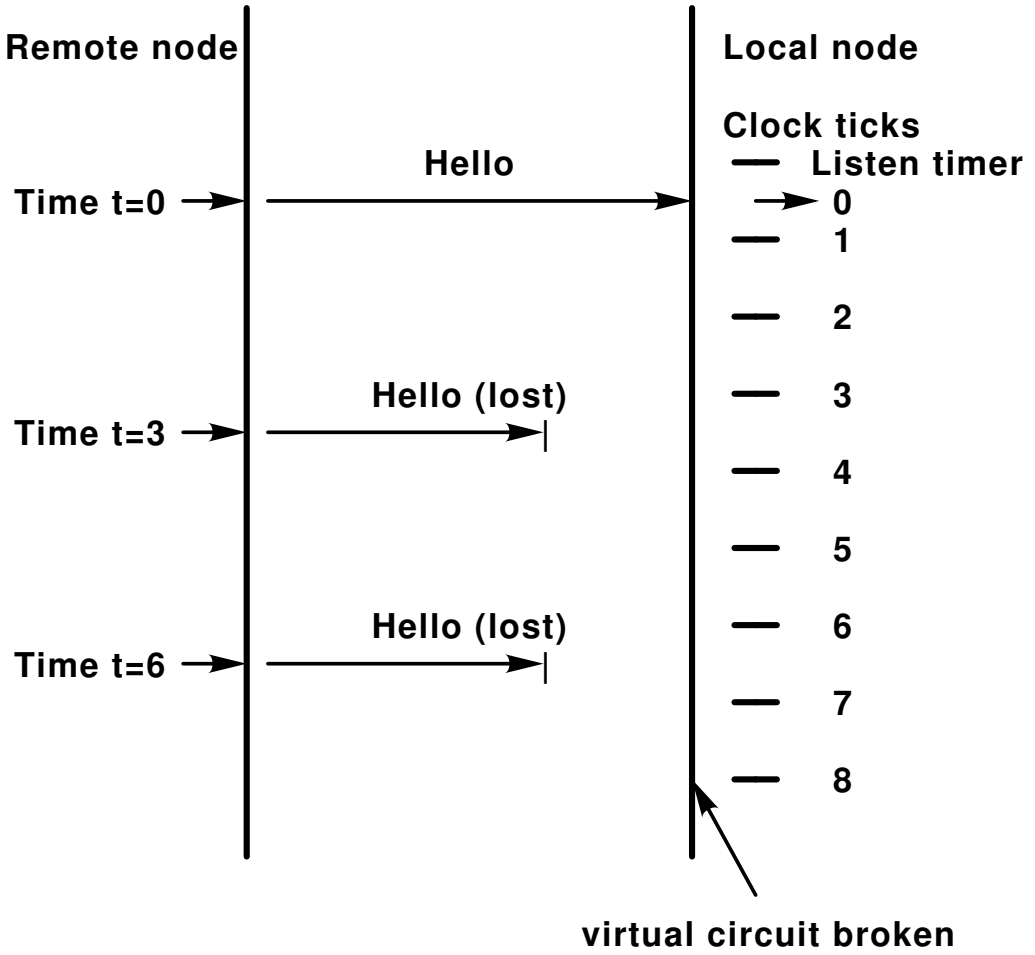
If the listen timer ever reaches 8, indicating that we haven't received any traffic from a node for a period of time somewhere between 8 and 9 seconds, we declare the virtual circuit to have failed.

Missing a single Hello packet won't trigger this, as you can see.

Failure detection mechanisms: Hello packet failure detection

This slide shows what happens if we miss 2 hello packets in a row, or if a node dies all of a sudden.

Figure 14: Hello packets with failure



Note that depending on exactly when the failure occurs, and how much the clocks on the two nodes are out of phase, detection time may take up to 9 seconds.

In versions of VMS prior to 5.4-3, the clock ticks effectively occurred only once every 8 seconds instead of every second, and we waited for 2 to occur before timing out. This resulted in worst-case detection times of up to 16 seconds.

TIMVCFAIL checking is always enabled for CI and DSSI. For Ethernet and FDDI, this is used only if TIMVCFAIL is less than 900 (9 seconds); otherwise the Hello packet mechanism is used in its place.

Avoiding state transitions

Up to now, we've been assuming that a single failure, unrepaired, causes a state transition.

Actually, if the system is configured with some redundancy, the cluster may be able to survive some failures without disruption.

Single point of failure vs. redundancy

- Multi-adapter CI/DSSI or Ethernet/FDDI
- Fail-over between CI/DSSI and Ethernet/FDDI

Multiple adapters or multiple interconnects can provide alternate communications paths should one fail.

Protection:

- Power

  Uninterruptible Power Supply systems can lower the frequency of outages due to power failures.

  This is relatively easy in data center applications, but some users go even farther and buy UPS systems for every workstation and satellite node in the cluster, since they're spread out beyond the protection of the UPS for the computer room. A small UPS systems only costs a few hundred dollars, compared with the thousands a workstation costs, so it may be a small price to pay for increased availability.

- Network

  Putting policies in place to control who can touch your network, and isolating your portion of the network using bridges and such, can help prevent outages of the LAN.

{Before showing the next slide, ask this question:}

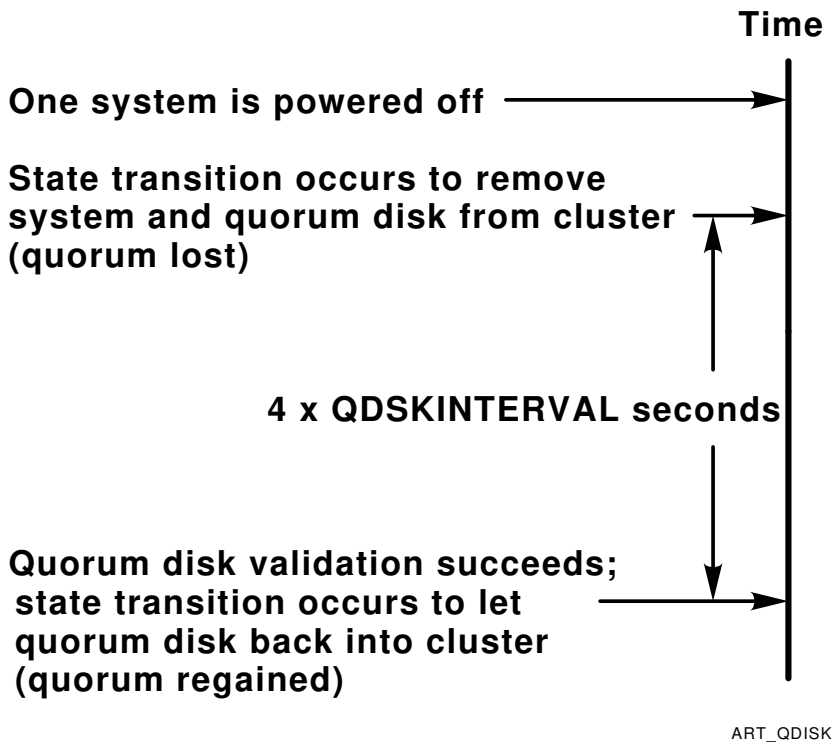
Here's a question for you: (ask for a show of hands)

Q: Let's say you have a 3-node cluster, and one node gets powered off abruptly. Do state transitions take longer if you have a quorum disk, or without a quorum disk?

{Show the next slide at this point.}

Effects of a state transition at user level: Quorum disk vote loss

Figure 15: Quorum disk revalidation



A: That was a trick question. Actually, the state transitions take about the same amount of time, but that relatively long period in between the two of them when quorum is lost makes the users a little upset.

As you can see, the degree to which a state transition affects users depends on the circumstances.

If quorum is lost as a result of the state transition, the effect will be much greater.

I mentioned a while ago that if a system is shut down normally, it sends out a "last-gasp" datagram. Starting with version 5.3 of VMS, we take advantage of this fact to avoid losing the quorum disk votes when a node goes down.

When this last-gasp datagram is received, the remaining nodes assume it isn't going to be trying to access shared resources, so we don't need to stop accessing the shared resources ourselves while we ensure that it's really gone away, so we continue to count the quorum disk's votes toward cluster quorum.

Prior to VMS version 5.3, we stopped counting the quorum disk's votes regardless of whether the node left the cluster normally or not. This means that under normal conditions now, loss of quorum is less frequent than before for configurations using a quorum disk.

A quorum disk is usually only used in the case of a cluster with only 2 voting nodes, although there are valid reasons why someone might want to use a quorum disk with more than 2 voting nodes.

You may want to consider using a *quorum VAX* instead of a quorum disk.

A quorum VAX is just another VAX system, booting independently (not as a satellite node), which supplies a tie-breaking vote and allows the system to continue despite the loss of one of the voting nodes.

{Anecdote:

I got a call from some folks who were building a cluster with a couple of VAX 9000s. They needed to meet a 10-second specification for application fail-over time. When they powered one machine off (which prevents the last-gasp message from getting out), they were seeing fail-over times close to a minute long.

The first thing I told them was to get rid of the quorum disk and use a quorum VAX instead, and explained what that meant. I talked with them a couple of days later and asked whether they'd gotten a quorum VAX. They said "Oh yes; we had another VAX 9000 just sitting there so we cabled that in." I said, "Well, that'll probably do the job, all right."

Effects of a state transition: Sequence of events

Let's look at the sequence of events during a state transition.

State transition tasks

1. Determine new cluster configuration.

   Quorum lost:

   - QUORUM capability bit removed from all CPUs

     If quorum is lost, we need to avoid I/O activity which would be uncoordinated while we don't have quorum. This is because the systems without quorum might just not be able to talk with the majority of systems, and the majority would have quorum and would continue to modify shared resources.

     To prevent uncoordinated activity, by default, all processes require a capability bit called QUORUM to be present on a CPU before they can be scheduled to execute. When quorum is lost, the QUORUM capability bit is removed from all the CPUs, so no processes can be scheduled to run.

     The visible symptom of this is that the processes show up in RWCAP (Resource Wait for Capability) state if you look at a crash dump of a system which is hung for lack of quorum.

     In the past, this was done by simply looping at IPL 4 during quorum loss; which prevented process activity, which occurs at IPL 0 and 2.

     Note duration of quorum loss with quorum disk vs. quorum VAX

   - Disks put into mount verification

     to protect the data

   Quorum not lost:

2. Rebuild lock database

   - Stall lock requests

     We hold off on new lock requests while we're busy dealing with the lock rebuild

   - I/O synchronization

     To ensure that all I/Os issued under the old set of locks completes before any I/Os which will come after the state transition, despite the efforts of certain controllers which may be tempted to re-order them for the sake of performance, we issue a command to MSCP controllers telling them to effectively put up a barrier between past and future I/Os so the old I/Os will all finish first.

   - Do rebuild work

     Here we do the actual rebuild work.

     Some of this stuff can be done in parallel.

   - Resume lock handling

     Once the rebuild is done, the backlog of lock requests can be handled.

Related issues

There are some things that are commonly associated with state transitions, but which are not actually part of state transitions themselves.

- Quorum disk validation
  - 4 x QDSKINTERVAL seconds

    (vs. x 2 sometime before ??; quorum disk watcher effect ??)
  - Quorum VAX as alternative
  - Last-gasp datagram effects
- Disk mount verification on quorum loss

  (to protect data) resulting in more I/O activity
- Node reboot activity
  - Startup work on rebooting big nodes

    When a node crashes, there's a state transition to kick it out. When it reboots, there's another to let it back in. During the reboot, response times for shared devices like the system disk may degrade, and this may affect users on other nodes.
  - Satellites rebooting
  - Disk rebuilds
    * Set ACP_REBLDSYSD = 0
    * Use MOUNT/NOREBUILD
    * Do SET VOLUME/REBUILD in a batch job late at night
- Shadowing

  { (???) Here I'm thinking about:
  - Merge copies initiated by a node crashing, or mini-merges
  - Full copies caused by the reboot of a node which serves a local disk as an additional shadow member (?)}
- Application recovery work
  - Users dumped off a failed node log back in to another node
  - Replaying journal files

Measuring state transition effects

Starting with VMS version 5.4, you can find out the type of the last lock rebuild, and how long locking had to be stalled in conjunction with a state transition.

These SDA commands show how to retrieve this information: Determine the type of the last lock rebuild:

```
$ analyze/system
SDA> read sys$system:scsdef
SDA> evaluate @(@clu$gl_club + club$b_newrbld_req) & FF
Hex = 00000002   Decimal = 2  ACP$V_SWAPPRV
```

Rebuild types:

1. Merge (locking not disabled)
2. Partial
3. Directory
4. Full

Determine the duration of the last lock request stall period:

```
SDA> define toff = @(@clu$gl_club + club$l_toff)
SDA> define ton = @(@clu$gl_club + club$l_ton)
SDA> evaluate ton-toff
Hex = 0000026B   Decimal = 619  PDT$Q_COMQH+00003
```

The duration time value you see is in units of hundredths of seconds, so divide it by 100 to get the number of seconds.

In the example here, the last rebuild was of type 2, or a partial rebuild, and locking was blocked for 6.19 seconds.

Note that if the rebuild type is 1 (indicating a merge rebuild), locking is not disabled, so a duration time is not applicable in that case.

```
{Any tools to measure other aspects of activity related to state transitions,
such as duration of a quorum loss or I/O recovery times (?)}

{Do we need to note here that an unlucky user who's waiting for a lock held by
a node which just died will wait for longer than the blocked-locking time?}
```

Minimizing duration of state transitions

So if you want to configure a system to minimize the impact of state transitions, how would you do it?

Configuration issues:

- Few (e.g. 3) nodes

  You'd have very few nodes, like exactly 3. {Those with LOCKDIRWT != 0 closely matched in CPU power for predictability (?)}

- Quorum VAX; no quorum disk

  You'd use a quorum VAX instead of a quorum disk (and this implies a minimum of 3 CPU nodes for quorum purposes).

- Isolate cluster interconnects from other traffic

  so the chance of outside factors causing trouble are minimized (i.e. isolate LAVC Ethernet segment behind a bridge)

  Consider not enabling Ethernet for cluster communications at all (by setting NISCS_LOAD_PEA0 to 0), but using CI or DSSI only.

Operational issues:

- Avoid disk rebuilds on reboot
- Let Last-Gasp work

  Avoid a Control-P/Halt or a system power-off as the shutdown method; crash the system instead (so the last-gasp message gets sent).

Minimizing duration of state transitions: System parameters

- V5.4 or earlier:
  - — PAPOLLINTERVAL = 1

    (vs. default of 5)
  - — PAMAXPORT = 31

    (or actual maximum CI/DSSI node number, no big deal) (vs. default of 32)

    (VMS checks the actual port, and cranks this down for DSSI (to 7) or for CI ports set in 16-node star coupler mode rather than 32-node mode.)
  - — PANUMPOLL = 32

    (or PAMAXPORT + 1) (vs. default of 16)
  - — PANOPOLL = 0 (leave polling enabled)

    Don't disable polling (Default is 0)
  - — PASANITY = 1 (leave sanity timer enabled)

    Don't disable sanity timer (default is 1)
  - — PASTIMOUT = 1

    Sanity timer of 1 second (vs. default of 5)
  - — RECNXINTERVAL = 1

    (note lower bound enforced by VMS (What is it???)) (vs. default of 20)
  - — NISCS_LOAD_PEA0 = 0

    Don't use Ethernet or FDDI here, because when the CI or DSSI quickly detects a failure, the cluster will then fail over to the Ethernet or FDDI, where we haven't yet detected the failure, and we'll end up waiting longer.
- V5.5 or later:
  - — TIMVCFAIL as low as 100 (1 second), as needed

    (vs. default of 1600, or 16 seconds)

    Note that there's a tradeoff between stability and quick detection here.

    Anecdote:

    Those VAX 9000 folks I told you about got their state transition times down quite a bit using a quorum VAX instead of a quorum disk. The state transition itself got down to around 8 seconds, but the application fail-over took more than the 2 seconds that was left in their budget. What they needed was faster failure detection and fail-over, so I suggested they try adjusting the TIMVCFAIL parameter.

    The fellow I talked to figured he might as well go all the way and crank it down to 100, or 1 second. The next time I talked with him, he said, "Expletive deleted! That thing's got a hair trigger on it, don't it?"

    A little experimentation will let you determine the proper trade-off between

    - quick detection and fail-over times

      and

    - resilience to ride through short, intermittent outages

— RECNXINTERVAL as low as 1 second

  (vs. default of 20)

  VMS won't let this go below TIMVCFAIL in actual operation; if you set it lower, the equivalent of TIMVCFAIL (but in units of seconds) is used instead.

— NISCS_LOAD_PEA0 can be 0 or 1

  Here, you can use Ethernet or FDDI if you wish, because the TIMVCFAIL mechanism will replace the Hello packet mechanism if you set TIMVCFAIL to less than 9 seconds, so the detection times can be similar on all the various interconnect types.

Other parameters of possible interest (???)

- LOCKDIRWT
- MVTIMEOUT (?)
- HBVS-shadowing (?) (Shadow member timeout?)

Questions?

Are there any questions?

Notes to presenter:

Here are some possible questions and answers you might encounter.

Q: Following your recommendations, how short a cluster state transition can be guaranteed?

A: We feel that 10 seconds total, where 5 seconds are allocated for cluster fail-over and 5 seconds are allocated for application fail-over, is easily achievable.

{Beyond that, more rigorous testing would be required, or something like a VAXft system might be a better choice.}

Q:

A:

Q:

A: