OSDL Cluster Architecture

by Joe DiMartino <joe@osdl.org>,
John Cherry <cherry@osdl.org>
and Daniel McNeil <daniel@osdl.org>

May 1, 2003

Introduction

There are many alluring aspects of clustering technology. A cluster can supply scalable performance much greater than a single computer can provide at a fraction of supercomputer prices. Some clusters provide close to continuous availability for services and applications. Ideally a single cluster could provide high performance and high availability.

Greg Pfister defines a cluster as *"a parallel or distributed system that consists of a collection of interconnected whole computers that are utilized as a single, unified computing resource."*[Pfi98] There are literally hundreds of variations of interconnected multiple computers that could be called a "cluster". The phrase "*single, unified computing resource*" conjures up a wide variety of possible applications and uses, and is purposefully vague in describing the services provided by the cluster.

At one end of the spectrum a cluster is nothing more than the collection of whole computers available for use by a sophisticated distributed application. At the other end the cluster creates an environment where existing (dusty deck) non-distributed programs can enjoy increased availability due to cluster wide fault masking and increased performance due to increased computing capacity.

Within that spectrum, there are many similarities in the software components used. Various clusters may change the way components are interconnected, strengthen or weaken a few protocol guarantees, alter the algorithms slightly to match the scalability scope, or export different programming interfaces, but the same basic principles of clustering remain unchanged.

It is relatively straightforward to write network applications that span multiple computers. It is difficult to write a *correct* distributed application in the face of asynchronous failures. For this reason, many applications rely on a trusted

set of programming libraries that help to abstract away the complications of the distributed environment. It is a goal of some clusters to provide a common programming framework for the creation of new distributed applications.

This paper focuses on the software technology needed to make multiple computers cooperate correctly and efficiently so they can be utilized as a single, unified computing resource.

Cluster Software Components

The composition of any particular cluster is driven by the needs of the services provided and by the applications that use those services. The availability of the system is largely determined by the reliability of the underlying components. This architecture presents high availability through redundancy and fault masking. It encourages the creation of distributed applications to increase performance and minimize recovery time, yet provides an environment where non-distributed applications enjoy the benefits of increased availability.

Building a distributed application in this network-savvy world gets increasingly easier, yet it remains a challenge to achieve correctness in the face of system failures or other asynchronous network failures.

Each component provides certain assurances that will simplify the implementation of subsequent layers by abstracting and hiding certain complexities. This provides an optimistic view that things are working more often than not and eliminates duplicate code for dealing with lower layer faults. In order to take advantage of this, each layer must mask faults where possible and provide timely notification of failures.

The software layers are presented here from the bottom up to emphasize the value provided to upper layers. The diagram in Figure 1 on the following page shows conceptually the dependencies of the multiple software layers.

1 Cluster Connectivity

The basis for any lasting relationship is good communication. This also applies to clusters. Start with standard networking technology, coalesce redundant connections into virtual circuits, then provide high bandwidth, low latency, ordered, reliable point-to-point transfers. Use redundant paths to load balance message traffic and mask faults. Monitor the health of each individual nexus and report failed VC when no path exists.

Cluster connectivity requires a strict tolerance on the responsiveness of the network and the ability to detect a path failure in milliseconds. The scope of

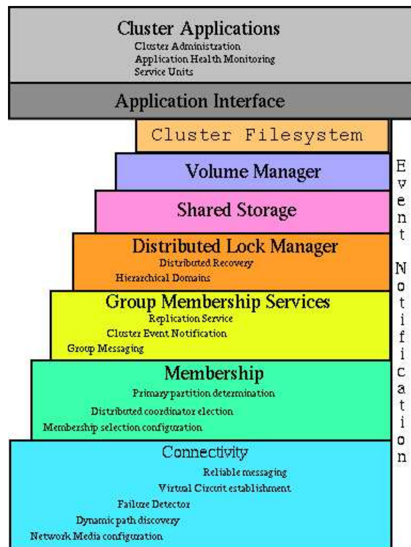

Figure 1: The layering represents a logical separation and each section suggests (not mandates) a code module division.

connections is typically confined to a one hop distance. It should be possible to provide reliability with minimal performance implications. For these reasons, a lighter-weight transport protocol than TCP is desired.

1.1 Network Media Configuration

Building upon standard IP (UDP/IP?) allows most network media to be considered as a choice for cluster messaging traffic. Any high bandwidth, low latency media will work, but multicast-capable media is preferred for obvious optimizations. The *network media configuration* layer hides this detail and exposes a set of IP addresses and subnet masks.

1.2 Dynamic Path Discovery

Using a *hello protocol*, nodes can broadcast messages on each of the configured network interfaces and collect the responses. In like fashion, responses are presented to valid incoming hello broadcasts. Connection paths are then established based on successful hello message exchanges. A path represents an individual point-to-point connection between two nodes. It is expected that multiple paths will exist for redundancy and load balancing.

1.3 Failure Detector

After a path has been discovered, it needs to be periodically monitored to avoid latent failures [CHT96]. This can be done in due course as messages are load balanced across each path, but must be done gratuitously during idle times. A hello protocol packet can be piggybacked on already outbound messages or sent alone during idle times. This differs from the hello protocol because these messages are multicast to specific known paths and not broadcast.

The frequency of these exchanges can vary as a function of the specified failure tolerance. Making this interval too short may result in false failures, as it is impossible to distinguish between an arbitrarily slow response and a crashed path. False suspicions must be acted upon to ensure liveness.

1.4 Link Events

Whenever there is a change in paths to other nodes as new paths to nodes are discovered or old paths are determined to have failed, the new link information is made available to link event subscribers. See Section 4 for more information about events.

1.5 Virtual Circuit Establishment

As each individual point-to-point path becomes stable, it is a candidate for inclusion as part of a virtual circuit. A virtual circuit represents the communication nexus between two distinct nodes. There will be at most one virtual circuit for any pair of nodes, although each circuit may consist of redundant or alternate paths.

A virtual circuit will be addressed by a unique name that represents an end point. Load balancing based on performance metrics and reliability is done here.

1.6 Reliable Messaging

The IP layer is responsible for fragmentation and reassembly of packets that exceed a media Maximum Transmission Unit size. This session layer protocol is responsible for proper sequencing of whole messages. Informally, *reliable messaging* implies there will be no gaps in an ordered message sequence between two virtual circuit end points and there must be no duplicate messages. In addition, messages will be automatically retransmitted until acknowledged by the intended receiver or until a failure detection notice is returned for the virtual circuit.

1.7 Connectivity Events

Whenever there is a change in the connectivity as new virtual circuits are established to new nodes or previous virtual circuits are lost to nodes now unreachable, the new connectivity information is made available to connectivity event subscribers. This connectivity event provides the connectivity information: a list of nodes currently having established virtual circuits, the list of nodes which have been added (new), and the list of nodes which have been lost. Section 4 describes the way events are sent and received.

2 Cluster Membership

A cluster without membership is just a bunch of computers.

Using the connectivity as described in Section 1, the cluster membership component can use the information provided by the connectivity event in subsection 1.7 and the messaging in subsection 1.6 to communicate with the other nodes and determine an "active node membership." Every active node in the resulting membership has the same view of membership.

The steps used to to create and maintain a membership are described below. Each of the protocols described must be designed to tolerate failures, since failure can occur at any time. If a membership change occurs in the course of determining a new membership, the membership can abort and restart the membership algorithm.

2.1 Membership Selection Configuration

It might be desirable for multiple disjoint "clusters" to share a network backbone, or to provide a gateway to another cluster. To illustrate this, imagine a high availability cluster that provides a dispatching service for a set of compute nodes configured as a high performance cluster. The dispatchers will intersect both cluster memberships.

2.2 Distributed Coordinator Election

When new virtual circuits are established or lost, the connectivity event signals a possible addition or loss to the existing membership. Extra compute resources can lead to a redistribution of the current workload and in many cases the transfer of up-to-date state. This addition of new members must not upset the integrity of an already functioning membership.

In a fully symmetric cluster, all potiental members with a valid virtual circuit will participate in the distributed computation to determine membership.

The membership protocol can be efficient if multicast messaging is available to exchange the needed information between all the nodes.

If efficient multicast is not available, the membership protocol can choose coordinator to manage the membership transactions. This will minimize the messaging requireed for all nodes to exchange the needed information. The method to obtain a coordinator can be simple as long as all nodes participating in the election are aware of the selection criteria, for example, the first member of the current membership could be chosen as sorted by node id, lexical sort of node name, or relative age of members.

2.3 Partition Agreement

The function of the elected coordinator is to suggest a membership, collect opinions about the suggestion from the peers and to finally install the membership. The criteria for membership will vary for different clusters. A common method for determining membership in a High Availability cluster is full connectivity. The coordinator must also collect connectivity information from peers and determine the largest set with N-way connectivity.

Many High Performance clusters have a predefined configuration that appoints a master (or head) node. The compute nodes usually have no say in the configuration and can participate in the workload as cluster members as long as they have connectivity with the master. Full connectivity with every other node in the cluster is not required in this environment.

2.4 Primary Partition Determination

No matter what clustering style is used, there is some software algorithm or hardware/software combo used to determine the "active node membership" and every active node shares the same view of membership.

Not all clusters use "quorum", so the term *primary partition* is used to avoid the whole quorum notion. It's an implementation detail of the membership algorithm.

Now that a sub-cluster component has been formed and the constituent members agree it is stable, determine if this component is the primary one. There can be at most one primary partition in a cluster. Many distributed services need this guarantee for correctness. Only members of the primary partition may access shared resources.

A non-primary partition is usually a transient state in some part of a cluster with (temporarily) broken comms. The life expectancy of a non-primary partition is questionable, although usually very short. Some clusters may Shoot The Other Non-primary-nodes In The Head, others will take their own life after some no-activity timeout, and some have nothing shared to corrupt so they simply wait in

isolation for news from their master or for manual intervention. And sometimes an outage is short enough that, although detected, corrective action is skipped.

In any case, it is the particular cluster and its membership algorithm that determines the primary-ness or non-primary-ness of certain nodes. It IS integrated into a cluster's membership algorithm and impossible to separate. The criteria that determine primary-ness varies based on the services that the cluster in question is trying to provide and on the topology of the shared resources.

An application that wants to run on a cluster is at the whim of the cluster - not the other way around. This is not to say that one style of clustering is better than another, only that they are different, and maybe more suitable for certain kinds of applications.

If more than one partition is designated as "primary", there is either more than one cluster, or no cluster at all. This is not to say that a non-primary partition can do no work, only that the work in isolation might be in conflict with work being done elsewhere in the cluster. Processes running in a non-primary partition must be willing to be restarted or merged with the work done in the primary partition.

2.5 Membership Events

Whenever there is a change in membership as determined by the cluster membership component, the new membership is made available to those interested. This cluster membership event provides the membership information: who is in the new membership, who has been added, who has been lost, and if this membership is primary. Section Section 4 describes the way events are sent and received.

3 Group Messaging Services

A group messaging service maintains the membership of processes that join and leave dynamically during the lifetime of a group. The topic of group messaging has been explored in great detail and used in systems like Horus [RBM96] and Transis [DD96] for creating an environment of virtually synchronous process group computing [Bir97]. In essence, *virtual synchrony* provides automatically managed membership for application programs, which are permitted to join and leave groups, membership changes are synchronized with the sending and delivery of messages and state transfer is implemented to appear atomic with respect to membership changes. Those semantics make this solution powerful enough to permit correct replication of data [Bir00].

3.1 Group Creation

Group creation is dynamic creation based on name (consistent name space in the cluster).

Group services can be provided in any sub-cluster partition, but the namespaces will eventually be merged into a common namespace in the primary partition, forcing each member of every group in non-primary partitions to re-join.

3.2 Message Ordering and Stability

Placing join and leave messages in the message stream as ordered messages is a powerful concept for distributed programming. This simplifies many protocols by making the failures atomic with message delivery.

Various message ordering protocols can be implemented to simplify the task of distributed programming.

4 Event Notification

Spanning all cluster software layers is an event notification service which provides a unified way to publish events to interested subscribers. The event service uses a publish/subscribe model, where one or more publishers can post events that will be delivered to all subscribers interested in (subscribed to) events about certain topics.

In this model, event data that passes through the event service is opaque to the event service itself and only makes sense in the context of a particular event topic. Event topics are administratively created. Some topics will be pre-created to align with common architectural components of a cluster.

4.1 Event Message Format

Event data that passes through the event service is opaque to the event service itself and only makes sense in the context of a particular event topic. All events share a common message format that includes an event header, some event properties and the actual event data.

4.2 Event Subscribe

An event API is provided to subscribe for notification of events in any topic. Events are delivered only for event topics with an active subscription.

4.3 Event Publish

The cluster software components are responsible for generating events and subsequently publishing these events to the event service for further distribution to all subscribers.

4.4 Cluster Events

- Link Events (see 1.4)
- Connectivity Events (see 1.7)
- Membership Events (see 2.5)

5 Replication Service

TBD.

6 Distributed Lock Manger

The basis for peaceful coexistence of cooperating parties is the ability to share well. Cooperating applications within a cluster must synchronize access to shared resources to avoid corruption of those resources. A distributed lock manager (DLM) provides advisory locking services used to coordinate access to any arbitrary shared resource.

A DLM does not enforce good sharing behavior, but rather provides cooperating applications with information regarding the state of a shared resource. All locks are advisory, that is, voluntary. The system does not enforce locking. Instead, applications running on the cluster must cooperate for locking to work. An application that wants to use a shared resource is responsible for first obtaining a lock on that resource before attempting to access it. Applications and services that can benefit from using a distributed lock manager are transaction-oriented, such as a database or a resource controller or manager. It can also be used for system services such as a distributed filesystem for shared files or meta-data.

6.1 Programming Models

A programming model for distributed locking that has stood the test of time is the one first popularized on the VAX Cluster [VCP93]. This model then became widespread on proprietary Unix(tm) after Oracle required its usage for Oracle Parallel Server and its predecessors [OLM94]. This model provides a rich set of locking modes and both synchronous and asynchronous execution. The DLM locking model supports:

- Six locking modes that increasingly restrict access to a resource
- The promotion and demotion of locks through conversion
- Synchronous completion of lock requests
- Asynchronous completion through asynchronous system trap (AST) emulation [1]
- Global data through lock value blocks

In addition to the traditional programming model described above, a System V Unix programming model is available in the openDLM source released by IBM [DLM01].

6.2 Lock resources

The lock manager defines a lock resource as the lockable entity. The lock manager creates a lock resource the first time an application requests a lock against it. A single lock resource can have one or many locks associated with it. A lock is always associated with one lock resource. The lock manager provides a single, unified lock image shared among all nodes in the cluster. Within this cluster-wide lock image, the lock manager maintains one master copy of each lock resource. This master copy can reside on any cluster node. Initially, the master copy resides on the node on which the lock request originated.

The lock manager maintains a cluster-wide directory of the locations of the master copy of all the lock resources within the cluster. The lock manager attempts to evenly divide the contents of this directory across all cluster nodes. When an application requests a lock on a lock resource, the lock manager first determines which node holds the directory entry and then reads the directory entry to find out which node holds the master copy of the lock resource.

By allowing all nodes to maintain the master copy of lock resources, instead of having one primary lock manager in a cluster, the lock manager can reduce network traffic in cases when the lock request can be handled on the local node. Handling the requests on the local node also avoids the potential bottleneck resulting from having one primary lock manager and reduces the time required to reconstruct the lock database when a failover occurs.

The DLM expects to operate in a cluster in conjunction with another cluster infrastructure environment that provides a consistent view of cluster membership (all nodes agree on cluster membership) and node liveness (the node is a healthy part of the cluster). Reliable cluster messaging is also required from the cluster infrastructure.

[1] In the VAX OS an *AST* is a routine that can be associated with a particular event and is invoked by the operating system when that event occurs. In Unix, this is commonly known as a signal. POSIX defines a fixed set of signals which includes only two locations for user-defined events. Many system programmers create alternate mechanisms for associating routines with events to avoid collision with existing application signal usage and semantics.

6.3 Hierarchical Domains

TBD.

6.4 Distributed Recovery

When a node fails, the lock manager instances running on the surviving cluster nodes release the locks held by the failed node. The lock manager then processes lock requests from surviving nodes that were previously blocked by locks owned by the failed node. In addition, the other nodes re-master locks that were mastered on the failed node.

7 Shared Storage

7.1 Cluster-wide Device Names

TBD

7.2 Multipath I/O

TBD

8 Volume Manager

8.1 Device Mapper

TBD

8.2 RAID Volumes

TBD

9 Distributed (Cluster) Filesystem

Having shared storage in a cluster gives location independence to even simple applications.

9.1 Served Access

TBD

9.2 Concurrent Access

TBD

Open Source Projects Considered

With all the open source projects available in various clustering "foundries", at least two things have become evident. First, this must be a hard problem because there are no acceptable alternatives that completely fulfill this architecture. The second, there are a myriad of choices for cherry picking the best technologies to satisfy each layer in this architecture.

This could be an exhaustive listing of all the applicable open source projects, but instead it is a pruned list containing only the packages on the short list...

References

[Pfi98] Greg Pfister, "In Search of Clusters", Second Edition, Prentice Hall PTR, 1998

[CHT96] Tushar Chandra, Vassos Hadzilacos, Sam Toueg, "The Weakest Failure Detector for Solving Consensus", June 1996

[DD96] Danny Dolev and Dalia Malki, "The Transis Approach to High Availability Cluster Communication", Comm. of the ACM, Vol 39, No. 4, April 1996, pp. 64-70

[RBM96] Robbert van Renesse, Kenneth P. Birman, Silvano Maffeis, "HORUS: A flexible Group Communication System", Comm. of the ACM, Vol 39, No. 4, April 1996, pp. 76-83

[Bir97] Kenneth P. Birman, Building Secure and Reliable Network Applications, Manning Publishing Company and Prentice Hall, 1997

[Bir00] Ken Birman, et. al, The Horus and Ensemble Projects: Accomplishments and Limitations, circa 2000

[VCP93] Roy G. Davis, "VAX Cluster Principles", Digital Press, 1993

[OLM94] Chuck Simmons, Patty Greenwald, "Oracle Lock Manager Requirements", Oracle Corportation, July 1994

[DLM01] Kristin Thomas, "Programming Locking Applications", IBM Corporation, 2001