Designing a Linux Cluster

Stephen Tweedie

Contents

CHAPTER 1

Overview of Thes Documents

This set of documents is not a comprehensive design spec for clustering. Rather, it is a set of miscellaneous documents including both discussion documents and work-in-progress design drafts, not for a whole clustering system, but for a core set of APIs intended to provide a comprehensive and robust infrastructure on top of which true clustering services can be layered.

So, you won't find any proposals for IP takeover or for clustered filesystems here. You _will_ find proposals for APIs which will let the IP failover manager communicate the state of the running IP interfaces to the rest of the cluster, or to allow other services to be started and stopped as appropriate if an IP address is migrated from one node to another.

The motivation for this work is primarily that although there are many distinct clustering projects under way for Linux, there is no general-purpose framework to provide solutions for some of the hard problems such as quorum management, massive scalability and management frameworks. A successful outcome would be a set of core cluster APIs which are both simple enough that arbitrary other (existing or future) cluster services can take advantage of them easily; and powerful enough that there is real benefit to be had from using them.

In this directory you will find the following documents: [Editor note: the documents are printed here as chapters 2-13]

goals.txt: Outlines in a little more detail some of the goals of the work envisaged.

principles.txt: A few design principles and justifications applicable to the whole project.

structure.txt: Describes the component layers necessary to achieve the initial objectives.

hierarchy.txt: Describes some of the implications of having hierarchical clusters.

Then we have documents describing individual components in some detail:

api.txt: General requirements for the cluster APIs, including at least some details of inter-process communication between cluster service processes on a single node.

recovery.txt: A manager for the local cluster processes on a node. This must deal with both the initial startup of processes, and the coordinated restart after a cluster transition.

communications.txt: The cluster communications layer: getting nodes to talk to each other.

integration.txt: The cluster integration layer: binding nodes into a coherent cluster.

discovery.txt: The discovery algorithm used for cluster reforming in the integration layer.

barrier.txt: The barrier API used for cluster-wide synchronisation of arbitrary services.

quorum.txt: Quorum management: how to tell if it is safe to acces shared cluster data.

namespace.txt: The namespace manager. Describes the namespace service and what it is intended to achieve.

CHAPTER 2

Clustering Goals

The primary goals of the cluster infrastructure project are to:

- Provide hooks which allow cluster-wide coordination between arbitrary services. Implementing those services is a separate problem.
- Provide a strong concept of cluster membership, with synchronised, transactional transitions between cluster states when a new node joins the cluster or an existing node dies.

The long term "vision" includes much which is not anticipated in the initial implementation. Some of this will be scheduled for future implementation. We also want the clustering services to accommodate third-party services which may never be part of this cluster core itself, but which will place certain requirements on the clustering which we have to take into account here. We are particularly concerned with:

- Scalability, both up and down. We want to be able to cluster a pair of 486es to provide highly available departmental web services out of the Linux cupboard that a sysadmin installed once. We want to be able to cluster massive number (tens or hundreds of thousands) of nodes for Massively Parallel Processing (note that we only want to deal with the admin/HA aspects of this --- the MPP interconnect is somebody else's problem --- but this still implies a need for hierarchical clusters).

- Scalable cluster filesystem support. Within one cluster segment, GFS is the sort of implementation we want to grow a cluster FS out of, with shared disk access on top of cluster-wide shared block devices and shared software raid.

On top of that, a mechanism such as AFS or CODA for migrating files between cluster segments will be necessary: the shared disk model does not scale well once distant nodes become involved.

- Provide a set of sufficiently simple APIs, without actually enforcing the way they are used, that nobody in their right mind will consider using any other basic cluster infrastructure to base their HA solutions on. (Well, it can’t hurt to dream, can it?)

CHAPTER 3

Cluster Design Principles

Remember the Unix philosophy:

Do one thing, but do it well.

There are two basic principles which pervade this design:

Modularity. Needless to say, modular code is easier to maintain than tightly integrated monolithic structures. The entire design of the clustering is intended to take into account the requirements of each module when considering other modules, but the mechanisms used to implement any specific module are never exposed to other modules directly except through specified, general purpose APIs.

This results in certain design features which are alien to most HA clustering implementations. For example, Quorum is never considered by the cluster integration layer. Quorum is merely another resource which comes and goes in the cluster as nodes join and leave. Obviously it is a critically important resource, and must be the first resource recovered after a cluster transition, but the impact of quorum management on the rest of the cluster layers is minimal.

State progression. It is possible to produce very complex state transition diagram when producing code which operates in the highly concurrent environment in which cluster code is expected to run. There is a guiding design principle which substantially simplifies many of these state transitions:

++ All components of the system which need to construct global (externally influenced) state and which have to deal with error conditions must maintain a strict priority ordering of states. Only after all neighbouring components have acknowledged transition to the same state are we allowed to begin controlled progression to the next state (ie. there is a barrier between each state progression). Error conditions (at least, errors which are expected to trigger cluster state transitions) ALWAYS trigger an immediate abort of the construction of the current state: we move instantly to a state lower in the state hierarchy on error, and resume construction of the higher state from there.

This principle is obeyed in many places. In the cluster communications code, any (unrecovered) communications error between two nodes triggers an immedate loss of link UP status, and we do not allow the link state to come back UP until we are sure that (a) the other endpoint has also left UP state, and (b) all other communication channels between the two nodes have also been purged of messages from the old UP state. In the cluster integration layers, we have various stages we must go through to build the new cluster: discovery, election, verification and commit. Any error in any of these stages triggers an immediate drop to a previous stage.

The important property to obey here is that whenever we fall back to such a lower state, we must have a mechanism in place which ensures that all of our neighbours will also return to that state before continuing: it is necessary to reestablish agreement with our neighbours that that has occurred before we can start to progress the state machine again.

CHAPTER 4

Cluster Structure

The following components form the core of the cluster design:

- Channel layer:point-to-point communications
- Link layer:reliable bound channel sets between node pair
- Integration layer:forms the cluster topology
- Recovery Layer:performs recovery and controlled service startup/ teardown after a cluster transition.

There are also four key services which are central to the basic cluster APIs:

- JDB:stores persistent cluster internal state (and used for the quorum database)
- Quorum Layer:determines who has quorum
- Barrier services:provides cluster-wide synchronisation services
- Namespace service:provides a cluster-wide name/value binding service.

Component Descriptions

Channel layer: The low-level physical communications layer. This layer maintains multiple interfaces (which might be IP, serial or SCSI, for example). Neighbourhood discovery is performed on each interface via broadcast, and after any neighbour is found, a handshake is performed which creates a permanent point-to-point channel to that neighbour over the given interface. The channel supports sequenced data delivery, heartbeat link liveness verification, and controlled reset on error.

Interfaces are entirely independent of each other. If the same host is found on multiple interfaces (ie. we have multiple connections to that host), the connection on interface is maintained independently of the others.

Each low-level channel has a metric which determines how “good” that channel is for carrying cluster traffic. A low-performance link is defined as one with a negative metric: serial lines should have this property, for example.

Link layer: Built on the channel layer, the link layer constructs a higher-level communications mechanism which binds together all available channels to any given host. The link is in one of four states at any time:

DOWN:No connection to remote host is held.

RESET:We have had an error, and are temporarily resetting all of the channels in the link.

DEGRADED:At least one channel is up and running, but its metric is negative.

UP:At least one “good” channel is up and running.

When constructing a new cluster state, the upper layers will use a degraded link to perform a clean cluster transition evicting one of the nodes on that link from the cluster. We can use the degraded link to perform this eviction cleanly, adjusting quorum to take its departure into account.

Integration layer This layer performs transitions of the overall cluster topology, merging neighbouring clusters, evicting dead or misbehaving members and ensuring transactional transitions between cluster topologies.

We have to define very carefully what a “cluster” is in this context. A cluster is an agreement between a set of nodes that all of those nodes can communicate with each other and are able to form a useful working group together. The cluster is not just the set of connected nodes: it is the *agreement* of connectivity. Cluster transitions, such as the joining of a new node into the cluster, are atomic operations in which all nodes agree to the new group topology.These transitions are transactional and atomic.

A key concept here is the cluster ID. The ID is unique to a single “incarnation” of the cluster. Whenever a cluster splits into multiple pieces, any surviving subcluster which has more than half the votes of the old cluster or more retains its cluster ID: all other machines are evicted from the cluster and must obtain a new cluster ID. (As a tie-breaker, a cluster with precisely half the votes retains its cluster ID iff it includes the previous cluster’s “cluster controller” node among its members.)

A new cluster ID is generated whenever a machine first enters the integration layer (either after eviction from a cluster or on initial startup). Such a machine considers itself to form a single-node cluster. As a result, we never have to deal with node joining or leaving clusters: we only ever have to deal with entire clusters joining or splitting from each other.

The cluster ID includes the nodename of the node which generated the ID, and a timestamp. This is done to generate a cluster ID which is unique for all time.

When a cluster partition occurs, at most one of the resulting new smaller clusters will retain the original cluster ID. However, the remaining cluster members are not all left alone to pick up the pieces: although they reenter the integration process as single-node clusters, they will not leave the cluster transition until as many as possible of those nodes have merged into larger clusters, each of which keeps the cluster ID of just one of the subclusters of which it is composed.

Each cluster also has a sequence number which is incremented on each cluster transition, providing applications with an easy way of polling for potential changed cluster state.

The cluster state transition which occurs when a new node joins the cluster or an existing node loses connectivity for any reason is described in integration.txt.

Recovery Manager: This is the layer which integrates the “Core” (for want of a better term --- this describes it as well as anything) of the cluster with the Rest Of The World --- ie. services.

The main job of the transition layer is to “recover” from cluster transitions completed by the integration layer. Recovery constitutes a multitude of operations:

++ Internal recovery of all registered permanent services must be performed. This typically involves reconstructing the global state of that service based on (a) the set of nodes in the new cluster and any differences between that and the previous set (note that care must be taken if we take a transition during this recovery!!), and (b) the state of this service on those nodes. For example, a distributed lock manager might recalculate the hash function used to distribute lock directories over the available nodes, and then reconstruct the lock database based upon the set of locks currently held on each node.

++ Recovery of user services. These services are not necessarily running all the time, and do not necessarily possess internal global state (although they may well rely on external global state, for example the contents of a shared cluster filesystem). On cluster transition, we need to be able to allow user services which have disappeared from the cluster to be reinstantiated on a new node (ie. failover), or to notify already-running services of a change of status (eg. failback after the service's original node returns).

Most importantly, we need to manage the order in which these services are restarted. There is no point in restarting the CGI server until the underlying cluster filesystem has recovered.

The transition layer must therefore know about the dependencies between services. Dependencies may be inferred from the use of Names in the Cluster Namespace: the Namespace can feed dependency information to the transition manager if appropriate. However, the Namespace on its own cannot control recovery: it is the transition manager which allows separate services to coordinate controlled stage-by-stage recovery over the cluster.

One final job of the transition manager is that it must be able to disable access to certain cluster services until recovery is complete (with or without the cooperation of the service concerned). It is quite possible that an application can continue quite blindly over a cluster transition as if nothing had happened (thisis High Availability in action!), but obviously any requests from that application to the cluster filesystem or to the lock manager must be deferred while those services are engaged in recovery.

JDB: A transactional database is required to store local state. Every node with a non-zero vote (a "voting member" of the cluster) is required to have a jdb on local read/write persisitent storage in which the Quorum layer can perform reliable updates to simple configuration and state information. Sleepycat libdb2 (as found GPLed in glibc-2.1) should be quite sufficient.

Quorum Layer: Keeps track of "Quorum", or the Majority Voting Rights, in a cluster.

Quorum is necessary to protect cluster-wide shared persistent state. It is essential to avoid problems when we have “cluster partition”: a possible type of fault in which some of the cluster members have lost communications with the rest, but where the nodes themselves are still working. In a partitioned cluster, we need some mechanism we can rely on to ensure that at most one partition has the right to update the cluster’s shared persistent state. (That state might be a shared disk, for example.)

Quorum is maintained by assigning a number of votes to each node. This is a configuration property of the node. The Quorum manager keeps track of two separate vote counts: the “Cluster Votes”, which is the sum of the votes of every node which is a member of the cluster, and the “Expected Votes”, which is the sum of the vote on every node which has ever been seen by any voting member of the cluster. (The storage of those node records is one reason why the Quorum layer requires a JDB in this design.)

The cluster has Quorum if, and only if, it posesses MORE than half of the Expected Votes. This guarantees that the known nodes which are not in this cluster can not possibly form a Quorum on their own.

There is one other thing we need to do to be completely secure here. If completely new nodes (ie. those never before seen in the cluster) are allowed to join a partition which has no Quorum, we must prevent the new nodes from adding their votes to that partition and disturbing the Expected Votes calculation of other nodes or partitions. To prevent this, a new node is not allowed to vote until it has, at least once, joined a cluster which already has Quorum. This means that during initial cluster configuration, the sysadmin must manually enable Voting Rights on at least one node before the cluster can obtain Quorum for itself.

Barrier services. Barrier services provide a basic synchronisation mechanism for any group of processes in the cluster. A barrier operation involves all the cooperating processes waiting on the same barrier: only when all of them have reached the barrier will any of them be allowed to proceed.

The barrier operation is required extensively by the recovery code for other services, which is what justifies its inclusion as a core cluster service.

Namespace services. The cluster namespace is a non-persistant hierarchial namespace into which any node can register names. The guts of the namespace API includes mechanisms by which processes can not only register names, but also set up dependencies on names.

All failover services will be coordinated through the namespace. If multiple nodes try to register the same name (and if they request exclusive naming), for example, only one will be granted the name. If that node dies, another's request for the name will succeed, and failover will be triggered once the name assignment is complete.

The namespace will also provide a natural mechanism for determining dependencies between services: any process can register a dependency on a name and will receive asynchronous callbacks if the condition of that name changes (either if the name disappears or its ownership changes due to the death of a process holding the name).

As a consequence of the binding of names to services, it becomes easy for any application to find all the services in a cluster or to locate on which node a specific service (such as, for example, a failover-capable printer spool queue) currently resides.

Finally, the namespace aims to make cluster administration simple by providing a simple method by which the admin can query all or selected bound names in a cluster, much as /proc already provides the dynamic information on a single Linux node.

CHAPTER 5

Hierarchichal Clusters

[Read past the definitions to the justifications bit if you just want to get an overview of what we're trying to achieve here.]

Definitions

For the discussion of hierarchical clusters, we need to make a few definitions.

First of all, let us define what a hierarchical cluster is. A single, FLAT cluster is some virtual entity born from a collaboration between (probably nearby) nodes on a network. We can call this a FIRST-LEVEL cluster. Those nodes are the clusters MEMBERs, and together they form the clusters MEMBERSHIP LIST. Each cluster has, at all times, a unique privileged node known as the CLUSTER LEADER. That node does plays a key role in coordinating cluster transitions, but other than that it does not necessarily perform any special work during the normal running of the cluster.

If we want to combine clusters into larger units, we want to be able to support the binding of many clusters into a single HIGHER-LEVEL cluster, or METACLUSTER. That cluster is formed from the clustering together of all of the cluster leader of the cluster's MEMBER CLUSTERS: those cluster leaders, together, form the metacluster's membership list. The metacluster's subcluster list is therefore a different concept from its membership list: if a subcluster undergoes a state transition and elects a new cluster leader, then that new leader will replace the old one in the metacluster's membership list, but the metacluster's subcluster list remains the same. The complete list of all member clusters of a metacluster, and all the members of those member clusters right down to the first-level clusters, forms the metacluster's SUBCLUSTER LIST.

A metacluster of first-level clusters constitutes a second-level cluster. A metacluster of those second-level clusters forms a third-level cluster, and so on.

We can observe that in a large corporate network being managed as a single cluster (we can dream, can't we? :), we really do not want all of our labs' test clusters to cause corporate-wide cluster transitions whenever a test machine dies. Not at the rate at which I reboot test machines, at any rate. This is the "S-Cluster" case.

On the other hand, we can also envisage large compute clusters of tens or hundreds of thousands of nodes in which we want to support high rates of cluster traffic on cluster-wide shared resources (such as cluster-wide locks). We simply *have* to reclaim such locks when a node dies (DLM semantics require it), so in this second, large "L-Cluster", every node is significant as far as cluster-wide recovery is concerned.

This implies that, when looking at any given level of the cluster hierarchy, there may be nodes underneath which participate in recovery, as well as those which do not. We define these as PEER nodes and SATELLITE nodes respectively. For any cluster, all of the peers taking part in that cluster's recovery form the clusters PEER LIST or PEERAGE. Any given node may be a member of the peerage for its own cluster and for any number of higher-level clusters. Above that level --- it's PEER LEVEL --- the node is a satellite. A true satellite in the first-level cluster has a peer level of zero.

*** MISSING: Define exactly what a satellite's relation with the higher level clusters is. In particular, does a satellite always proxy through peers in its own cluster? There may not be any such nodes. The satellite may _have_ to go to some outside node in a higher level cluster.

*** Think also about what happens if a node is a peer only up to a certain level: it loses the ability to be a cluster leader above that level. Does this affect proxying much? I need to draw this out before expanding on this much, and before completing any more detail on the recovery specs.

This has one major implication for the design of the integration layer. The fact that only cluster leaders can be members of a higher level cluster implies that only nodes with the highest peer level in any cluster may be cluster leaders for that cluster. If we did not observe this rule then we might find that a higher-level cluster could not convene because none of the nodes capable of being peers at that level were leaders of their respective clusters.

Connectivity

The peerage for a cluster contains all nodes in that cluster expected to partake in cluster-wide shared state such as lock management. Any peer may be acting as a lock holder or a lock master at any point in time. As a result, we should expect that a node be connected to all its peers at all times.

However, although a flat cluster guarantees any-to-any connectivity, a metacluster only guarantees such connectivity between its own cluster members (ie. the leaders of the subclusters). We need to augment this with a fault-reporting mechanism by which any peer can report suspected loss of connectivity to another, so that one of the partially connected nodes can be removed from the cluster. The metacluster mechanism does not need to provide the fully-connected guarantee itself as long as there is an independent fault manager somewhere in the software stack which can deal with a breakdown in communications.

Justification

What does this big mess of stuff actually add to our clustering? First of all, we note that some form of hierarchy is absolutely necessary in order to scale. We cannot guarantee any-to-any connectivity between a million nodes without swamping our communications fabric. In an L-Cluster of such a size, we might expect the MTBF to be under sufficient control that we only suffer a cluster transition every few minutes, but it is hard to imagine establishing any-to-any comms between that many machines quickly enough that we can recover in time for the next failure.

In a distributed S-Cluster of similar size, the situation is different. We expect a lower MTBF (or at least the mean time between transitions, MTBT) to begin with (because we may have test machines and workstations being rebooted for perfectly valid reasons). We also expect to have much less bandwidth between some of the metacluster members (we may have remote clusters linked over external IP connections) and to have very asymmetric load capacity (say, only a few servers capable of acting as metacluster leaders at each site).

How can we deal with these problems? First of all, in an L-Cluster, the limiting factor is the requirements for synchronisation between all machines in the cluster when something goes wrong. Verification of any-to-any connectivity requiring even only a million broadcasts will be hideously expensive. However, with a dense enough fabric, we might expect that broadcast from one specific node can be done efficiently enough, even if only simulated via point-to-point interconnects. We can use the cluster hierarchy to distribute broadcast messages from the cluster leader to each cluster member, ie. to the leaders of all the subclusters, and repeat the process down the cluster hierarchy until all nodes have been contacted. The acks from each node can likewise be propagated back up the tree and merged into a single ack at the top level. The total number of packets transmitted over the entire fabric is O(n) but the total forwarding distance is only O(log(n)).

In other words, as long as recovery of the cluster services (such as the DLM) can be performed using such fan-out/fan-in broadcast+ack communication primitives, recovery of large numbers of machines should be feasible in a short time. By using hierarchical clusters, the individual cluster membership transitions involved can be lightweight, typically only involving ten or so nodes at once (the most complex cases involving death of a cluster leader). However, that membership transition can result in the eventual recovery of large numbers of nodes, because we can allow a low-level cluster transition to result in peer recovery of any metaclusters involved.

If a cluster transition only involves the arrival or departure of nodes which are satellites to the metacluster, then the metacluster’s peer list is unaffected and no recovery is needed at all. This is the property which allows us to conceive of extremely large S-Clusters using this design. The fact that nodes may be peers only up to a certain level and satellites above that level allows us to limit the number of peers which are present in the higher levels of the metacluster. This reduces both the number of transitions in those higher levels (keeping cluster availability high), it also reduces the high-level cluster’s traffic.

In short, we envision two scenarios. In L-Clusters, the top-level cluster traffic is high but the fabric is dense, and the main job of the hierarchy is to limit the cost of completing cluster membership transitions. Recovery itself is always done at the top level of the cluster, because most resources in the cluster are shared around the entire cluster.

In S-Clusters, most resources are local to subgroups in the cluster hierarchy. The cluster hierarchy's main job is to prevent there from being many transitions at all in the top level cluster, even if lower level clusters are transitioning regularly. The cluster hierarchy removes the need for the top levels to carry any traffic at all when an internal node of some subcluster dies, as long as that node's peer level is less than the metacluster's own level.

Naming

In a cluster hierarchy, any one node may in fact be a member (or a peer, or a satellite) of many nested clusters. When a process wants to access a cluster resource, it will need to indicate which of these metaclusters it is talking about. We also need a way to identify other nodes in the cluster or metacluster.

There is one major requirement for the naming which we must not forget, however. It is important that the clustering API supports the ability for existing clusters to be bound into metaclusters without upsetting any applications already running on the old clusters. In other words, adding a new level of metacluster should not invalidate the names being used to identify existing clusters and nodes. This basically prevents us from using any naming which relies on a unqiue name for the root (top-level) cluster in the hierarchy.

Consider as an example a hypothetical cluster I might set up. Say I have a cluster “SCT” for my own PCs. I might configure sub-clusters “DEV” for the development workstations, and “TEST” for test boxes. My primary machine might then be “DESKTOP”, with a cluster pathname “SCT/DEV/DESKTOP”.

That “SCT” cluster could be bound into a “SCOT” metacluster for all Red Hat sites in Scotland, and _that_ cluster could be bound into the top-level“REDHAT” cluster.

Consider that Alan Cox may have a similar set of clusters in Wales for his own machines: so we have a“WALES” cluster in “REDHAT” too, with subclusters “ALAN” and “DEV”.

How does naming look in this example cluster hierarchy?

Well, my desktop box in this example is present in four different clusters: “REDHAT”, “SCOT”, “SCT” and “DEV”. Remember that my desktop box may actually be a peer of any or all of these levels, and as such may actually be a cluster leader for any of these clusters: as such, that machine is not just several levels removed from the REDHAT cluster, it might actually be the master cluster leader for that whole metacluster.

So, it doesn't make sense to say that that machine belongs to one level of the cluster and not to others. There is no default level of the cluster which the API works at: the machine may be a cluster member of, and will be a peer of, many different clusters at once, all of them equally legitimate clusters.

So, these four names --- "REDHAT", "SCOT", "SCT" and "DEV" --- must be unique, so that they can be referred to as local cluster names all on equal footing. It follows that no one path through the cluster hierarchy must ever include two clusters or metaclusters with the same name.

How then can I refer to Alan's main desktop machine from mine? Well, Alan's box is not in the "SCOT", "SCT" or "DEV" clusters but it does exist under "REDHAT", so I can refer to his development cluster as "REDHAT/WALES/ALAN/DEV" and to his machine as "REDHAT/WALES/ALAN/DEV/DESKTOP". Note that if we then decide to bind the whole "REDHAT" cluster into a higher-level "LINUX" cluster including other organisations, "REDHAT/WALES/ALAN/DEV/DESKTOP" still uniquely identifies Alan's desktop machine. In this case, the name "LINUX/REDHAT/WALES/ALAN/DEV/DESKTOP" will be an equally valid name for the same machine.

There is an important related question here. Do we want a cluster namespace in which cluster peers are named too? In other words, in the example above, does the name "DESKTOP" resolve to my desktop machine, or is it an unknown name because there is no cluster with the name "DESKTOP"?

This really depends on which software component is doing the lookup. As far as the cluster APIs are concerned, when you request a cluster handle to operate on, you always supply a cluster name and a security domain, never a node name. As a result, the question of namespace clashes between cluster names and node names simply does not arise.

However, some interfaces may want to allow both nodename and clustername references transparently to the user. For example, we may want to be able to telnet to, or to be able to send print jobs to, either a specific node or just to the cluster (relying on the cluster software to redirect the connection to an appropriate machine). In such cases, "<clustername>/<nodename>" will be the correct way to produce a composite nodename including the clustername. However, for all of the low-level cluster APIs, such forms will not be recognised: the cluster name and node name will always be distinct things as far as the API is concerned.

Recovery

In a hierarchical cluster, we have different membership lists which can change when transitions occur. At each level of the cluster, we have both the cluster membership and the peer membership to worry about.

These two types of recovery are quite distinct things. In a large L-Cluster, for example, when a node dies it will cause a transition in the smallest cluster of which it is a member, and will of course also cause a peer membership change there.

However, in the higher-level metaclusters in that L-Cluster, there will be no cluster membership transitions (unless we were unlucky enough that the node which died was the cluster leader of its local cluster, in which case there will be membership transitions at higher levels as new a cluster leader is elected).

There _will_, however, be a peer membership change in all parent clusters of the local cluster of the dead node. That is the whole point of the L-Cluster: cluster leaders in each metacluster can communicate the information that a certain node is dead upwards to the top-level metacluster, and we can then take any necessary corrective action to perform a controlled transition on the peer membership of that cluster without having to perform expensive verification of the integrity of the clusters and the connectivity between every single surviving pair of nodes in the full cluster.

Consider then what happens if we have, for example, a large cluster filesyste using a distributed lock manager which is consistent over the entire L-Cluster of several thousand nodes in a large High Performance Computing cluster. On death of a node in that cluster, a few localised machines perform a cluster membership transition, and then they perform a peerage recovery (performing recovery of any cluster services which are operating at the level of that local cluster).

That local cluster then sends a message to the next higher level cluster indicating that there is a change in the peer list for that cluster, and peerage recovery happens for that cluster... and so on all the way to the top-level cluster. Therefore, when one machine joins or leaves the cluster, we do not have to perform a full cluster membership transition for the whole cluster: rather, we have the much simpler task of dealing with a peerage transition in which we are simply told by one of the cluster members exactly what has happened to the peer list. We don't have to work out what went wrong.

This is necessarily a more scalable scenario than having every member of a 10,000-node cluster have to negotiate with every single one of its neighbours every time a node joins or leaves the cluster! But is it scalable enough? Can we perform recovery efficiently for coherent services such as a DLM on such a large cluster when the peerage changes?

I believe we can, because the hierarchical nature of the cluster gives us a way to contact all nodes in the cluster efficiently. On peerage transition, the cluster leader can begin the transition by sending a command to each of its cluster members. This is a hierarchical cluster, so the top-level metacluster members are actually the cluster leaders of the next-level-down metaclusters, and those cluster leaders can then fan-out the message down to all of their cluster members, and so on down the cluster hierarchy until all peers in the entire cluster have been contacted. Replies can fan-in up the cluster hierarchy in a similar manner

As long as peerage recovery can be expressed in terms of such fan-out/fan-in broadcast operations, plus limited point-to-point traffic between specific nodes, we can use the cluster hierarchy to make peerage recovery sufficiently scalable to work well even on enormously large HPC clusters.

What about the S-Cluster case? In a typical S-Cluster, this distinction between peer and membership transition again does exactly what we want it to do. If we have some remote location in a company with its own internal metacluster, and further sub-clusters inside that according to the pattern of use of the machines there, then in an S-Cluster we would only have a few machines at that entire site which were peers for the organisation's higher level clusters. As a result, any node arrivals or departures within that site, as long as they don't concern those high-level peer servers, will not cause any membership _or_ peer transition in the top level clusters.

Irregular Hierarchies

Think about the implications of:

- Different cluster subtrees having different depth

- Binding clusters together into metaclusters without reconfiguring every member of every subcluster.

Authentication

Permission --- to join a cluster, to become a peer (especially for metaclusters). If we have different cluster passwords for a metacluster, then how do we decide which one wins (ie. which is the real metacluster, which is spoofed? Over an organisation, can we realistically rely on nobody doing the wrong thing?)

LocalWords: subcluster metacluster's metacluster ie subclusters MTBF comms IP
LocalWords: MTBT acks ack metaclusters MEMBERs cluster's DLM proxying
HPC LocalWords: transitioning localised neighbours organisation's spoofed

CHAPTER 6

Cluster API

Here we will discuss a number of issues relevant to the creation of clustering APIs. These issues only concern the API as expressed on a single node. They may deal with communication between a user process and a cluster server process on one node, for example, but we be will completely ignoring issues to do with communication between cluster nodes here.

Issues to be dealt with uniformly

In designing the way we build APIs for the cluster core services, there are a number of specific problems we have to solve, including:

- Dealing with client death
- Security
- Cluster node naming
- The specialised needs of recoverable services
- Efficient ways to specify async event delivery

Those are the problems themselves, but we also need to bear in mind that to keep the code maintainable we need to aim for:

- Consistent APIs
- Isolation of the future complexity of hierarchical clusters in a forward-compatible manner
- Modular implementation of the common API functionality such as security and event delivery.
- Flexible implementation. As an absolute minimum, I require that all the APIs being developed are capable of being implemented both in kernel space (only on Linux boxes) and in user space (using threads if necessary, and preferably portable to other Unixen).

Dealing with client death

However the API is implemented, it is absolutely imperative that the code implementing the service on each node is able to detect the death of a client process which is holding a cluster resource, so that it can release that resource. If the service is being implemented in the kernel then obviously it is easy enough to track process death, but we need this functionality for services implemented by server daemons too.

A cheap way of obtaining this notification is for the service to be implemented over sockets. Unix domain sockets have reasonable efficiency for local process intercommunication, and the server can easily detect the death of a process connected by such a socket.

A kernel-based implementation of cluster functionality will probably use dedicated syscalls instead, but kernel implementations always have more freedom to play clever games to achieve the necessary functionality. It is the user-space implementation which is more constrained by available unix functionality, so right at the start I will make the assumption that all requests passed between client processes using the cluster APIs and cluster service daemons implementing those services will be done using unix domain sockets. (Other communication channels, such as signals and shared memory, can be used to augment the socket-based communication too, of course.)

Security

Why do we need security?

Our filesystems, shared printer queues and cluster namespace will all be using cluster resources. They will _all_ require barriers for recovery; some will require cluster locks, some will require cluster name bindings.

If we allow unrestricted access to these resources by any user, then we have just allowed an unprivileged user to violate the integrity of the entire cluster core.

Simple Unix uid mapping is not sufficient to solve the security problem. In a large heterogeneous hierarchical cluster there may be many uid spaces participating in the cluster. We also want to support partitioned namespaces: not only should individual resources (which we might conceivably uid-protect) be inaccessible to the wrong user, we may sometimes also want the namespace itself to be unbrowsable to unprivileged clients: thus we need protection on the whole namespace too, not just uid protection on individual objects.

For the purposes of the core cluster APIs, I propose a very simple mechanism to allow for multiple security domains: simply use the filesystem to mediate the communication between client processes and the cluster service daemons, and use filenames to name security domains. For example, we might have pathnames

/tmp/cluster/<CLUSTER-NAME>/sockets/namespace/USER
/tmp/cluster/<CLUSTER-NAME>/sockets/namespace/SYSTEM

which refer to unix-domain sockets by which clients processes can connect to and send request to the local node's cluster namespace daemon. There would be no difference between these two files except for permissions: normal filesystem modes and attributes can be used to set appropriate permissions on each such socket file.

Note that we define these two security domains to be valid on all cluster systems. SYSTEM is the default security domain in which privileged cluster services operate; USER is the default domain for application cluster requests.

There is a second advantage to this mechanism: it allows us to pass security credentials for these cluster sockets around using standard unix fd-passing. This leaves the option open that at some point we can implement a security server to authenticate individual processes' access to higher privilege levels, granting those privileges by passing back an appropriate socket fd.

If we have many such security domains, then obviously we want some way to ensure that they can be set up automatically in each socket directory and that permissions are set up appropriately on each host. However, at this stage this mechanism simply lets us say that security domains via socket permissions provide the functionality we want: the management on top of that is another layer which we can ignore for now. (The API won't care who is responsible for setting up these sockets as long as they exist.)

If we want to apply uid-based ownerships to objects within a specific cluster service then we are certainly free to do so, so long as we can detect the uid associated with a connection to one of the server sockets.

Note that the core API library services being shared by the various cluster APIs can use these security domains in whatever way they want. They may choose to define each security domain as a separate namespace or to have them share a namespace. They may choose to allow browsing between security domains or not. The correct decision may very well depend on the service: lock manager domains may want to be totally opaque to each other, but the cluster namespace service's whole point is to make all the cluster name bindings visible in a single namespace, for example.

*** I'm open to suggestions concerning whether or not uid/gid/mode *** protection should be implemented as a required option in the APIs. *** The requirement that the cluster should still work if we cross uid *** mapping domains makes this unclear. The real question is: should *** arbitrary unprivileged code have access to the cluster namespaces at *** all? If so, then uid protection on resources is necessary.

Cluster Node Naming

Consider a hierarchical cluster containing a top level cluster ORG, and subclusters all the way down to ORG/UK/EDIN/DEV/TEST/TEST1. A node in that cluster --- say, ORG/UK/EDIN/DEV/TEST/TEST1/TESTBOX2 --- may want to participate in cluster services at any of these levels. We need to be to specify exactly which of these layered clusters we are referencing in every single API call to a clustered resource.

Fortunately, the pathname-based access to cluster services proposed for security domains is also ideal for separating out access to different clusters: we simply include a cluster-name component in the pathname used to access the cluster socket. We already have a requirement that any cluster name will never appear more than once in a fully qualified hierarchical cluster or node name, so these names are guaranteed to represent unique references to a specific level of the cluster hierarchy on any given node.

As far as the API is concerned, I propose that all access to cluster services by a process be preceded by a call to

extern struct cluster_handle get_cluster_handle(const char *cluster_name, const char *security_domain, int flags);

and that the "struct cluster_handle *" returned by this call be used as the first parameter to every subsequent call to cluster resource APIs for that cluster. This allows the user to talk to multiple clusters and multiple security domains at once, while still hiding the details of how we might implement these features.

Recoverable Services

Things are more complex when we are doing recovery. The APIs must be able to continue to work selectively during recovery.

As an example, the barrier API may be being used by a pair of cooperating user processes on two different nodes when a third node joins or leaves the cluster. That cluster transition does not affect the barrier, so the user applications should not notice any change: the barrier API should just be stalled temporarily during the transition. However, an internal cluster service such as the namespace service is a completely different matter: it may rely on the barrier API in order to synchronise its own recovery.

Similarly, lock manager traffic may be suspended during transition and resumed afterwards, but a clustered filesystem may want to perform its own lock manager operations during the recovery period. Note that this example shows that we must be prepared to have the same resource visible both during recovery and for normal operations: we cannot simply partition the resources visible during recovery into a separate security domain as described above.

The default behaviour for the API _must_ be that a cluster transition causes a temporary stall but no other observable behaviour for applications which are not using resources affected by the transition. For APIs such as the namespace and lock manager, a node death releasing a resource ought to be largely indistinguishable from a voluntary release of the same resource as far as the effect on other nodes in the cluster is concerned.

So we have two problems:

- How do we specify that a specific API request is recovery-privileged and should not wait until the end of recovery?

and

- How do we restrict this functionality to privileged processes only?

We can achieve both of these by adding a recovery-ok flag to the flags in the “flags” handle to the get_cluster_handle call, and restricting that flag to be legal only on the SYSTEM security domain. That allows an application to obtain a separate cluster handle to pass through API calls which want to run during recovery. It avoids polluting the API of other calls with recovery information.

Asynchronous Event Delivery

One of the important things that the cluster API must be able to do is to deliver events asynchronously to processes. Cluster transitions, loss of a barrier if a participating node or process dies, or notification that another process wants to steal a lock are all async events which a process using the cluster API will want to deal with.

So, how do we handle cluster callbacks to the user process? The normal Unix mechanism for this is to use signals, of course. However, we have several problems with that:

- We do not necessarily want our cluster service daemons to run with privileges to kill every process in the system; and
- It is impossible to pass arbitrary data through a signal on Unixen which do not implement posix queued signals;
- The client cannot tell how many signals were received;

- There are a limited number of signals available but arbitrarily many different events which may occur in a cluster (in a lock manager API, for example, each lock a client process owns may have its own callback routine to be called when that lock is wanted by another node).

We can address all of these problems by using sockets. We can allow the client to give the server an arbitrary cookie of information to be associated with a potential future event. If that event occurs, the server will simply pass that event back to the client via a unix domain socket reserved for those callbacks. Even if the Unix implementation being used only supports SIGIO, that is enough for the local library part of the API to trap the IO, decode the cookie and perform the callback.

This obviously restricts the caller's freedom to use SIGIO. That's unfortunate, but unavoidable if realtime sigio is not available. On recent Linux kernels we can use realtime queued sigio to use a different signal for the async callback socket, to avoid interfering with the normal signals.

Obviously, a mechanism by which the signals can be blocked is required, and this mechanism must be exposed in a portable way to the user.

More of a problem, we must have a way to sycnchronise these cookies with other foreground events. That is the responsibility of the API library in the user process. For example, if the user creates a lock with a callback, then deletes that lock, there is a chance that the server has already sent us a callback on that lock: the API must, when it is decoding the cookie, detect that the callback is no longer valid and must silently discard it rather than delivering it to the application.

LocalWords: APIs async Unixen namespace uid namespaces unbrowsable unix fd ok LocalWords: service's ORG subclusters struct const

CHAPTER 7

Recovery

Recovery Types

In most existing cluster systems, recovery is a process initiated directly after a successful cluster transition, and that is the end of the story. That is also more or less true in a flat, peer cluster in our cluster model, but the existence of either satellite nodes or a cluster hierarchy complicates things somewhat.

We need to make a clear distinction, then, between different events which can occur in a cluster or metacluster. (Refer to the definitions in hierarchy.txt: they are quite important here.)

- CLUSTER TRANSITION is the event which occurs when a cluster's membership list changes.
- CLUSTER RECOVERY is the recovery initiated with respect to that membership list.
- PEER RECOVERY is the recovery initiated with respect to _every_ peer in the cluster's peerage.
- SATELLITE RECOVERY is the recovery initiated with respect to satellites for which this node is responsible.

Recovery Daemon

Once a cluster initiates recovery, we need to signal the various cluster daemons to recover in some given order. Obviously, the task of starting up the cluster in the first place also has to establish cluster services in some order, and the order in each case is dictated by the various dependencies between services. In other words, startup of cluster services when a new node joins a cluster is related to recovery.

We can also notice that there are certain cluster services, such as the comms and barrier services, which are relied upon to provide cluster-wide synchronisation primitives. Without those primitives in place, there is no cluster-wide recovery. There is therefore dependency between recovery services on a single nodes as much as there is dependency between nodes.

Therefore, we need to have a local daemon which can order services to recover in the appropriate order, even in the absense of cluster-wide synchronisation. Obviously one of the first services to be recovered should be the barrier synchronisation service so that later recovery stages can rely on that for cluster-wide synchronisation.

This daemon will also be responsible for the startup of the local node's variou cluster daemon processes, as those need to be started up in the same order in which we deliver recovery orders. As a secondary issue, this same master daemon will be responsible for detecting the death, or failure to respond, of any cluster service daemon and to kill and restart the entire cluster stack if that is detected.

/cbin/init starts up all internal cluster components, and on failure (process dies, process fails to respond in a given timeout) will kill and restart all components fro scratch.

One possibility is to organise it along the lines of inittab, with a controlling /cetc/inittab file:

```
0:/cbin/ccomms
0:/cbin/integrate
0:/cbin/barrier
```

1:/cbin/nameserv
2:/cbin/confrelay
3:/cbin/quorum
4:/cbin/cdb

-----

* On cluster transition start (or peerage transition):

init signals each cluster component that we have begun transition. (Tag the transition with the new local transition sequence number.)

* On cluster transition end:

Once all components have ACKed the transition, go through each component, one by one, doing:
+ Send a recovery event with the new transition sequence
+ Wait for a recovery ACK with the right sequence number

Wait for a second, recovery-complete ACK to arrive from all services.

Complete the recovery barrier.

CHAPTER 8

Communications

How do applications in the cluster talk to each other?

This is a more complex question than it first appears. There are a number of things to note:

- We may want communication path failover on hardware failure.
- Flow control is a _very_ complex thing, and we do not want to invent tcp if we can avoid it, so reusing existing networking code is important where possible.
- In the future, we will want to allow cluster communications to use advanced, high performance transports such as Myrinet or VIA.

The cluster integration layer protocol is a special case here because it has an additional requirement: it must be able to support arbitrary additional, low-performance communication channels between nodes to allow arbitration when a cluster fault occurs. For example, two nodes on a shared scsi bus may eventually allow integration-layer communications over that scsi bus by using target-mode in the scsi controllers, and a backup serial or parallel connection between nodes may also exist. The sole purpose of such backup connections is to allow the cluster to negotiate the graceful exit of one node from the cluster, so that (for example) if the ethernet between the nodes in a two-node cluster dies, we can recover without losing quorum. We cannot expect necessarily to be able to reuse the normal network stack for such communication paths in all cases.

For all other communications, I propose that we simply use the normal socket API, with these additions:

- We should provide an augmented gethostbyname() to return IP addresses for nodes named in the cluster namespace. We will expect all cluster nodes to have an IP address, regardless of what other extra transports may be available.
- Normal inter-application communications should begin by binding a socket using modified versions of "bind" and "connect". The "cluster_bind" and "cluster_connect" variants will not use the standard struct sockaddr to name the local and remote host: rather, they will allow a cluster node name to be provided instead.

These modified calls will simply return a socket which the application can use as before. In the first case, such sockets will always be IP sockets. In the future, we may have VIA support too, and in this case adding VIA sockets to the kernel will allow existing cluster applications to use the new transport without modification.

However, the normal socket API will never get the very best performance out of VIA: it will always be necessary to provide an additional API to support pre-posting of read buffers, for example, if we are to provide the very best zero-copy performance over VIA. Applications will need to be coded to that additional API if they are to make use of the highest levels of performance, but the use of sockets with cluster naming as our standard inter-application cluster communications API will at least allow VIA to be used with some respectable performance gain.

Note that the use of normal IP sockets for inter-application traffic has two implications: such traffic will not necessarily observe cluster transitions, and the cluster software itself will not be able to route around failures. That is fine: there is no need for most applications to see cluster transitions, as the various cluster APIs offered by recoverable services such as cluster filesystems and lock management will keep node failures transparent to the applications.

As for routing, we assume that enough routing will be performed as a part of the IP-management cluster services that any IP sockets running between the nodes in our cluster will survive any single node death. Ensuring that property will be an important job for the IP address failover core service.

In a hierarchical cluster, the cluster layer does not guarantee that all peers can necessarily see each other at all time. In the presence of routing glitches, two or more cluster leaders may form a metacluster even though some members of one subclusters are not visible to all members of another subcluster. Wemust have a fault reporting API defined to allow the applications to verify that communications are still working between any two arbitrary peers, so that if any pairwise connection does fail, the fault can be remedied by the eviction of one or other of the failed nodes.

For core, recoverable cluster services, there is precisely one communication primitive that we will rely on to be truly scalable in the case of large, flat HPC clusters. Namely, any cluster leader must be able to send a reliable broadcast RPC to all peers in that cluster. That will involve the cluster leader passing the RPC to all its cluster members, which of course is the same as passing it to the cluster leaders of all the subclusters. Those will pass it down to the next-level-down cluster leaders, and so on until every cluster leader of the lowest-level cluster have received the message; at which point the RPC can be executed on each cluster peer, and the resulting ACKs passed back up the cluster chain to the metacluster leader.

This fan-out/fan-in primitive provides a fast way of propagating cluster transition orders to every peer in the cluster when a peer transition occurs, and it will be important to support this for very large scale clusters.

CHAPTER 9

Integration

*** : comments regarding open discussion points for the reader @@@ : comments regarding open discussion points for the author. Kick me if there are still too many of these by the time you read this document!

The Integration Layer

Concepts

Cluster Controller: The single node which is responsible for coordinating a cluster transition. The first task during cluster transition is election of such a Cluster Controller.

Cluster Map: A “Cluster Map” is the picture, maintained at any node in a cluster, of what the cluster appears to look like from the point of view of that node. Every cluster has a Cluster Map which indicates the members of that cluster. This is the “Current Cluster Map”.

During a Cluster transition, we must also maintain a “Proposed Cluster Map”, which describes the state we think we are moving to. This map is dynamic: a cluster transition may involve resetting of communication links all over the cluster, resulting in many nodes temporarily dropping and (hopefully) resuming communication with their neighbours, or multiple new nodes finding and rejoining our cluster. The Proposed Cluster Map is updated on each such communications event. Once the Proposed Cluster Map has stabilised, the Cluster Controller makes a Commit Cluster Map out of it and begins a 2-phase commit of that new state around the cluster.

API

No, there’s nothing here yet.

Overview

Overview. The Integration Layer is supposed to do absolutely nothing while the cluster is running normally. Only when circumstances change is it invoked. The Events which affect the Integration Layer are usually generated by the Link Layer. They include:

++ **New Link.** This event merely announces the discovery of a new node on the network, and the establishment of a Link to that node. No link state is implicit in this message: we expect a Link State event to follow, and no cluster transition is invoked until we get that Link State event.

++ **Link State.** A Link’s state (Down, Reset, Degraded, or Up) has changed. Oops: time to begin a cluster transition. If a cluster transition is already in progress, this requires us to update the Proposed Cluster Map and adjust the transition state if necessary.

Every Link State event causes a cluster transition to be initiated on the local node.

++ **Operator Intervention.** We must support external commands to modify the cluster map. For example, the operator may request that a specific node be taken out of the cluster for maintenance; node shutdown must also perform a clean cluster exit.

Operation Requirements

Before launching into a design discussion, we need to be clear about some of the properties required of the cluster integration layer. This section will act as a rationale for some of the design decisions below.

Remember that all cluster transitions merely involve merging or partitioning of existing clusters. This property is a result of the design decision to consider both newly restarted hosts and hosts evicted from an existing cluster to be single-node clusters in their own right: the construction of a new, larger cluster of such hosts is merely a sub-case of the merging together of two clusters, rather than being a special case in its own right.

Partially connected clusters. This happens. We need to deal with the situation where some members of a cluster can see some, but not all, of the others. This can result from a fault occurring in a working cluster. We also need to deal with the situation where we are proposing the merging of two or more smaller clusters, but where not all of the merging machines can see each other.

We can define the Cluster Superset at any node to be the set of all nodes which are connected to that node, or which are in the Cluster Superset of any connected nodes. The Cluster Superset is the transitive closure of the direct point-to-point node connectivity relation.

We have several requirements resulting from the need to deal with partially connected Supersets. First of all, the cluster integration protocol needs, at some level, to propagate cluster connectivity information over all the nodes in the Superset. When we merge a set of machines into a larger cluster, we need somebody to select the largest fully-connected subgraph of the Cluster Superset (or, rather, the subgraph with the most votes) to form a new cluster.

We need to ensure that this configuration is stable. Any attempt by one of the rejected nodes to join the new cluster must be rejected. We can ensure this by automatically failing any merge between clusters if any of the nodes in one of the merging clusters is unable to see any of the nodes in any other cluster in the merge.

Operation Requirements: Cluster merge. A cluster merge is the process which occurs whenever any node detects the existence of a neighbouring node which ha the same cluster name but which is not currently a member of its cluster. The cluster merge simply combines the two newly connected clusters together into one larger cluster. Given that all nodes start off operating as single-node clusters, this mechanism will deal with individual nodes joining an existing cluster as well as with two previously-partitioned clusters reestablishing connection with each other.

As we just mentioned, it is important for a cluster merge to result in a new cluster which actually works. We require that clusters must be fully connected at all times, so this means that a merge must preserve this property. We can simplify our operation enormously by ensuring that we only ever try to merge working clusters. If we do this, then by the time we come to do a merge, the merging clusters will already have elected their own Cluster Controller nodes (CCs), so the merge becomes a simple matter of one CC handing control of its cluster's nodes to another CC. The CCs can exchange information about each others' clusters, and within each merging cluster the new proposed cluster membership can be propagated by the CC so that pairwise connections between all the newly discovered nodes can be established. If any of these connections fail, then the CCs concerned have to abort the merge, and must not make another attempt at a merge until some other cluster transition event occurs which makes it useful to retry the merge.

What happens, then, if we attempt a cluster merge and find that there are just a few pairwise connections between the clusters which cannot be established? We reject the merge, but it doesn't need to stop there. By breaking up one or more of the smaller clusters to evict some of the nodes which are failing to establish cross connections, we might be able to establish a new larger cluster. However, at the same time we do not want to risk breaking up existing clusters unnecessarily just to attempt such a modified merge.

We can introduce a new rule which helps here: after a failed cluster merge, the cluster with the most votes never, ever tries to break up. However, we can allow smaller clusters to fragment voluntarily. If a CC decides not to proceed with a merge with a higher-voting cluster, it can look at which inter-cluster connections succeeded and which failed, and can try to identify exactly which of its own members are in fact able to see all of the other cluster's members. If it finds that some of its nodes would in fact be able to cross over to the other cluster, it can evict those members from the local cluster. Those evicted cluster members will immediately attempt a cluster merge themselves, and in that process should always look for the largest neigbouring cluster to join, which in this case will be the larger of the two clusters which tried to merge in the first place.

The result of this will be that we can donate nodes from one cluster to another to grow the larger of the clusters, while preventing partially-connected nodes fro ever disrupting the membership of the larger cluster. The cluster breakup mechanism will therefore never break quorum.

Operation Requirements: Cluster breakup and communication faults. Cluster breakup is potentially a little more complex, since when a disconnection event occurs in a working cluster, we do not yet know the extent of the damage: perhaps one single node has failed, or perhaps _all_ of the other nodes have become "failed" because our network connection to them has died.

One thing to note straight away is that a single channel error event must not be sufficient to trigger cluster disintegration: if only one channel in a point-to-point link (ie. only one route of multiple redundant routes between a pair of hosts) has died, we must retain the cluster.

We still want to trigger a cluster transition of some form, as the loss of any channel always causes a complete link reset of the communications between those two nodes and we need to recover the context of any messages which were in transit between the nodes at the time. However, if the link does recover successfully (say, it fails over to a backup ethernet wire), we can record the resulting cluster transition as a null event: we still need to perform some recovery to retry any messages which were in transit when the error occurred, but we do not need to recover any cluster-wide global state in this case.

*** DISCUSSION POINT: ***

Does the above point make sense? We _do_ need to perform some recovery in this case, so as far as the cluster integration layer is concerned this section is entirely accurate. However, we need to think about the visibility of such null transitions to higher layers of the cluster software.

We can provide reliable data pipes between processes in the cluster at a higher level using packet sequence numbers which recover themselves over a cluster transition. Think about two communicating high-level processes --- a web server and a CGI server, say --- in the case where an unrelated node drops out of the cluster. Those two processes should not be affected by the cluster transition, other than at most a temporary stall in the servicing of cluster IO and DLM requests. The IO pipe's internal recovery from to channel failover can be transparent to the applications.

It is already planned for cluster transitions to be given a brief synopsis of whether any significant node changes have occurred during the transition, so by just making the rule that all communication errors produce a full cluster transition, we are not forcing all of the rest of the cluster software to do full recovery. On seeing the flags “no cluster membership change” in the transition block, they can simply avoid doing persistent state recovery and limit themselves to communications recovery.

General operation

OK, I admit it. It would be inaccurate to describe the Integration Layer’s operation, as described in the rest of this document, as being purely a consequence of the requirements above. Some of the requirements already stated are actually requirements which arise from design decisions, not functional requirements. Please bear with me on this --- I *think* that where I’ve described a mechanism, rather than a rationale, in the above text, I have good cause in that the mechanism represents a broad design decision (rather than a mere point of detail) which simplifies the entire layer’s design. I’ll now try to explain the broad principles of operation I envisage for the Integration Layer, how it all works together and why the overall plan makes sense.

You might have noticed that I have simply assumed, in the above, that every single cluster transition can be expressed as a combination of cluster breakups and cluster merges. This really, really simplifies things enormously. For one, it means that we never, ever have to deal with individual nodes in this model: every isolated node is nothing more than a one-node cluster with an already-elected cluster controller.

Cooperation between different, already-established groups of nodes therefore reduces to a problem of point-to-point communications between individual CC nodes. Such cooperation between currently-unclustered nodes is *always* a cluster merge operation, never anything else (except that, possibly, we may want to add diagnostic and monitoring code on top: that does not invalidate the Integration mechanism as presented here).

A second major advantage of the merge/breakup concept is that it deals very cleanly with errors which occur during a cluster transition. If a cluster detects an internal error (loss of a node or link) while in the progress of completing a cluster transition, it immediately drops everything and sorts out its internal problems first.

Thirdly, it also simplifies the management of complex faults in a cluster where complete any-to-any connectivity is lost. Such faults are dealt with in the breakup phase of the transition, and in that phase, we always know _exactly_ who is already in the old cluster and we have to deal with them alone: we never have to deal with any nodes outside the cluster until the cluster's own internal state has been reestablished.

It is important to refer back here to the general design principle of hierarchical phases of operation. In this merge/breakup model, cluster breakup is a more fundamental operation than cluster merge. Remember that the hierarchical phases principle means that any error takes us to a previous phase, and forward progress is only ever achieved by consensus. By dealing with the cluster breakup phase prior to the cluster merge phase, we ensure that merging is always a cooperative operation between working clusters: any internal cluster error is dealt with before we ever get as far as merging clusters.

If, during a merge, one of the merging clusters detects the loss of an internal node, that cluster simply aborts the merge and reverts to the prior phase, reconfiguring itself to deal with its internal error before bothering with the rest of the world. The fact that we will start rebuilding this state by reentering the abort phase must ensure that we propagate the error to the rest of the nodes we were merging with, so that they themselves can detect the abort of merge phase.

The expected consequence of this is that the cluster which detected the internal loss of node will quietly resolve its internal state before trying to join anybody else. Either it will complete that before the rest of the Cluster Superset finishes merging (in which case it will try to trigger another merge), or before (in which case the in-progress merge will be aborted). Either way, the previous, aborted merge is automatically resumed only after the component clusters have dealt with node loss. Cluster merging simply does not have to deal with node loss: it only has to deal with a general "oops, something went wrong" error which always results in a regression to the abort phase.

Transition Phases

A Cluster Transition involves progress through a set of distinct phases. The execution of each phase may be very rapid if we happen to know in advance what sort of transition is occuring (eg. controlled eviction of a single node does not have to perform a complete CC reelection if the remaining nodes can all still talk to each other afterwards, and the evicted node was not previously the CC).

This section will contain a complete description of each of these phases. The descriptions will begin with the objectives of each phase: what the pre-conditions and post-conditions are. The pre-conditions are sufficient to describe those things which the phase does NOT have to achieve, but can simply assume have already been achieved. The post-conditions describe the objectives of the phase by establishing the final results that the phase is required to achieve.

++ Abort Phase:

Tell everybody that something terrible has happened.

Pre-conditions: None at all: we can't assume anything here since this is the state we enter on getting an error of any description, either a recursive error during the integration process or a fault detected during normal running of the cluster.

Post-conditions: All cluster nodes which are still communicating with us, and which are in the current Commit Cluster; as well as all nodes which we may have been trying to merge with since beginning this cluster transition; have also returned to Abort Phase. The connectivity of the Commit Cluster is moderately stable.

Mechanism: The only inputs to the abort phase is the previous Committed Cluster Map, which exists at all times and which is not actually modified by the Integration Layer transition mechanism at all (at least, not until the entire transition process ha completed and a new Commit Map has been established); and the current state of each Link with nodes with the same cluster name.

When no Link state changes concerning neighbours in the Commit Map have occurred within a certain period (the Cluster Settling interval), we can enter Discovery Phase and start sending Discovery messages on all Links.

If a neighbour sends us a Discovery message before our Settling interval is complete, we buffer it for use once we have settled and entered Discovery phase ourselves. We must not actually respond to that message until we have Settled.

++ Discovery phases:

Gather information about the new state of the Cluster Superset

Discussion: The first process we enter in a cluster transition is the breakup phase: the eviction of any unreachable nodes from the old Cluster, plus potentially the eviction of further nodes if the remaining nodes are no longer all fully connected to each other.

To do this, we need somebody actually make the decision on what the “best” fully-connected subset of the old Cluster is still viable. We will call the node which makes this decision the Cluster Elector for the breakup phase.

We also need to remember that when we lose contact with a node in the cluster, that node may decide to evict itself from our cluster and form a new cluster on it own behalf, but communications with that node may be reestablished before we have finished our own cluster transition. This is dealt with by observing that the lost and recovered node must have a changed Cluster ID, and preventing nodes with the wrong Cluster ID from participating actively in the Breakup phase. The discovery phase therefore also needs to establish, for each node, whether the neighbouring nodes still have the same Cluster ID as we do.

In this situation, the lost node will just have to wait until all of the nodes which have still got the old Cluster ID have finished their eviction phase and have entered Cluster Merge phase (in other words, the whole mess is dealt with before we leave the cluster transition altogether, but we don’t have to deal with it just yet: the ordering of breakup phase before merge phase naturally deals with this sort of fault). If no nodes survive with that cluster ID, then that’s fine too --- the resulting smaller clusters will enter the merge phase eventually

We define the term Related Nodes to refer to the set of neighbours of a given node which (a) are still connected to that node, and (b) have the same cluster ID (which necessarily implies that they were part of the same Commit Set as us previously).

The Related Node Superset of the old Commit Cluster Map is the transitive closure of the Related Node function, and includes all of the old nodes which are connected together over any multi-hop route of Related nodes.

Cluster transitions can take arbitrary amounts of time, due to the fact that the Abort phase will stall for as long as a communication Link is unstable (in other words, faults which are transitioning rapidly are bad news --- we need to think about mechanisms by which to dampen the effects of such nodes, but such a mechanism probably involves voluntary exit of a node experiencing such problems and doesn’t directly change the Integration Layer’s behaviour at all).

As a result of this, we can imagine a not-quite-partition of the cluster, in which the cluster partitions but then a single node which acts as a bridge between the partitions comes back. If that node has not yet completed its own cluster transition (evicting itself from the previous cluster in the process), then all is fine: that node is still Related to some others, and the Abort which follows the link state change when the node rejoins its neighbours will eventually lead to the two partitions joining the same Related Node Superset.

If the "bridge" node has transitioned, on the other hand, we have two Related Node Supersets in the new cluster which are theoretically able to merge but which are divided by an unrelated node. At this point we are just doing breakup, not merge, though, so we have to deal with the Related Supersets as quite distinct entities. Therefore, we do _not_ want to pass discovery information over the bridge node: discovery information must only be passed to Related Nodes, not beyond.

Mechanism: For the Cluster Elector, we choose that node with the greatest"Value Function", where the Value Function of a Node is some function ordered by:

- -ve number of degraded links to nodes which are still in the node's Commit Map. We want to select a value function which minimises the amount of degraded link traffic in the common failure modes. Do we do this best by maximising the number of non-degraded links active at the Elector node, or minimising the number of degraded links? Probably the latter, so the value function (for now) always prefers nodes with few degraded links. Then,
- CC: The existing CC in the Related Node Superset is always preferred as Cluster Elector if it is still reachable via any route through that Superset (unless it has many degraded links due to recent faults). Failing that,
- Total number of votes from Related Nodes present on, or on nodes Related to that node; then, in case of tie, by
- Number of other nodes Related to that node; then,
- "Metric" of that node (a static configuration value intended to represent available CPU power); then,
- Lexicographic order of the node's unique nodename.

The important point about this function is that it is unambiguous. Any node can generate a Value for itself, and pass that to other nodes; given that information, any two nodes will always agree on which node has the higher Value.

*** Note that by using the degraded link count as a high-priority sort key and the node Metric as a low order key, we end up preferring to minimise degraded link traffic rather than placing the Elector on the most CPU-capable node in the cluster. Comments? Maximising the connectivity of the resulting cluster in the Eviction phase may require a good CPU --- should we optimise for that rather than optimising for communications at this point?

What we now need is a mechanism by which all connectivity information in the entire Cluster can be propagated to a single node which all nodes agree is the best one to be Cluster Elector. The mechanism needs to be able to deal with situations like this:

```
Node:A-------B-------C-------D-------E
Value: 4---------1------3-------2-------5
```

where we have very little connectivity in the cluster: maybe only the point-to-point serial cables between the nodes have survived, and each cluster can see only its immediate neighbours. We cannot simply propagate connectivity information to our neighbour with the best “value”, since nodes B and D in this example will end up sending their information in opposite directions. So, first of all we will decide who is to be the Elector by propagating Value information between nodes. Only once one node decides it has won the Election will we propagate information, and the Elector will guide that phase by explicitly telling the other nodes what decisions it has come to.

++ Discovery Phase.

Pre-conditions: All of our neighbours within the Commit Map have either entered abort state or are unreachable: we have Reset our Links to them and have waited long enough to be sure that either they have already completed the Reset negotiation or they aren’t going to.

Post-conditions: Everybody in the local Cluster Superset has propagated its local view of cluster connectivity (ie. which other nodes it can still see) to a new node which will become the “Cluster Elector”, which controls the move to the next phase.

Mechanism: In this phase we want to propagate the Election Value function between nodes, and we want to propagate connectivity maps for the entire cluster back to the Elector. We choose the following algorithm:

Each node starts by sending its own value function to each of its Related neighbours.

Each subsequent message between nodes includes a Value Function plus a non-empty list of connectivity maps. The Value Function of any message is the maximum of the Value Functions of any nodes whose connectivity is listed in the map. (The connectivity of a node is just a complete list of all other nodes still connected to that node, plus the Link state: DEGRADED orUP.)

When a node has received the initial Value Functions of each of its neighbours, it can start to propagate connectivity maps. Propagation works from the node with the lowest Value upwards.

The node with the lowestValue begins by sending its connectivity map to its neighbour with the highest Value.

When nodes has receive messages, they update their local database of connectivity maps to maintain both (a) the total set of all nodes whose connectivity information is known, and (b) a list of connectivity maps which are known to each neighbour. If a node discovers, in this manner, a new node which has a higher value function than any

[Described more fully in discovery.txt]

++ Eviction Phase

Pre-conditions: one node --- the Elector node --- has a full map of the cluster connectivity and knows that it is the Elector. All other reachable Related nodes are still blocked somewhere in the Discovery phase.

Post-conditions: we have established a Cluster Controller within the current Commit Cluster to guide us through the rest of the cluster transition, or we have left the cluster (and reformed a new cluster which will try to merge if possible with any neighbouring clusters).

The transition from Discovery phase to Eviction Phase is guided by the unique Elector node. The Elector is responsible for making a unilateral decision about which of the current members of the cluster should survive the transition and which get evicted.

This phase decides which node is going to act as Cluster Coordinator (“CC”) for the remainder of the transition.

Mechanism: The Elector node makes a decision about the new shape of the cluster. It must calculate the fully-connected subset of all surviving related nodes, which maximises:

- Total Votes; then,
- Total Metric (CPU capacity); then
- Total number of nodes.

This subset of nodes becomes the New Cluster Map. The Elector then determines, from that map, the maximal subset of nodes which is fully connected by non-degraded links only. This further subset is the “Functioning Subset” of the cluster, and the nodes in that subset are “Functioning” nodes. Other nodes are marked as “Degraded” nodes.

Discussion: Degraded Nodes

It will be up to individual nodes to decide which of their internal services to start, and which to suspend, on a transition between Functioning and Degraded mode. The most important reason for maintaining Degraded nodes in the cluster is to sustain Quorum: any votes belonging to a Degraded node still count towards the Quorum in a cluster, and can therefore mean the difference between life and death of critical cluster services on a cluster failure.

Consider a cluster of 5 voting nodes: a communications fault may remove high-quality access to three of those nodes but we may still have degraded links from the remaining two nodes to one of the failed nodes. In that situation, we still have 3 votes out of 5 --- enough, barely, for quorum --- as long as that degraded node remains in the cluster. However, if we remove it from the cluster (even cleanly, returning its votes), we are left with only 2 votes out of 4, and we lose Quorum.

However, in most cases, we hope that loss of high-quality communications to a node does not lose _all_ degraded communications: that’s why we have backup serial interconnects, for example. In the above example, if the 2 evicted nodes lose direct contact with the surviving cluster, they can still exit from the cluster cleanly as long as there is some multi-hop path between the nodes. In this case, Quorum management must be involved in the cluster transition. For good quorum maintenance, it is absolutely critical that we perform that eviction in a clean manner which allows the expected votes in the surviving cluster to be adjusted if necessary.

to that set of nodes any other Related nodes which can see each node in the Functioning Cluster, but which have Degraded connections to one or more Functioning

Nodes. These nodes with Degraded connections can still belong to the new cluster, but they are marked as Degraded, not Functioning, members. They are still present for sysadmin and vote tiebreaking purposes,

Error recovery: The Eviction phase is not quite as simple as previous phases with respect to error recovery, but it is not too complex to understand as long as we bear in mind exactly what sort of recovery we need to deal with. The reason that the Eviction phase introduces new complications is that up until this phase, we have not started changing the membership of the cluster (as defined by the Cluster ID at each neighbouring node). During Eviction, we are for the first time performing a controlled modification of the cluster membership.

Fortunately, we can still reuse the existing error recovery mechanisms. If we lose our Link to a Related node during this phase, we just restart the cluster transition by going back to Abort phase. This is fine. It doesn't matter exactly who in the cluster has completed Eviction and who has not. Any nodes whose eviction has completed will now have a new Cluster ID; nodes which have not will still have the old Cluster ID. The Discovery phase explicitly limits itself to discovering only Related nodes, so after any error, we already have the required recovery mechanism in place to work out who has and who has not yet completed the Eviction process.

++ Propagation Phase

This phase propagates all information about nodes still reachable in the cluster, and inter-node communication paths still working, to the CC node.

++ Proposal Phase

The CC generates a new Commit Cluster Map, and proposes that map to every node.

++ Commit Phase

The CC completes the transition by promoting the Commit Cluster Map to be the Current Cluster Map, and requests all other nodes to do the same. The cluster transition is seen by each node to be complete once this occurs: the CC sees the transition first and other nodes see it once the Commit message arrives.

Note that within each of these phases, we may have to pass information to nodes which are not necessarily connected directly to the CC. If a cluster fault has ended up producing a partially-connected set of nodes, then at least CC election information and cluster eviction information must be passed, preferably along with a new cluster map indicating to the rejected node just which machines it could not see in the cluster (for diagnostic purposes, not for correctness: you cannot base your local correctness on the state of an invisible node!).

The details of each phase are as follows:

Abort Phase: The aim of the abort phase is to make sure that every node which is a member of, or which is connected to, the cluster, has entered cluster transition state. When we first begin the abort phase, we reset every Link within the cluster. No Resets occur after this: they are not necessary, since any received communications error represents an implicit guarantee that the remote node has entered the cluster transition phase itself, anyway.

We need to be aware that the entire cluster transition has some pretty fundamental complications: we can get a new cluster transition during it. For example, a cluster merge can be brought to a screaming halt by the death of one node in one of the merging clusters. Therefore, we need to be able to reenter abort phase at any point in the cluster transition, and this new abort must also reset communication links with the new members we were trying to merge with at the time.

We make the simple rule that all Links to nodes with the same cluster name must be reset during a transition, even if those links are to nodes currently known not to be in our cluster.(Remember, a cluster partition due to partial cluster connectivity will result in two or more separate clusters from the Integration Layer point of view, but although those clusters will have different IDs, their basic cluster name will be the same.)

However, we do not have 100% synchronous cluster-wide state transitions (this job would be _so_ much easier if we did!). When we do any state transition, we cannot be entirely sure that our neighbours are currently in the same state that we are.

We may be in abort phase when our neighbours are not. They may still be in abort phase when we have already decided to exit that phase. *** @@@ I need to think about this a good deal more when the exact transition between states has been outlined in a great deal more detail than we have here. ***

From this, it also follows that a Link reset from a node not currently in our cluster must be treated as a serious error if that node is currently trying to merge with us (so we need to reenter abort phase), but if a reset arrives on a Link to a node which is not currently in the cluster due to a previous failed cluster merge, it does not necessarily cause an abort. It might, however, cause the local cluster to begin reevaluation of the partial connectivity of the Cluster Superset, initiating a voluntary Abort to start the cluster merge process.

Discovery Phase:

Election Phase:

The Worry Set

We maintain, at all times, a set of nodes called the Worry Set. This set contains a list of all nodes which we care about at the moment. A Link State change for a node in the worry set always restarts the cluster transition with a move to Abort Phase.

Why do we need this? Nodes outside our immediate cluster should not be able to kill the cluster if they die. If we have evicted a node (due to voluntary exit or due to a fault) or have rejected an attempt by that node to merge with our cluster, then that node may still have a connection to us but the state of that connection is not important to the cluster. We have already decided that the other node is not going to join in the fun, so we shouldn’t care if that node disappears.

However, things get more complicated during a cluster transition, where the Current Cluster Map and the New Cluster Map are quite distinct things. When we are in the middle of proposing a cluster merge, for example, the new prospective members of the cluster are not yet in our cluster, but if a link to one of them disappears, we most definitely need to abort the merge. On the other hand, if we have already decided that some remote node is to be evicted from the cluster during the cluster transition, and that node then dies, then we probably don’t care.

During the normal running of a cluster, the Worry Set is the set of all nodes in the Current Cluster Map.

During the cluster breakup phase, the Worry Set must be the same: during that phase we are concerned above all else in the discovery of what our old cluster now looks like, finding out which nodes and which links have survived. We are not at all interested in any other nodes at this point.

Into the merge phase, we are in a different situation. All of the nodes which contact us and try to merge with us must be added immediately to our Worry Set, so that the entire merge mechanism can be aborted and restarted if one of those neighbours dies. Once into the merge phase, we add to the Worry Set every member of every cluster which tries to merge with us. We remove them from the Worry Set if we end up deciding to reject the merge.

Once the cluster transition is complete, of course, the new Commit Map becomes the Worry Set.

LocalWords: CC ie pre neighbour ve nodename minimise optimise optimising txt
LocalWords: maximises sysadmin tiebreaking ID

CHAPTER 10

The Discovery Algorithm

A few notes about the discovery algorithm used in cluster breakup to determine the best fully-connected subset of surviving machines for forming the new cluster.

We have a value function, and we want to achieve three things:

- Decide on which nodes has the best value function;
- Propagate all connectivity maps to that node; and
- Propagate that node's cluster reconfiguration decision to the rest of the nodes, even those which are only indirectly connected to it.

Consider the network of nodes/value functions:

Example D1:
Node:A-------B-------C-------D-------E
Value: 4-------1-------3-------2-------5

The nodes B and D both see themselves as local minima for the value function and both see a local maximum nearby. How do we propagate the appropriate value functions across the network, without using an algorithm which introduces too much traffic in a still-fully-connected network?

Whatever the algorithm, we need to have a termination condition. We decide to terminate once a node is able to determine that it is the Elector, ie. that it has the best value function anywhere in the graph.

We can only make this decision at any node once that node:

- Posesses the value functions for every point in the graph, and
- Is sure that it has seen the entire graph. To know that, it must have seen the connectivity maps for every node in the network, and must have checked that every neighbour mentioned in each of those maps has also provided a map.

Every node knows, at all times, the state of each of its Links to neighbours and the names of those neighbours, so no special communications step is needed at a node to determine the node's own connectivity map.

Assume that we start the discovery algorithm by, on each node, telling each of our neighbours what our value function is. Each node then sends its local connectivity map to the neighbour with the best value function. In a fully connected cluster, the value exchange is O(N**2) and the map exchange is O(N). That's OK for now --- the RESET traffic when we do a cluster abort is O(N**2) anyway, and the value packets are small, so this is only a small incremental cost over the RESET traffic.

Now, we want to prevent any further unnecessary traffic in the common cases where the network is largely connected. We decide that only nodes which are best-so-far --- nodes whose connectivity maps indicate that it has the best value of any nodes seen so far --- may progress the discovery algorithm further.

They do so by identifying "edge" nodes in the transitive closure of the locally-held connectivity maps --- nodes which are mentioned as neighbours in the known connectivity maps but whose own maps are not yet known.

A best-so-far node proceeds by sending out routed "ping" messages to those edge nodes, using the already-known connectivity maps to provide the route. The ping messages include the sender's value function and the list of nodes which the message must pass through on the way to the edge node.

On receiving such a packet, an edge node (identified because we have reached the last hop in the route) sends back its local connectivity graph (plus any other connectivity maps it may happen to have). Every other node on the route just increments the hop count in the packet. We maintain the full route in the packet when we pass it, to allow the reply also to be routed.

For every such ping packet, each node updates its local copy of the “highest value seen” value function by maximising it with the sender of the ping. Any ping with a lower value function is dropped. As a result, if two locally-maximal nodes (A and E in example D1 above) both start the flood-fill edge expansion algorithm, intermediate nodes (like C above) will eventually detect the “better” of the two nodes (E), and will stop A’sping packets from propogating further.

Each cycle of edge expansion proceeds by the best-so-far node pinging all edge nodes and waiting for all responses. The ping dropping described in the previous paragraph ensures that this process gets terminated whenever any node detects that some locally-maximal best-so-far node is actually not the best node in the cluster.

The entire procedure terminates once a best-so-far node has no more edge nodes, and by this time we know for sure that it has found the entire connectivity graph. Other locally-maximal nodes (like A above) may still be waiting for ping responses, but when the newly-determined Elector starts the next phase of the cluster transition, those nodes will break out of that state in response to a reconfigure command from the Elector.

CHAPTER 11

Barrier Operations

Goals and discussion

Barrier functionality is commonly required in distributed systems. It provides a synchronisation mechanism by which multiple nodes can coordinate activity.

The basic definition of a barrier is that it provides a guaranteed synchronisation point in distributed processing which is valid over all nodes in the cluster: it strictly divides time into a pre-barrier and a pos-barrier phase. The barrier may not necessarily complete at exactly the same time on every node, but there is absolute guarantee that the barrier will not complete on any node unless all other nodes have begun the barrier.

Why is this useful? Sometimes in cluster operations you have some task which is broken up into phases, and you need to make sure that all nodes in the cluster have completed one phase of the task before you let anybody move on to the next phase. Cluster recovery (after a cluster transition occurs) is full of such cases: for example, when you recover a shared-data service such as the cluster namespace, you want to suspend processing of new namespace requests everywhere before you start any recovery; and you want to make sure that all nodes have completed their namespace recovery before you allow any node to start processing new namespace requests. Barriers between these phases allow such synchronisation, and it is because the cluster's basic recovery process requires barriers that a barrier API must be included in the core cluster services.

Functional requirements

There are several different things that we want barriers to be able to provide. In thinking about barrier functionality, remember:

> You cannot have a barrier without knowing in advance what the membership of the barrier is.

Say you want to synchronise the start of some work being performed in a distributed application. The definition of the barrier is that once all clients have requested the barrier, the barrier completes. However, when the first two or three processes in the application start up and request their barrier, how does the barrier API know that there are in fact more process not yet connected to the barriers which will want to participate in the barrier?

In other words, as the processes are starting up, we are faced with the possibility that all processes *currently* connected to the barrier have triggered a barrier progression, but that there are still processes which have not reached the barrier because they have not yet even registered with the barrier API.

So, we cannot progress the barrier until after we know that all users of that barrier have registered with the barrier API, *and* all registered clients of the barrier API have requested the barrier itself.

In other words, we have to have some mechanism in the API for closing off the list of participants in the barrier.

Therefore, barrier use is in two phases. First, one or more processes around the cluster register as users of a named barrier. Secondly, we perform barrier synchronisations: any process can request a barrier, and the operation only completes when all processes with an existing connection to the barrier have made the same request.

There are several different ways we may want to specify barrier membership. Most application processes will want to be able to use an API which either says "there will be exactly N members of the barrier, don't allow it to progress until that many processes have joined",

"All barrier joins have been registered, enable the barrier now", or

"Trust me, I'll not advance the barrier until all interested processes have joined, so freeze the membership list as soon as you see a barrier advance."

In addition, recovery really wants a special case:

"Exactly one process from each node in the cluster will participate in the barrier, and the barrier is destroyed on cluster transition."

Having this functionality present in the barrier API allows cluster recovery and cluster startup to share the barrier API as a way of synchronising during cluster transitions. In particular, it means that when a node starts up, the server processe implementing higher level cluster APIs can use the barrier API to ensure that recovery of the existing nodes in the cluster waits for the corresponding server processes in the new node to be up and running before the barrier is advanced.

Multi-stage recovery is important for many services: for example, in the cluster namespace, we must wait for all nodes to stall processing of new requests before we start recovering the namespace state; and we must wait for all nodes to finish that recovery before re-enabling the API processing. We can use multi-step barriers for that.

We associate a cluster-wide count with each barrier. Whenever a request is made for a barrier "NAME:<count>", the request will be blocked until all other connected barrier clients have also made a request for barriers "NAME:<n>" where n >= count. This allows some clients to miss out some stages of the barrier sequence if they want to.

CHAPTER 12

Quorum

When it comes down to it, quorum is essentially the deceptively complex busines of establishing some property which can only ever be held in at most one partition of a partitioned cluster.

Quorum is at first glance a relatively simple topic. In any cluster, we have the concept of the number of votes present in active, surviving members of the cluster; and the total number of votes (the “Expected Votes”) present in the cluster including down or unreachable nodes. If we can see an overall majority of expected votes, then we have quorum.

Unfortunately, things are not quite so simple (you _knew_ I was going to say this, didn’t you? :) There are a number of issues which complicate the whole business:

- Dynamic cluster configuration. Ideally, we would like the cluster to maintain ExpectedVotes itself, automatically. When a node first starts up, how does it know how many other nodes it has to wait for until it decides that it has established a quorum? Existing cluster implementations often solve this problme by forcing expected votes to be a static configuration variable requiring manual setup. We really want to do better than that if we can.
- Voluntary exit versus node failure. When a node loses primary communication with its peers, it may still have a backup, degraded communication link over which it can negotiate a clean, voluntary exit from the cluster. Similarly, if the sysadmin brings a node down for routine maintenance, the node can withdraw

from the cluster cleanly. In both cases, we might prefer the vote of that node to be withdrawn from the cluster in a controlled manner which adjusts the ExpectedVotes of the remaining nodes, to maximise the chance of retaining Quorum (remember, if we removea Vote, we _always_ make it harder to retain Quorum over future failures unless we also adjust the Expected Votes to compensate).

To provide a Voluntary Exit mechanism, we want to have a way by which nodes can withdraw not only their vote, but also their Expected Vote contribution from the cluster. If we allow this, the condition must be persistent: the node cannot then be allowed to rejoin the cluster and give its Expected Vote back unconditionally. If the Expected Votes could be cast back in all cases, there would be a danger that a number of nodes which had been shutdown and voluntarily exited from the cluster, might then try to reform a cluster on their own if they happen to reboot themselves into a separate partition from the original cluster.

We can deal with these concerns in the following manner:

- We will maintain a cluster-wide “quorum database” listing each node and the votes owned by that node.
- The quorum database will be persistent, and must be replicated (with a serial number for conflict resolution) on every voting node in the cluster.
- Modifications can be made to the quorum database ONLY IF QUORUM IS ALREADY HELD.

Now, if a node is removed manually from the cluster, the cluster’s expected votes can be adjusted accordingly. If a cluster partition occurs and a manually-removed new node rejoins a non-quorate partition, it is impossible for quorum to be regained accidentally, no matter how many such nodes rejoin: quorum is required before those new nodes will be able to vote.

The observant reader will note that this makes it impossible for a brand new cluster to obtain quorum. We must provide a bootstrap facility to allow the system administrator to add a casting vote to the quorum database manually before a newly configured cluster can achieve quorum. The casting vote will grant quorum (it will be the only vote in the quorum database, so votes == expected votes), and once that happens, all of the nodes in the new cluster can then register themselves in the quorum database.

Of course, if a node leaves the cluster unexpectedly, then its vote should remain in the quorum database even though the vote is no longer being cast: the quorum database simply grants a node the right to vote.

Extra votes

The case of the two-node cluster is a notorious special case for quorum. In this case, it is in principle impossible to have a symmetric quorum with the same number of votes, and still to have quorum survive a single node failure. That make failover of quorate services hard on a two-node cluster!!

The basic problem is that if a node loses contact with its partner, it has no way to be sure whether the other node is actually dead (and therefore failover should occur), or whether in fact the failure was in the connection between the two machines instead.

There are two solutions to this problem in widespread use. One is to make sure that the partner is dead by killing it, hard -– SGI's FailSafe product allows one node to deactivate the power supply on its partner, for example. The second is to add an extra, external vote of some description to act as a tie-breaker. A "quorum disk" --- a disk, usually SCSI, which is connected simultaneously to both nodes in the cluster --- is often used for this.

Any alternative quorum sources can be integrated into this quorum design. The only restriction is that the quorum source must be registered in the quorum database before its votes may be used.

This cluster design already offers one feature designed to distinguish between communications split and node failure. The use of "degraded" backup connection between nodes for the cluster integration protocol allows controlled negotiation of the eviction of a single node from the cluster if a partition occurs on the primary cluster interconnect.

Is there anything we can do if a node dies altogether, though, to recover its lost vote? The answer is strictly "no" if we cannot tell the difference between a dead node and a disconnected node.

However, if we have faith in our backup communications to the lost node, and if we can convince ourselves with good confidence that the lost node is in fact truly dead (and therefore it cannot be part of a partitioned cluster), then we can use a “casting vote” concept to recover quorum:

- Whenever true quorum (quorum without a casting vote is achieved), any casting vote present in the quorum database is removed and a single “floating vote” i registered. The casting and floating votes do not contribute towards the expected votes. The casting vote counts towards quorum votes, but the floating vote doe not.
- If any cluster transition results in the loss of true quorum but semi-quorum (we have exactly half of the expected votes) remains, AND if we can verify absolutely for sure that at least one of the lost nodes is truly dead, THEN we can convert any existing floating vote into a casting vote.
- The casting vote mechanism will also be used to “kick-start” an initial cluster a described above: manual creation of a casting vote will enable quorum to be established in a new cluster.

The judgement of whether another node is dead or not will have to be configured by the system administrator. By default, the cluster will never ever risk creating a cluster partition, and will not use a casting vote.

CHAPTER 13

Namespace Manager

The “Cluster Namespace” is a simple concept: it is a cluster-wide table of “NAME=VALUE” pairs, much like the environment variables of a standard Unix process. The namespace is a dynmic table: names do not survive a cluster reboot. Each name in the table is owned by a process in the cluster, and will be removed if that process dies or its node leaves the cluster.

The namespace intended to provide an API to query and locate all services around a cluster. It is not a service management API, in that it is not intended to provide a general interface for interaction with services.

It is, however, intended to be the single clearing-house through which all queries to locate a service are directed. For example, all printers queue servers in a cluster may register their printers in the cluster namespace under the name “PRINTER/<printer-name>=<printer-type>”. Any user can query for all “PRINTER/*” names to find the printers in the cluster, and the query reply will include the cluster node for each printer returned. It would also be possible for a printer, having been registered, to export extra information about itself such as “PRINTER/<name>/STATUS=idle” if it wished. Similarly, exported NFS directories, exported network block devices and so on can all be registered with such a namespace.

The fact that the namespace API knows about individual processes is key to making this work. An application export a name, in such a way that the name disappears if the application dies, but it is also possible for another application to query that name and set up an active dependency on it. In this case, the dependent application will receive an asynchronous notification from the namespace service if anything happens to the name in question, for example if its value changes, or it dies altogether, or its host moves from one node to another due to failover.

There is another critical property of the namespace layer: a namespace registration may be made either shared or exclusive. A shared name assignment simply means that multiple instances of the name may be present. For example, in the printer case, any number of hosts might offer a printer named "PRINTER/DEFAULT", and a print request to the default printer may appear on any of those printers.

An exclusive name assignment will only be granted to one process in the cluster at once. However, that does not mean that only one node can request the name. If two or more processes request assignment of the same name exclusively, then the first will be granted the name, and the others can stall until the name becomes available.

This provides a flexible mechanism for managing failover. A service can try register the same name on each node, and the namespace will ensure that it is granted only on node node. The request can include a preference value, in which case the node with the highest preference for that name will be granted it. However, if that node dies, the existing queued request for the name on another node will be granted, and the service on that node will be able to continue.

Any other services, on that node or on any other, which were dependent on the old name can request a callback so that if such failover occurs, they can deal with the change in service, so client requirements for failover are manageable as well as server requirements.

In practice, I expect that there will be a local failover service on each node which uses a simple scripting configuration to allow the user to set up failover groups of multiple services started in a particular order. In such cases, the use of an exclusive name "FAILGROUP/<name>" can be used to make the failover of each failover group atomic around the cluster.