Cluster Concepts

Cluster

There are many different types of clusters. Common types include:

- load sharing clusters, which divide work among the members.
- high availability clusters, where back-up nodes take over when primary nodes fail.
- information sharing clusters, which ensure the dissemination of information throughout a network.

There are so many different goals and architectures that Greg Phister (in "In Search of Clusters") came to the conclusion that it is very difficult to even define the term. About the only thing we can say for certain is that a cluster is a networked connection of nodes, all of whom agree that they are part of a cluster.

Membership

If a cluster is defined as a networked connection of nodes who consider themselves to be participants in the cluster, then obviously "membership" is a key concept.

We can distinguish two types of membership:

- potential, eligible or designated members
- active or currently participating members

This distinction is important because only active members can communicate with one-another. Thus it is that the term "membership" is most commonly used to describe only the currently participating members.

It is very important to know who the current (active) cluster members are. We may, for instance, be required to make sure that each of them has been informed of some operation before we are allowed to perform it. Cluster membership often comes with responsibilities (e.g. a guarantee to respond to certain requests within a certain period if time). Thus it is vital that we know when nodes enter and leave the cluster.

In most clusters, a node has to be explicitly configured or provisioned into the cluster ... so that the set of potential members is well known, and perhaps even closed to new members. There are, however, some types of clusters where any node is welcooe to join at any time.

Degree of Coupling

Horizontally scaled systems generally seek maximum independence between the participating nodes. If they share no resources, there should be little need for them to coordinate their activities with one another. The nodes in such systems are said to be *loosely coupled*. Loose coupling can be a very good thing:

- if there are no shared resources, there is no danger of conflicting updates from other servers. This means that each node can safely cache frequently used data, without fear that it will be invalidated by updates from other servers. This caching can greatly improve performance.
- if there are no shared resources, there is no need to synchronize their use, making the code simpler and eliminating potential bottle-necks.
- if there is little communication between nodes, they can operate completely in parallel and should provide very good scalability.
- if there is little coordination between nodes, it is unlikely that a bug or failure on one node will affect others, improving the reliability of the nodes and availability of the services.

Maintaining coherent views of changing objects, and synchronizing parallel updates from multiple writers greatly complicates systems (adding new modes of failure) and reduces parallelism (reducing performance). Sometimes, however, sharing is inevitable; Consider a database server which must service many thousands of requests per second to a single, shared, database. Distributed systems that share resources and coordinate activities with one another are said to be *tightly coupled*. The ultimate extreme might be a *single system image* clustered operating system that runs on many nodes, but shares all state and resources so perfectly that applications cannot tell that they are not all running on a single computer.

Node Redundancy

In a clustered system, work is divided among the active members. To reduce distributed synchronization, it is common to *partition* the work (e.g. desigate each server responsible for a certain subset (e.g. a file system, a range of keys, etc) of requests, and route all requests to their designated owner). In such systems, we can talk about *primaries* (the designated owners) and *secondaries* (nodes who are prepared to take over for a primary if he fails).

There are two fundamentally different approaches to take to high availability:

- Active/Stand-By
  The system is divided into *active* and *stand-by* nodes. The incoming requests are partitioned among the active nodes. The stand-by nodes are idle until an active node fails, at which point a stand-by node takes over his work.
- Active/Active
  The incoming requests are partitioned among all of the available nodes. If one node fails, his work will be redistributed among the survivors.

An active/active architecture achieves better resource utilization, and so may be more cost-effective. But when a failure occurrs, the load on the surviving nodes is increased and so performance may suffer. An active/stand-by architecture normally has idle capacity, but may not suffer any performance degradataion after a failure.

We can also look at how quickly a successor is able to take over a failed node's responsibilities. In some architectures, all operations are mirrored to the secondaries, enabling them to very quickly assume the primary role. Such secondaries are called *hot standbys*. In other systems, the secondary waits to be notified of the primary's failure, after which it opens and reads the failed primary's last check-point or journal. The former approach results in more network and CPU load, but much faster fail-overs. The latter approach consumes fewer resources during normal operation, but may take much longer to resume service after a primary has failed.

Heart Beat

Ideally nodes will announce the fact that they are joining the cluster, or are about to leave it. This is not always the case:

1. a system may crash.
2. the clustering applications may crash.
3. a node may become so busy that the clustering applications cannot run.
4. a network interface or link may fail.

Since we cannot be sure that member will notify the other members before he leaves the cluster, we need a means of detecting a member who has dropped unexpectedly out of the cluster. The most common techique is called a "heart beat". A heart beat is a message, regularly exchanged between cluster members. These may be special messages, or they may be piggy-backed on other traffic. If too much time passes without our having received a heart-beat from some node, we will assume that node has failed, and is no longer an active cluster member.

The failure of a node may (in some clusters) have serious consequences (e.g. the freeing of all resources allocated to that node, and the aborting of all in progress transactions from that node). To prevent "false alarms", many systems perform heart-beats over multiple channels, or have a back-up link with which they attempt to confirm a failure before reporting a node to be dead.

Cluster Master and Election

It is often convenient to elect or designate one node to be the cluster master:

- Coming to a mutual agreement between multiple nodes can be a complex process (e.g. Three Phase Commits). If one node is designated a cluster master, that node can serve as a central point of syncrhronization and/or control for operations in the cluster.
- Rather than requiring all nodes to heart-beat one-another, it is more economical to simply have all nodes heart-beat with the cluster master. He will detect the failure of any other node, and all nodes will detect his failure.

The election of a cluster master may, itself, be a complex process ... but having performed that process may eliminate the need for any further negotiations. There are numerous well established election/concensus algorithms. One of the best known is Leslie Lamport's Paxos algorithm.

Split Brain

A pathological network failure might divide a cluster into multiple sub-clusters, which cannot communicate with one-another. Such an event is sometimes referred to as a "partitioning" of the network. If the cluster manages critical resources (e.g. a database or nuclear warhead), it is possible that the independent sub-clusters will all continue operating and make independent but incompatible decisions. Such a condition is called "split-brain" (as if two halves of our brain were working independently and at cross-purposes).

There are two standard approaches to preventing "split-brain":

a. quorum
b. voting devices

Quorum

If there are N potential members in a cluster, we can build in a rule that says a cluster cannot form with fewer than (N/2)+1 members. This acomplishes two purposes:

- It makes it impossible for any partitioning to result in two viable sub-clusters (because N nodes cannot be divided into two groups that both contain at least (N/2)+1 nodes).
- It ensures that any decision made (and persisted) by this quorum will be remembered by any future quorum (because any group of (N/2)+1 nodes will have at least one member in common with every other group of (N/2)+1 nodes that has ever existed.

The problem with using a numerical quorum is that if (N/2)+1 nodes have been damaged, it will be impossible for the surviving nodes to form a new cluster ... even if there is no split-brain.

Voting Devices

If there is a single piece of hardware in the cluster, that must be present for the cluster to function, and that can only be owned by one node, that device can be used as a voting device.

Consider, for instance, a shared disk. If that disk is absolutely required to provide service, a node that no longer has access to that disk cannot provide service (and hence is not eligible to form a cluster). But what if two nodes can both talk to the disk, but cannot communicate with one-another? They may be able to use the disk as a voting device ... e.g. by writing a recent time-stamp into a well-known block.

Some clusters include resources that can easily serve as voting devices. There are also specially built (very reliable) voting devices that exist solely for this purpose. If there is a voting device, a cluster could be formed by a single node ... because the voting device would prevent split-brain.

Fencing

What if, you were not only sufferring from schizophrenia, but the other side of your brain had actually gone rogue, and was trying to commit acts of mayhem against you and others? In some clustered systems, it must be assumed that if a node has fallen out of the cluster, he is no longer trust-worthy ... and must be "fenced-out" of the cluster. There are two common approaches to fencing:

- reservable devices
  Some devices can be told which interface to listen to, and not to listen to the other interface. This is often done with dual-ported disks. The node that has seized the quorum device will then instruct the quorum device not to accept commands from any other node.
- remote power control
  Some clustered systems come with remote power controllers, and a node that has seized control of the cluster from a previous (apparently failed) cluster master will often power-off or reset the previous master, to ensure that he does not continue to vie for control of the cluster and its resources.