Mission Critical Linux
Kimberlite Design Specification

Authors

Tim Burke (burke@mclinux.com)
Ron Lawrence (lawrence@mclinux.com)
Jeff Moyer (moyer@mclinux.com)
Greg Myrdall (myrdall@mclinux.com)
Richard Rabbat (rabbat@mclinux.com)
Brian Stevens (stevens@mclinux.com)
Dave Winchell (winchell@mclinux.com)

Last Revision: June 28, 2000

Mission Critical Linux, LLC
100 Foot of John Street
Lowell, MA 01852
Tel: 978−446−9166 Fax: 978−446−9470

Mission Critical Linux
Kimberlite Design Specification

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of

the principal authors of the Document (all of its principal authors, if it has less than five).

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the
Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the
Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of

Table of Contents

1Purpose....13
2Goals....13
3Major Design Components....13
    3.1Host Membership....13
        3.1.1Node Self Monitoring....14
        3.1.2Monitoring Other Cluster Members....14
            3.1.2.1Monitoring via Shared State Partition....14
            3.1.2.2Heartbeat Pinging....14
            3.1.2.3Mechanism to Break Out of Inconclusive States....14
    3.2Shared Storage....15
4Design Approach....16
    4.1Design Pros/Cons....19
5Messaging Subsystem....21
    5.1Overview....21
    5.2Design Goals....21
    5.3Message Service Internals....21
    5.4Conclusion....22
6Logging Library....22
    6.1Overview....22
    6.2Internals....22
    6.3API....23
7Service Manager....23
    7.1Overview....23
    7.2Design....23
        7.2.1Shared Service Information Disk....23
            7.2.1.1Service Identifier (ID)....23
            7.2.1.2Service Owner....24
            7.2.1.3Service States....24
            7.2.1.4Service Information Locking....25
        7.2.2Initialization....25
        7.2.3Cluster Node State Change Events....25
        7.2.4Service Placement Policy....25
            7.2.4.1Preferred Node....25
            7.2.4.2Relocate when Preferred Node Joins the Cluster....26
            7.2.4.3Service Start Arbitration....26
        7.2.5Service Start and Stop Operations....27
            7.2.5.1Start and Stop Failures....27
    7.3Service Resources and Configuration File....27
8Heartbeat Mechanism....32
    8.1Introduction....32
    8.2Design Considerations....32
    8.3Implementation Details....32
9Shared State Disk Partition....33
    9.1Where to Place the Shared State Disk Partition?....33
    9.2Shared Disk Protections....35

9.2.1Software Protections....35
9.2.2Hardware Protections....36
9.3Quorum Daemon....36
9.3.1StartQuorumDaemon() {....36
9.3.2QuorumdBody() {....36
9.3.3StopQuorumd() {....37
9.3.4checkPartnerActive() {....37
9.3.5shootPartner() {....37
10Power Switch Options....38
10.1Background....38
10.2Purpose....38
10.2.1Observations:....38
10.3Assumptions....39
10.4Example Implementation: The RPS−10 Switch Behavior....39
10.5Power Switch Related Failure Scenarios....40
10.5.1Scenario A 40
10.5.2Scenario B....40
10.5.2.1Scenario B1....40
10.5.2.2Scenario B2....40
10.5.2.3Scenario B3....41
10.5.3Scenario C....41
10.5.4Scenario D....41
10.5.5Scenario E....41
10.5.6Service Availability Disruptions....41
10.6Implementation....41
11Dumps and Panics....42
12Cluster Startup/Shutdown Scripts....43
12.1Cluster Start....43
12.2Cluster Stop....44
12.2.1Additional Information and Particularities....44
12.3Cluster Restart....45
13Failure Scenarios....45
13.1Scenario: Two Nodes Up − Pull SCSI Cable from One Node....45
13.2Scenario: Two Servers Up − Pull Cluster Shared State Disk....45
13.3Scenario: Split−Mirror Scenario: Mirrored Shared State Partition....46
13.4Scenario: Network Cable Pull(s) When Running Cluster....46
13.5Scenario: Two Node Boot: Full Network Outage or All Hardware Fully Operational....47
13.6Scenario: Server Hangs....47
13.7Scenario: Server Panics....48
13.8Scenario: Cluster Daemon Dies....48
13.9Scenario: Site Power Outage....48
13.10Scenario: Planned Maintenance (TBD)....49
13.11Scenario: Clean Shutdown....49
13.12Scenario: Storage Outage − Data Disk....50
14Reference....51
15Appendix A. Messaging Subsystem API....52

15.1Server−Side Routines........55
15.2Client−Side Routines........56
16Appendix B. Power Switch API........57
16.1Discussion........57
17Appendix C. Disk Locking Subsystem........59
17.1Overview........59
17.2API........59
17.2.1Locking Primitives........59
17.2.2Callers........59
17.3Locking Algorithms........59
17.3.1Node Lock........59
17.3.2Disk Lock........59
17.4Scalability Implications........60
18Appendix D. State Diagram for Service Manager........61
18.1Complete State Diagram........61
18.2Service Start Arbitration........62
18.3State: Stopped, Event: Host Down........63
18.4State: Stopped, Event: Host Up........63
18.5State: Stopping, Event: Host Down........64
18.6State: Stopping, Event: Host Up........64
18.7State: Starting, Event: Host Down........65
18.8State: Starting, Event: Host Up........65
18.9State: Running, Event: Host Down........66
18.10State: Running, Event: Host Up........66
18.11State Diagram for Service Takeover........67
18.12State Diagram for Giving up Service........68
19Appendix E. Design Notes on Disk−Based Configuration Database........69
19.1Problem Statement........69
19.2Implementation Options........69
19.3Implementation Approach........69
19.4Low−Level Implementation........69
19.5Higher−Level Implementation........70
19.6Usage Instructions........70
20Appendix F. Design Alternatives........72
21Appendix G. Lexicon........73

1 Purpose

The purpose of this specification is to describe the architectural model for Kimberlite, a cluster implementation to be offered by Mission Critical Linux, Inc. The specification focuses on the major design topics at a high level. Limited historical background information is provided to capture the motivation for certain design decisions.

2 Goals

The project goals are:
1. Provide high availability for a targeted solution space.
2. Ensure data integrity.
3. Use commodity hardware.
4. Fast time−to−market.

We will initially focus our efforts on deploying highly available NFS servers and database servers. Of secondary priority will be support for highly available "production services" such as FTP, telnet, Samba, DNS, mail and print services.

3 Major Design Components

In order to provide a cluster implementation, the main design challenges are:

1. Host membership: this involves knowing which nodes are considered to be cluster members.
2. Shared storage: infrastructure to ensure that the disk devices are accessed by only one server at a time; and that I/O requests cannot be initiated in an unsynchronized manner.
3. Services Infrastructure: ability to define high availability services and provide a means of having them stopped and started in response to cluster state transitions.
4. System Management: allowing the specification of configuration and tuning parameters, as well as monitoring current operational status.

The following sections highlight the challenges of each of the above major design components. While the fundamental goals of each of these components may appear to be simple, there are deceptively complex issues involved in assuring correct operation.

3.1 Host Membership

Host membership is implemented as a peer algorithm. This approach is preferred as it is problematic to develop a centralized algorithm. The following are the key building blocks of the host membership algorithm:
1. Node self monitoring
2. Each node monitors other cluster members
3. Mechanism to break out of inconclusive states.

Each of these components is detailed in the following sections.

3.1.1 Node Self Monitoring

As detailed later, a foundation component is the "Quorum" disk partition (also referred to as shared state disk) used to represent a node’s status. On a frequent basis, each node updates its own timestamp on the shared state partition, as well as monitors the partner’s state. If a node is unable to access the shared state partition (i.e. SCSI cable pull), that node will take itself out of the cluster by rebooting itself. Inability to access the shared state partition at cluster startup time will cause a node to not commence cluster operation.

Each node also periodically monitors the status of the remote power switch (detailed later). This affects policy decisions as to whether services can be safely failed over.

3.1.2 Monitoring Other Cluster Members

Each node monitors the status of other cluster members in order to determine when service state transitions are warranted. Example service state transitions include:

- Cluster node startup
- Cluster node shutdown (planned and unplanned)

Based on a node’s state transitions, services will be started up and balanced according to placement policy.
There are two means of monitoring the state of other members:

- Monitoring status information in the shared state partition
- Heartbeat pinging over Ethernet and serial ports.

3.1.2.1 Monitoring via Shared State Partition

Each node has specific areas on the shared state partition where it represents its current state (UP/DOWN) as well as a timestamp field that gets updated periodically. A node whose state is UP and fails to update its timestamp within a specified grace period will be considered failed and forcibly removed from the cluster via a remote power cycle. In this manner, the shared state partition is the cornerstone of the host membership algorithm. The cluster will still remain operational even in the event of outage of all heartbeat channels.

3.1.2.2 Heartbeat Pinging

Cluster nodes also monitor each other through a set of "heartbeat channels", which can include any number of Ethernet connections (both LAN−based as well as point−to−point). Additionally, we support point−to−point serial connections as heartbeat channels.

The heartbeat pinging algorithm serves as policy input to the host membership algorithm. In this model, the redundant heartbeat channels independently monitor connection status. The status of each channel is then OR−ed together to surmise if a node appears active from a heartbeat perspective. The heartbeat derived node status is queried in the event that the "Monitoring via Shared State Partition" suspects the other node to be failed.

3.1.2.3 Mechanism to Break Out of Inconclusive States

The most problematic cluster scenario is the case of a hung node. In this situation, a node would fail all monitoring checks (both shared state partition and heartbeat pinging). Here the surviving cluster member would takeover services formerly provided by the hung node. At this point, the hung node could become "un−hung" at any time and issue I/O operations to the shared storage disks associated with highly available cluster services (i.e. NFS exported filesystems or databases). Such uncoordinated I/O access to the same partition results in data corruption. Therefore it is critical that one definitively knows that a failed node is not at risk of emitting I/O operations before services can be safely failed over. This is referred to as "I/O Barrier" (also known as "I/O Fencing").

The mechanism we have chosen to protect against a failed node from emitting I/O operations is to use a power switch that is controlled by a serial connection. This allows each node to power cycle the other in the event of a failure scenario.

3.2 Shared Storage

Clustered web servers which support "static content" are well suited to the current LVS (Linux Virtual Server) [3] project. In this model, each cluster member has a copy of effectively read−only data which is returned to web clients. For e−commerce sites, static content support is deficient as they have dynamic content. An example of dynamic content would be an online shopping site. This requires that all of these servers of dynamic web content interact with a common back end data store.

The back end data store can be implemented by a variety of approaches, such as:

- NFS
  - Pros: NFS support exists and is relatively mature in Linux.
  - Cons: NFS is not a very optimal algorithm. It provides weak client−side coherency. However for relatively low to moderate transaction rates its performance may be acceptable.
- File Copying − Via approaches such as **`rdist`**, and **`rsync`**, you can make a copy of your transaction data over to other cluster members.
  - Pros: the technology exists today and is mature
  - Cons: Performance is unacceptable to handle anything other than trivially small transaction rates. Basically only useful for updating the static content on an infrequent basis.
- Database − Examples include mySQL, PostgreSQL, Oracle, Sybase and IBM DB/2.
  - Pros: when going through the raw device it is performance optimal by avoiding the filesystem buffer cache.
  - Cons: there can be only one cluster member at a time which is the database server. Other cluster members could be backup servers to takeover in the event of failure of the active server.

The Kimberlite implementation provides a high availability framework for coordinated access to shared storage. Shared physical storage is required for the cluster shared state partition and is also used for highly available cluster services. The cluster implementation provides primitives which are utilized by services requiring access to shared storage. Highlights include:

- Service start/stop infrastructure guaranteeing that a service is only running on one

node at a time.
- Filesystem mount and unmount mechanisms associated with service start/stop. This includes escalating levels of "forced unmount".
- A low−level lock synchronization mechanism layered on top of the shared state disk primitives. This is used by the service start/stop infrastructure to protect against inherent race conditions.

The driver requirements for shared storage are simply the ability to support multi−initiator configurations. Our initial efforts will focus on using shared SCSI buses. There is nothing inherent in the architecture which would preclude qualifying on FibreChannel.

We do not rely on SCSI reservations or any other low−level storage substrate mechanism to reserve devices, or otherwise try to block I/O operations from failed nodes. Historically, usage of SCSI reservations has proven problematic in terms of device qualification. While there are inherent benefits to the finer granularity that SCSI reservations could afford, we intentionally wanted to avoid as many hardware dependencies as possible in an attempt to produce an expedient and reliable implementation.

A beneficial side−effect of not relying on SCSI reservations is that it enables us to have a single physical disk being associated with more than one highly available cluster service. Contrast that to SCSI reservation based schemes whereby separate partitions on the same disk cannot be associated with more than one independent service as typically SCSI reservations are at the device level (not partition level).

4 Design Approach

The main solution attributes for our implementation are as follows:

- Focusing on a 2−node solution where it is required that a service be provided by a single node at a time. Shared storage for cluster services. (We will separately consider the LVS−like situations where it is acceptable to have multiple servers concurrently. Consequently, LVS may be appropriate for web servers, while this single server infrastructure would be appropriate for Database or NFS servers.)
- Redundant communication paths (serial, SCSI and Ethernet) used to assist in determining node membership.
- Shared disk partition used to maintain shared state such as who’s−serving−what as well as acting as a quorum device for node membership. Shared disk redundancy is made possible via redundant multi−ported RAID controller.
- Each cluster node is plugged into a power switch with a serial port connector. The serial port connector is connected to the other cluster node so that it can shoot the other cluster node as an I/O Barrier. Note: it is assumed that each cluster member and the remote power switch it is controlling will be on the same power (strip). This way if someone powers down node 1, then it won’t be powering down the power switch that node 0 is controlling. In addition, it is assumed that each cluster member (and its power switch) will be on a separate electrical circuit.
- Design Assumption: An assumption is being made that support for multiple services is

a requirement. The reason the multiple service distinction becomes important is because it adds complexity over a single service model. Examples of the areas of complexity introduced with multiple services are:

- Service Load balancing. (Particularly as nodes come and go.)
- Shooting a failed node. When you have a single service model, if one node determines that the other has failed it can safely shoot the other node prior to taking over the service. For cases where you have more than one service, the shooting scenario is more complicated to handle in an optimized manner. This is due to the fact that 2 nodes could each be servicing different services and shooting down the other node to safely bring up Service 1 would also take down Service 2.

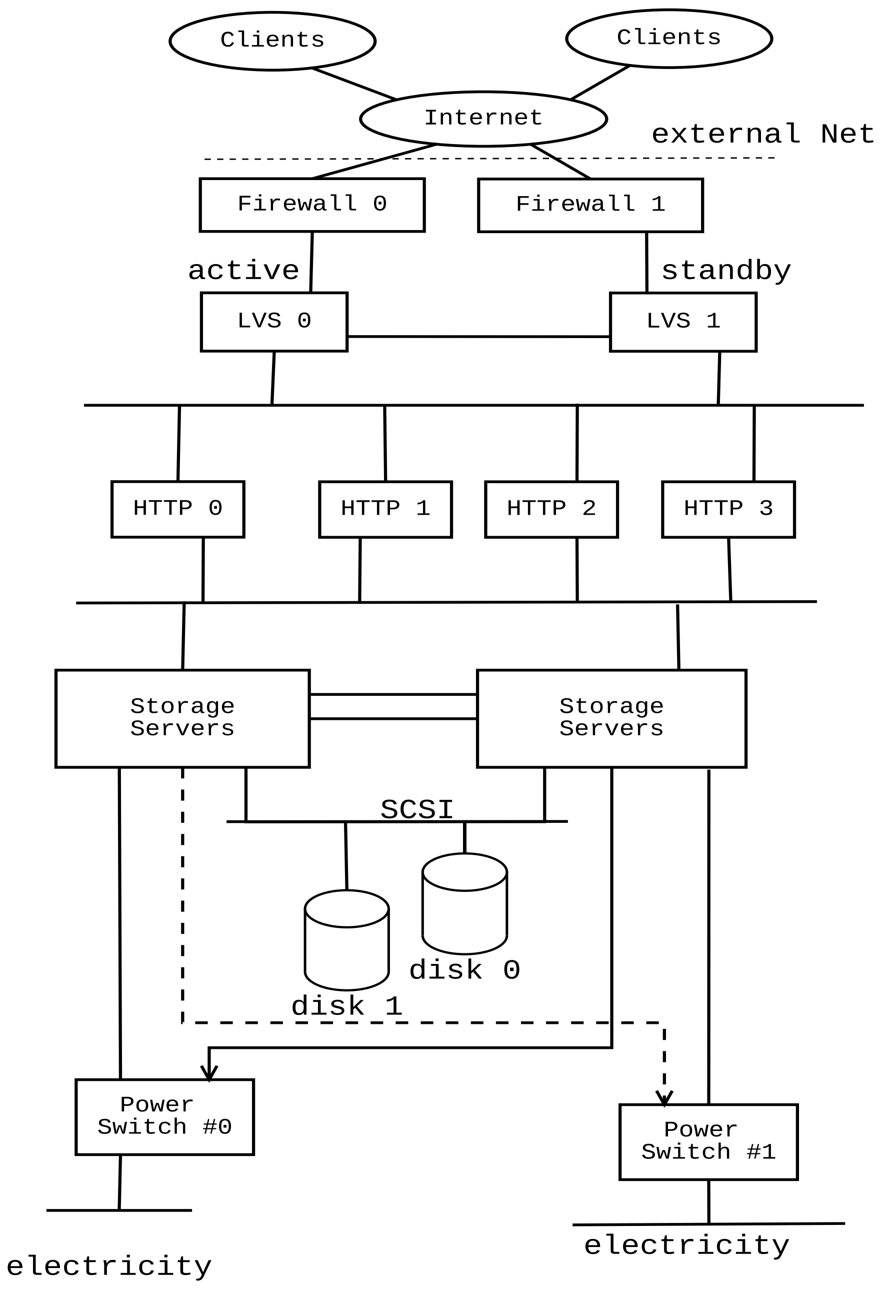


Figure 4.1. Design Goal of Kimberlite

The design requirement is that we will support more than one service. This allows both cluster members to be providing services in an active/active manner (rather than an active/passive model).

There are mainly 4 main components:

1. ***Service Manager:*** responds to changes in membership status by initiating service start/stop.

2. ***Quorum Daemon:*** Ensures that all nodes that are allowed to run services have sufficient quorum to be cluster members. Low level services maintaining shared cluster state and configuration and determining host membership. The Quorum Daemon performs all cluster inter−node coordination via a shared storage disk (i.e. not over standard communications channels).
3. ***Heartbeat Mechanism:*** The Heartbeat provides status of communication channels (serial and Ethernet channels) to the Quorum Daemon as input for making membership decisions.
4. ***System Management and Configuration:*** TBA.

Virtually all of our cluster implementation is implemented entirely outside of the kernel. All of the daemons are user space utilities written in C. There are several portions of the Service Manager infrastructure written as (ksh) shell scripts. Portions of the management/configuration infrastructure are also scripted in Tcl as well as HTML for the GUI. Based on these dependencies, we require the systems to be installed with the optional packages containing support for ksh & Tcl.

In addition, we are using the SWIG library [4]. SWIG is a software development tool that connects programs written in C, C++, and Objective−C with a variety of high−level programming languages. SWIG is primarily used with common scripting languages such as Perl, Python, and Tcl/Tk. In our case, it takes the declarations in the C header files and uses them to generate the wrappers that Tcl/Tk needs to access the underlying API. Ensure that Swig is installed on your system and is in the search path of tools used during the build process.

The only kernel level requirements are:
1. Raw I/O patch − in order to be able to access a raw device (i.e. bypassing the memory resident buffer cache), the RAW I/O kernel patch must be applied. This patch is available for both 2.2 and 2.3 variants. The raw I/O patch was not implemented by Mission Critical Linux, Inc., and is readily available.
2. Optional kernel patch to facilitate crash dump support, that we have developed at Mission Critical Linux. The protection against unsynchronized I/O operations by failed nodes consists of power cycling the partner node any time it has not shutdown cleanly. One case of unclean shutdown is when a system panics. Normally in the event of a panic you want the ability to save a crash dump for post−mortem diagnosis. Power cycling a failed node gets in the way of crash dumps as the memory resident system snapshot gets erased on power cycle. Mission Critical Linux is developing a kernel patch whereby when a node panics it will send a message over the heartbeat serial channel to the partner node. Upon receipt of this message the partner will safely conclude that the partner node is Down and that there is no need to shoot the partner.

By limiting our kernel dependencies, the Mission Critical Linux cluster implementation can be deployed on a wide range of Linux distributions.

4.1 Design Pros/Cons

Pros:

    - Gets us in the cluster space for shared storage in a timely manner.
    - Provides strong data integrity semantics.
- Cons:
    - Initial implementation restricted to 2−node clusters (for the storage server layer).
    - Requires additional hardware, although there are numerous vendors for power switches; so, it's by no means a custom piece.
    - Doesn't protect against operator error. For example an administrator on Node1 could see that a filesytem on the shared storage isn't currently mounted on this node and manually mount it. Similarly someone could dd, fsck, newfs. But one could argue that many of the same holes exist on a single system.

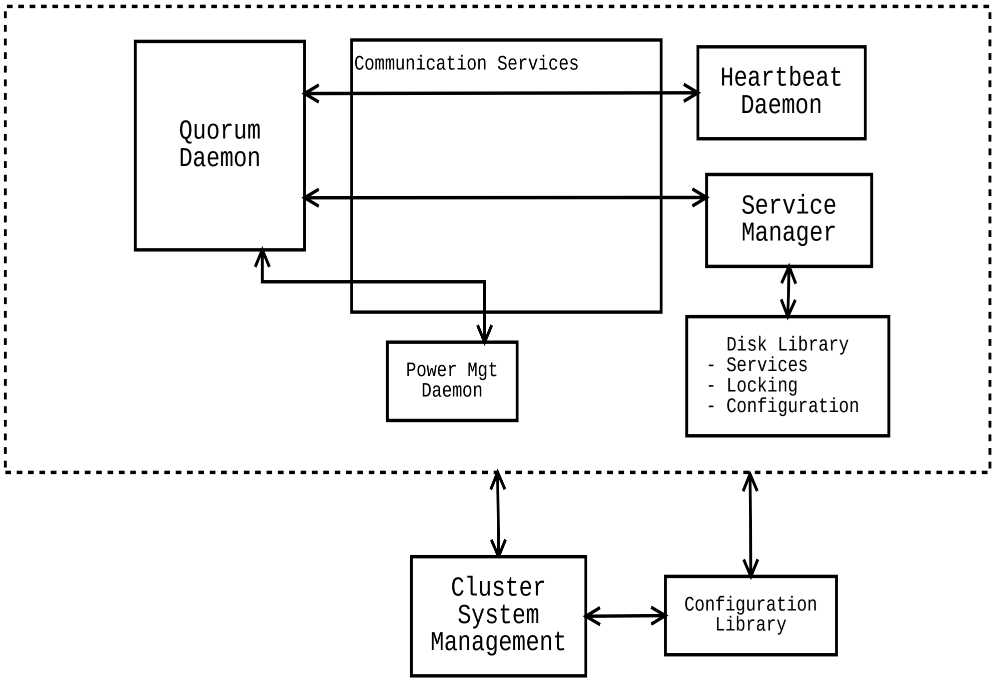


Figure 4.2. Subsystem Communication.

Figure 4.2 depicts the set of daemons and services comprising the cluster infrastructure. Interconnecting lines and arrows highlight the daemon interactions.

The communication services are used by all daemons and associated subsystems.

The Disk Services Library provides low−level locking primitives that facilitate locking across the cluster as well as among multiple processes on a single node. The Locking library is discussed extensively in **Appendix C**. The Service Manager uses the Disk Services Library for on−disk service states. The configuration library uses the Disk Services Library to store cluster configuration information ensuring a consistent view by all cluster members.

Cluster System Management is done with the help of the Configuration Library. The Configuration Library writes to/ reads from the Cluster Configuration file, a part of the Disk Services Library. We use **`syslog`** to handle all issues of message logging.

Communication between the main subsystems uses the Communication Services Subsystem, described in the next section.

Refer to **Appendix F** for a summary of alternative design approaches that were considered prior to settling on this selected model.

5 Messaging Subsystem

5.1 Overview

The messaging subsystem provides a means for cluster daemons to communicate with each other. It is only a basic communications library that operates on file descriptors. The current implementation uses TCP/IP for the underlying protocol. Over this, there is a protocol spoken by the library, and above this there is the inter−daemon communications protocol. This document will focus *only* on the message service layer.

5.2 Design Goals

The goal of this implementation is to provide a simple means for daemons to communicate, while allowing as much flexibility for the underlying protocol layer and manipulation by the users of the library. This means that if at some point in the future we decided TCP/IP was not a good protocol to use, it could easily be changed. It also means that any functionality not directly implemented in the library API could be provided by any user. For example, if the application needed to check return values from **`select()`**, it should be able to do this on its own. The library's API is detailed in **Appendix A**.

5.3 Message Service Internals

Internal to the messaging library, there are a few key data−structures which keep state information for connections. Whenever a connection is made and a file descriptor allocated from the system, the library adds the file descriptor to a list. In the list, we document the file descriptor and its associated state. Thus, a file descriptor which is allocated for listening is put in the LISTEN state. A file descriptor returned from an open call is put in the CONNECTED state, and so on. Upon calling **`msg_close()`** on a file descriptor, its entry in this table is removed.

In the interest of performance, the library also keeps a list of properly formed addresses used to contact the other processes or the other node. This is much better than forming the destination address every time a call to **`msg_open()`** is made. It also keeps us from performing unnecessary memory allocations and frees, thus helping overall performance of the subsystem. These addresses are indexed in a central data structure by the process id discussed in **Appendix A, Section 15.1**.

Finally, we have the messaging protocol itself. This is a very simple protocol designed only to pass data with as little overhead as possible. The message header consists of a version number and a size field that represents the size of the payload. This header is placed on the message before it is sent out, and is stripped off at the other end of the connection before the data is passed up to the caller.

5.4 Conclusion

The resulting interface defined provides an easy to use and very flexible method for communicating between cluster daemons, and even between nodes. It has proven to be easy to change the underlying protocol, as it was initially written to use unix domain sockets. Thus, the design goals were achieved, and the end result is a solid, easily−maintained messaging library.

6 Logging Library

6.1 Overview

Logging functions in much the same way as **`syslogd`**. There are library calls to get and set the current logging level, as well as to log a message. Logging levels are set on a per−daemon granularity, as specified in the cluster configuration file. These levels are as described in **`sys/syslog.h`**:

```
#define LOG_EMERG   0 /* system is unusable */
#define LOG_ALERT   1 /* action must be taken immediately */
#define LOG_CRIT    2 /* critical conditions */
#define LOG_ERR     3 /* error conditions */
#define LOG_WARNING 4 /* warning conditions */
#define LOG_NOTICE  5 /* normal but significant condition */
#define LOG_INFO    6 /* informational */
#define LOG_DEBUG   7 /* debug−level messages */
```

To change the level, one may run the provided **`clu_config`** utility. For example, to change the service manager logging level to 3, one may execute, assuming that the cluter configuration file called in this case `cluster.cfg` is in the `/etc/opt/cluster/` directory:

```
clu_config −f /etc/opt/cluster/cluster.cfg −o
/etc/opt/cluster/cluster.cfg −p svcmgr%logLevel 3
```

Messages are timestamped, and preceded with a string describing the daemon from which the message comes with the associated PID.

6.2 Internals

The logging facility is made up of two components, the client side library, and the logging daemon. The library filters messages based on severity, thus allowing for a per−application logging level. It then uses **`syslog`** for the logging. To define what file the cluster log goes to, one needs to edit **`/etc/syslog.conf`** and add one line as follows:

```
local4.*                              −/var/log/cluster
```

Notice that we use local4, 4 chosen at random. The user can change this number in the cluster database, should this conflict with any existing applications on the system. The user loglevels for syslog are local1−7.

6.3 API

```
int clu_set_loglevel(int severity)
int clu_get_loglevel(void)
int do_clulog(int severity, int write_to_cons, const char
*fmt, ...)
#define clulog(x,fmt,args...) do_clulog(x,0,fmt,##args)
#define clulog_and_print(x, fmt, args...) do_clulog(x, 1,
fmt, ##args)
```

The primary interface is a function called **`clulog()`**. This function takes a severity level, and a format string with arguments. **`clu_set_loglevel()`** is used to set the severity at which messages are logged (or filtered, depending on how you look at it). Its return value is the last logging level. To retrieve the current cluster logging level, use **`clu_get_loglevel()`**. The logging levels represent exactly those used by syslog. Any function wishing to use these calls should include **`sys/syslog.h`**.
`clu_log_and_print()` is provided mainly for applications that do not remove themselves from the controlling terminal and wish to have messages printed to **`stdout`**. This call takes the same arguments as **`clulog()`**.

7 Service Manager

7.1 Overview

The Service Manager is a daemon that runs on all nodes of the cluster to manage availability of cluster services. It ensures that each service is running only once in the cluster and chooses a cluster node if more than one are available to run a service. The Service Manager has been designed to run in a cluster larger than two nodes; however, it has only been tested in a two node environment.

7.2 Design

7.2.1 Shared Service Information Disk

The Service Manager uses the disk state library to manage and view service state information. This library provides a **`getServiceStatus()`** interface to get service information and a **`setServiceStatus()`** interface to modify service information. The library also provides the function call **`removeService()`**. This information is stored on a shared disk within the cluster that is viewable by all cluster nodes. There is one view of the states of all services that all nodes can see. The information that is stored for each service is: service identifier, service state, and service owner. Changes to these fields on the shared disk are done only with the service information cluster lock.

7.2.1.1 Service Identifier (ID)

The service identifier is a number to identify this service across the cluster. The number is defined in the cluster configuration and assigned to a service file when the service is created. This unique number is generated by the service add tool.

7.2.1.2 Service Owner

The service owner is the cluster node that currently owns the state of the service. Only one cluster node can operate on a service at a time, and thus, an owner is defined for this. A cluster node might only own the service for a short time to transition it to another state or it may be running the service and own it for months.

7.2.1.3 Service States

The service state defines the state that the service is currently in. This state may be a transient state or a persistent state. Please refer to **Appendix D** for a complete state diagram.

Table 7.1. Service State Description.

Name	***Description***
Service Stopped SVC_STOPPED	A service that is in the stopped state is a service that is not running on any cluster node and is a candidate to run. A service in the stopped state does not have an owner assigned to it and all of its resources are not configured on any cluster node.
Service Starting SVC_STARTING	A service in the starting state is a service that has been requested to start on a cluster node. It is in this state until either the start of the service is successful or it fails. This is a transient state and it is owned by the cluster node starting the service.
Service Running SVC_RUNNING	A service in the running state is a service that has all of its resources configured on a cluster node. This is a persistent state in which the cluster node that owns the resources owns the service.
Service Stopping SVC_STOPPING	A service in the stopping state is a service that has been requested to stop on a cluster node. It is in this state until either the stop of the service is successful or it fails. This is a transient state and it is owned by the cluster node stopping the service.
Service Disabling SVC_DISABLING	A service in the disabling state is the same as a service in the stopping state except the resulting state will be **`SVC_DISABLED`**. This is a transient state which is requested by a cluster system administrator. Disabling a service means to stop the service and not start it unless requested by the cluster system administrator.
Service Disabled SVC_DISABLED	A service in the disabled state is the same as a service in the stopped state except the service will not be started unless requested by the cluster system administrator. In normal operation the Service Manager will skip over any service in the disabled state and not change its state.
Service Error SVC_ERROR	A service in the error state is a service in which its state can not be determined. Moving the service out of the error state requires cluster system administrative intervention. It is unclear what resources associated with the service might still be configured on a cluster node.

7.2.1.4 Service Information Locking

Access to this shared service information requires locking as more than one process may try and change this information at the same time. This can be from different cluster nodes or from more than one process on the same system. The Service Manager uses the **`clu_lock()`** and **`clu_unlock()`** interfaces provided by the cluster lock subsystem to coordinate service state change requests. The Service Manager may read service information without a lock; however, it will always request a service information cluster lock before changing it. Refer to **Appendix C** for details on the locking synchronization primitives.

7.2.2 Initialization

On startup, the Service Manager initializes the service state information on the shared service disk. If the Service Manager is not running, cluster services should not be running either. Thus, when the Service Manager starts, it checks the state of all services and if any are claimed to be in an active state on itself it will stop those services and change their state to **`SVC_STOPPED`**.

After service initialization, the Service Manager contacts the Quorum Daemon via the Message Service to let the Quorum Daemon know it is ready to manage services. The first message the Service Manager expects from the Quorum Daemon is a local **`NODE_UP`** event. Until this event happens all other events are ignored by the Service Manager. The Service Manager will not respond to any node change event until it knows it is up and running locally.

7.2.3 Cluster Node State Change Events

The Service Manager waits for host event changes to manage service state changes. The Quorum Daemon is responsible for sending these host events to the Service Manager. When a host comes up (**`HOST_UP`**) or goes down (**`HOST_DOWN`**) the Quorum Daemon sends a message to the Service Manager via the Message Service specifying the node ID that went down and its new state. On receiving these events the Service Manager determines if it should start or stop services locally. Service state changing depends on the service placement policy and the current nodes that are up in the cluster.

7.2.4 Service Placement Policy

The service placement policy is the guideline for where the Service Manager should start a service. Services can have preferred nodes or start on any node available. Through the use of these placement policies, the system administrator can effectively establish a form of load balancing.

7.2.4.1 Preferred Node

Services can have a Preferred Node defined. This means that if there is more than one node in the cluster, the service will start on the preferred node. If the preferred node is not available to run cluster services the Service Manager will start the service on another cluster node.

7.2.4.2 Relocate when Preferred Node Joins the Cluster

A service may also relocate to the preferred node if it becomes available. This is referred to as "relocate when preferred node joins the cluster". The Service Manager on the cluster node running the service stops the service on receipt of the **`NODE_UP`** event from the Quorum Daemon. The preferred node coming up will fork off a process to wait for the stop of the service before starting it. If the stop never completes, this forked process will time−out.

7.2.4.3 Service Start Arbitration

Service start arbitration is done by the Service Manager when it needs to determine if it should start a service. When the Service Manager decides that a service needs to start, it runs the **`arbitrateService()`** call. This function returns a boolean indicating if it should start the service.

Services in the **`SVC_DISABLED`** and **`SVC_ERROR`** state are not started as described by these states in section 7.2.1.3.

If a service is active (in the **`SVC_STOPPING`**, **`SVC_RUNNING`**, or **`SVC_STARTING`** state) and the owner is in the **`HOST_DOWN`** state the service needs to be started. Likewise, if the service is in **`SVC_STOPPED`** state it requires starting. In these cases the Service Manager first decides if the preferred node is in **`HOST_UP`** state and, if it is, lets that node start the service. If the local cluster node is the preferred node, then the Service Manager starts the service locally.

If the node state transition given to the Service Manager is a **`HOST_UP`** event then it determines if the service relocates when a preferred node joins the cluster. The Service Manager uses the same logic to determine which server is the preferred node. The only difference here is that the starting node must wait for the service to stop before it can start the service. The Service Manager on the cluster node running the service will be in the process of stopping the service as it also gets the **`HOST_UP`** event declaring that the preferred node has booted. If the service is in **`SVC_STARTING`** state, the service will not be stopped and relocated when the preferred node joins the cluster. It may have taken a long time for the service to start and avoids the Service Manager from ping−ponging a service betweeen two nodes.

If the service does not have a preferred node, then all cluster nodes are candidates to start the service. The cluster node that acquires the service information lock first will start the service. If a system is heavily loaded this will allow for systems with lighter system loads to start the service.

7.2.5 Service Start and Stop Operations

Services are associated with start and stop configuration scripts. These are shell scripts which are forked off by the Service Manager, so more than one start or stop of different services can occur at a time.

When a start is requested by the Service Manager all of the resources for a service are configured. If some resources are already configured the start is deemed successful. All that is important is that the local node owns the resource. This also helps in the failure cases below.

When a stop is requested by the Service Manager all of the resources for a service are deconfigured.

7.2.5.1 Start and Stop Failures

When a service starts or stops, these operations can fail. In these cases, the following general guidelines are followed.

If a service is stopping, but fails, the service is then started. If the subsequent service start fails, the cluster node is rebooted. This allows another cluster node to make the service available. This is necessary to avoid having a service in an indeterminate state, possibly resulting in service unavailability.

If a service is starting and it fails the service is then stopped. If the subsequent service stop fails, the service is put into the **`SVC_ERROR`** state. The service is put into the **`SVC_ERROR`** state because it is not clear from the Service Managers perspective how much of the service start and stop and been completed. There may be resources on the system that can not be configured on more than one system at a time. A service in the **`SVC_ERROR`** state requires cluster system administration intervention. The cluster system administrator needs to remove all resources associated with the service in the cluster node that it failed on. A node reboot would remove all resources, but is not necessary. When this is done, the service may be manually moved out of the **`SVC_ERROR`** state into the **`SVC_STARTING`** state and started on another cluster node.

7.3 Service Resources and Configuration File

A service description consists of a definition of resources that are associated with that service. The services section is located in the cluster configuration file, **`cluster.cfg`**. All entries after this label until end of file or another label are related to services. Each service has a section in the service definition section that defines all of the configuration information and resources assigned to it. The service description beings with a the string "**`start service#`**" and ends with the string "**`end service#`**". The '**#**' is the number of the service entry and is referred to as the "service ID". Service numbers should be assigned sequentially, beginning with 0.

Note: the service ID for each service must be unique in the cluster configuration file (two services cannot contain the same service ID).

It is important to note that all service attribute names must be correctly spelled and are case sensitive. If not correctly entered into the service description the Service Manager will not be able to find the settings or resources defined for the service. Also make sure that the name used in start and stop sections are the same (i.e. "start service0", "end service0").

Following is a definition of all service fields and valid values that can exist within the "**`start service#`**" and "**`end service#`**" strings of a service.

Table 7.2. Service Resources Description.

Name	***Description***
`name = "string"`	Each service has a name that is a character string. This name is a human readable handle for the service. The Service Manager and other components in the cluster use the service identifier to refer to a service. The service name is the only required attribute of a service that the user is required to define.
`disabled = yes or no`	The service disabled setting is the manner in which a cluster system administrator can keep a service in the stopped state. In general the Service Manager wants to have services running if they are not. This option allows the administrator to keep a service stopped for maintenance or any other reason.
`preferredNode = node1 or node2`	The preferred node setting defines a cluster node that the service prefers to run on, if the node is up. If the node is not up the service will run on any other node with equal priority.
`relocateOnPreferredNodeBoot = yes or no`	If a service has a preferred node, then it can relocate to that node when the node becomes a cluster member (is up). This allows a service to always relocate to the preferred node whenever it is a member of the cluster.

Name	***Description***

`userScript ="/pathname"`	Customization of a service can be done using a user−defined script. This script is run at service start time and service stop time. It contains specific instructions and information about the service to start applications or configure the service. The script is similar to the scripts found in /etc/rc.d./init.d. The value of this entry is a fully qualified (starts with "/") pathname of the script. The script is called with $1=start or stop and $2=service name.
	
	On service start the user script is run after all other resources have been started. On service stop the user script is run first before all other resources have been removed.
	Following is a sample user script:
	`#!/bin/bash`
	`action=$1`
	`svcName=$2`
	`case $action in`
	`’start’)`
	`  echo "Running user start script for service $svcName"`
	`  ;;`
	`’stop’)`
	`   echo "Running user stop script for service $svcName"`
	`  ;;`
	`esac`

`start network#`
` ipAddress = w.x.y.z`
`   netmask = 255.255.255.0`
`    broadcast = w.x.y.255`
`end network#`
	A network section exists in a service description entry to define an IP address for that service. If more than one IP addresses is to be configured for a service there will be multiple entries starting with network0 and incrementally increasing. Within each section there is an attribute called "ipAddress" which takes a dotted decimal IP address as a value. This IP address will then be aliased to a physical network interface on the cluster member upon which the service is currently running. This is a "floating IP" address which is distinct from a fixed IP address associated with a system’s network interface.
	When a user assigns an alias to a network interface, it does not inherit the IP settings of the interface. If none are specified, the service manager will find the appropriate netmask and broadcast address.
	However, the user may, if he/she wishes so, override these values by specifying the netmask and broadcast address of a service in this resource.

Name	***Description***

`start device#`
 	
`name = "/device_pathname"`
 	
`start mount`
`  name = "/pathname"`
`  options = mount options as in /etc/fstab, e.g. rw,nosuid`
`  forceUnmount = yes or no`
`end mount`
 	
`owner = /etc/password name`
 	
`group = /etc/group name`
 	
`mode = numerical mode, e.g. 0755`
 	
`end device#`
	A device section exists in a service description entry to define devices and its associated attributes and configuration for a service. If more than one device is to be configured for a service then there will be multiple entries starting with device0 and incrementally increasing. Each device section must contain a device name. This is a fully qualified pathname of the device (e.g. `/dev/sda1`).
	If this device is file system based then a mount point needs to be defined. A mount subsection defines all of the mount attributes for a device.
	A mount name is required for a file system based device and is a fully qualified pathname of the mount point (e.g. `/var/cluster/mnt/myservice/usr/foo`). Any mount options that are desired are defined in the options attribute. If the mount point should be unmounted with "force" then the attribute forceUnmount can be defined as yes. The force unmount option kills all processes on a mount if an umount request fails.
	
	Each device can have permissions associated with it. These are owner, group, and mode. These are normal file permissions that one would use with `chown`, `chgrp`, and `chmod`. If the device is a file system based device then these permissions will be applied to the mount point (as defined in the mount subsection of the service). If the service is a raw device then the permissions are applied to the device (e.g. `/dev/sda1`).
	
	Note: if raw devices are used then the `/etc/rc.d/init.d/rawio` file must be modified to map the device names to the raw device names. Details for this can be found in the "Kimberlite Cluster Installation and Administration [5]" manual.

The following is an example service section in the cluster configuration file.

```
[services]
     start service0
          name = kimberlite
          disabled = no
          preferredNode = cluster_node1
          relocateOnPreferredNodeBoot = yes
          userScript=/var/cluster/scripts/kimberlite

          start network0
               ipAddress = 192.168.1.160
          end network0

          start network1
               ipAddress = 10.0.0.160
          stop network1

          start device0
               name = /dev/sda1
               start mount
                    name = /usr/projects/kimberlite
                    options = rw,nosuid
                    forceUnmount = yes
               end mount
               owner = kimberlite
               group = cluster
               mode = 755
          end device0

          start device1
               name = /dev/sdb3
          end device1
     end service0

     start service1
          name = testService
     end service1
```

8 Heartbeat Mechanism

8.1 Introduction

The Heartbeat daemon monitors the status of cluster members from a high−level communication perspective. Heartbeat works by having hosts communicate over a set of redundant communication paths. The paths considered in our implementation include Ethernet and serial ports. The Quorum Daemon performs additional membership monitoring as well using a disk partition on a shared SCSI bus. Heartbeat actively monitors the communication paths such that it can provide input to the Quorum Daemon in quorum determination.

8.2 Design Considerations

In considering Linux−HA [1] "Heartbeat" code, developed by the Linux−HA project, it became apparent that some design decisions that were made in Linux−HA could not be accommodated in our design. In particular, we wanted to de−couple service management from host membership services. Linux−HA makes service decisions solely on the communication channel state, which can cause data integrity issues under certain failure scenarios. Some attributes are broadcast over subnets, making that incompatible with a possible scenario involving multiple clusters on one subnet. By de−coupling service management from communication channel monitoring, we were able to properly implement quorum and I/O barriers in the Quorum Daemon layer and use it to drive service placement.

Heartbeat has knowledge of the cluster configuration and communicates with other nodes over different channels. It communicates over the Ethernet lines using UDP, and over serial lines in raw mode. A node heartbeats to other nodes every 2 seconds (configurable parameter with default setting) and gives up after 5 tries (configurable parameter with default setting), thereby allowing the other node several opportunities to respond to it, before declaring it Off−line over a particular channel.

8.3 Implementation Details

The "Heartbeat" daemon runs on every cluster node. A daemon running on a certain node monitors the other node(s) by pinging them over multiple physical access lines, mainly the serial line, the private Ethernet link and the public network.

In order to build knowledge of the cluster configuration, Heartbeat makes use of functionality provided by **`get_clu_cfg()`**. This function reads a configuration file (the default or a user−provided file). It sets up the name of the cluster and its members, determines the identity of the node reading the configuration file, scans in channel configurations (both network and serial interfaces). The function **`get_clu_cfg()`** also has some error−checking functionality to make sure that the configuration in the file is consistent with actual hardware.

Other issues and details about the implementation are discussed at length in the Quorum Daemon section.

9 Shared State Disk Partition

There is a shared disk partition that can be concurrently accessed by both cluster nodes (consequently, it is accessed as a raw device, not a mounted filesystem in order to, for example, eliminate any cache−ins by the nodes). There is a section of this partition used to represent cluster global state information for each node. Each node writes the following information:

- **Node status**: a node marks its own UP/DOWN state. This is necessary to facilitate clean shutdown.
- **Activity timestamp**: The activity timestamp is updated by each node on a periodic basis. The frequency is configurable with a default period of every 5 seconds. This represents the system's timestamp. If a node is active, it results in this timestamp changing (increasing) over time. (Recall that each node has its own activity timestamp.)
- **Service descriptions**: Each service has a corresponding on−disk representation. This provides a means for cluster members to have "shared state" describing the status of each service. This state is used to control service startup placement.
- **Node locking status**: Each cluster node has a disk resident lock data structure. This is used to serialize access across nodes to the service description entries. This synchronization ensures that a service's state is only modified by a single cluster member.

- **Cluster Configuration "database"**: in order to ensure that all cluster nodes have a common view of cluster configuration information (i.e. Subsystem parameters, service descriptions), we represent this information in the shared disk. This avoids complex synchronization issues inherent with alternative approaches such as maintaining consistency among copies of configuration files residing on local filesystems of each cluster member. Refer to **Appendix E** for details.

There is also an area of the partition in which each node represents a list of the services that it is currently serving.

The scheme has separate portions which are write−only to each node. For example, Node 0 may write its info to block X, while Node 1 would read block X. This can help mitigate read/modify/write problems if both nodes tried to concurrently read/write the same block.

We provide a partition initialization utility that is used at install time. The utility is invoked by running '**`diskutil −I`**'. This initializes the partition to indicate that nobody is serving anything. In this manner the partition contents aren't random garbage which could cause the algorithms to make incorrect assumptions.

9.1 Where to Place the Shared State Disk Partition?

- One option is to have a single partition which represents all services.
  - Pros: Simplifies design to have everything in one place
  - Cons: Single point of failure. Therefore we require a redundant RAID controller. We also maintain a shadow copy (detailed later).
- Another option is to have a separate partition per service
  - Pros: Not a single point of failure in that loss of a partition will only impact a single service. However, this still represents a single point of failure per individual service.
  - Pros: Suppose you had lots of storage that necessitated usage of more than one SCSI bus. Consider the case where the shared state partition is on Bus 0 (accessed by SCSI Adapter 0) , while the data resides on disks on both Bus 0 and Bus 1 (accessed by SCSI Adapter 1). Now suppose that SCSI Adapter 0 fails (or has its cable unplugged). In this case the host would no longer be able to serve the service, and ideally it should intelligently fail it over to the other cluster server who may have connectivity to the storage. In order to make such a scheme work, we would need to have a separate shared state partition on each shared SCSI bus (on a bus granularity; not a service granularity).
  - Cons: Increases management complexity
  - Cons: Increases processing overhead and code complexity. For example, you have to update Heartbeat intervals and service state in multiple locations. To ascertain the state of a potentially failed partner node you have to collect potentially inconsistent state data.

Our design model consists of a single logical shared state partition, which is physically mirrored, rather than separate partitions per service. Having a single shared state partition makes for a simple algorithm for the Quorum Daemon to determine if the other host is down. We support more than one shared storage SCSI bus for capacity purposes. However, there is a single shared state partition. To eliminate this from being a single point of failure, it can be configured as a redundant RAID volume, as detailed in section 9.1.

Beyond the number of partitions there is also the question of whether the shared state disk partition should be on an entirely separate physical disk.

- Pros: Avoids controller optimizations governing head seeks from starving out access to the activity timer region. Example: Oracle running on a disk heavily using a second partition of the disk could queue enough I/O’s to keep the heads out of the infrequently used shared state partition.
- Cons: Costs you a whole disk.

Fortunately, the design affords the flexibility of either having the shared state partition being on the same or different disks from the services. This will allow us to support both configurations and make recommendations based on our experiences.

NOTE: Having multiple nodes concurrently accessing a disk on a shared SCSI bus requires the kernel be configured with "Raw I/O" support. Without Raw I/O support, reads and writes would be cached in the memory resident buffer cache. This would result in a node not seeing the writes posted by another node. Such behavior precludes correct concurrent access. Fortunately, Linux does provide Raw I/O support as patches in both 2.2− and 2.3−based kernels.

9.2 Shared Disk Protections

A foundation component of the cluster design is the representation of cluster state information on shared storage disks. This state information includes node states (up/down), service descriptions, lock synchronization primitives, etc. Since the shared cluster state as represented on disk is vital to correct operation, the implementation goes to great lengths to protect the integrity and availability of this shared state information. This section highlights the provisions for not making this shared disk a single point of failure.

9.2.1 Disk Access Library

There are a set of APIs provided which are used to access the various "data structures" represented on the disk. Examples include APIs to access:

- Member state information
- Service descriptions
- Configuration database

These disk access APIs abstract the callers from low level details such as:

- Knowing the physical layout of the partition to determine where the requested data resides.
- Performs validation checks, including checksums.
- Repairs corrupted data using the shadow partition copy.

• Meeting the stringent requirements for accessing a device using rawio. (This includes transaction size and alignment.)

Note: The disk access APIs are not currently safe for usage in multi−threaded applications or forked instances inheriting common file descriptors. In order to safely call the disk access APIs in these situations, it is necessary to bracket the calls by calling the provided lock synchronization primitives (documented separately later).

9.2.2 Software Protections

Each piece of cluster state information is represented on the disk in separate 512−byte blocks. In addition to the state information itself, we also store some meta−data information in each "structure". This meta−data information includes:

1. A magic number − particular to each type of state information.
2. A version number − allowing for inter−operability with future releases.
3. A checksum − used to detect corruption of the data structure.

The following describes how the redundant shared state partitions are used to achieve maximal fault tolerance. Each write of cluster shared state information to the shared state partitions is issued to both the "primary" and "shadow" partition and the checksum is set appropriately. Failure of either of these 2 writes will cause a node to remove itself from the cluster. The read path for cluster shared state information will randomly issue the read request to either the primary or shadow copy. Upon success the data is returned. In the event of failure (detected by checksum and/or magic number discrepancy) a read will be issued to the other shared state partition. If this succeeds, the failed copy will be written with the repaired data.

As a result of the read/repair algorithm, the shared state partitions become self−correcting as they are read. There may be portions of shared state information which are infrequently read. One such example is the partition header which is typically only read in once as a cluster member is started. Such infrequently accessed data would not automatically be repaired if it were not read in periodically. In order to close this window of vulnerability where in the face of error you are down to a single partition, the entire shared state partition is periodically read in. This scan of the share state partition is performed by reading in only one of the categories of shared state information per interval. There are 5 such categories. The current scan rate is every 30 seconds, resulting in a full sweep of the partition every 2.5 minutes.

The main goal of the shadowing scheme for cluster state information is to protect against user/operator error. A user performing a ‘mkfs‘ or ‘dd‘ accidentally to one of the two quorum partitions is one such error. In this case, since two copies existed, reads to the corrupted partition will fail, but reads to the other partition will be retrieved properly. When the cluster updates its state information on both partitions of the quorum disk, it will write correct information to the corrupt partition.

If the partitions are placed on separate disks with no RAID redundancy, on different SCSI buses, one could fall in the pathological double failure whereby each system could access only one of the partitions. This would cause each system to assume the other is down. In addition, separate disks with no RAID redundancy will lead to the whole system going down (since we always want to write to both partitions).

9.2.3 Hardware Protections

This is the case where a SCSI disk fails, a SCSI adapter fails and the system is not capable of accessing the disk. Through the usage of a RAID box with redundant controllers and appropriate RAID redundancy (such as mirroring) the individual disk that the shared state information resides on is not a single point of failure.

In a multi−ported RAID controller configuration, should a node lose connectivity to the shared state disks (i.e. SCSI cable pull or PCI adapter failure), that node will remove itself from the cluster (and/or) be power cycled by the surviving node. In this case the partner node will continue to provide cluster services.

9.3 Quorum Daemon

The primary roles of the Quorum Daemon are:
1. to ensure that only nodes with quorum are allowed to be active cluster members.
2. to provide I/O Barriers necessary to ensure data integrity.
3. To represent shared cluster state by representing node status as well as describing services state.

The Quorum Daemon will be implemented as a user level utility. The current design requires "Raw I/O" access, the implementation of which is discussed in the previous section in terms of kernel level requirements.

The Quorum Daemon will be started up by the cluster initialization scripts (detailed in a later section). The Quorum Daemon will perform a set of initialization tasks such as:

9.3.1 StartQuorumDaemon() {

- If the Quorum Daemon is unable to access the shared state partition (i.e. SCSI cable pull, adapter failure), it will return an error status. The result is that this node will then fail to become a cluster member.
- Any other mop−ups which may need to be done to remove any state from previous boots, for example, to initialize the node’s own state and lock structure.
- Commence timestamp pinging by calling **`QuorumdBody()`**.

9.3.2 QuorumdBody() {

This routine contains a loop that runs until the Quorum Daemon is stopped as part of a clean cluster shutdown. The following steps will be performed periodically.
- This daemon will update its activity timestamp even if it is not currently serving any services. If the update to the timestamp fails (after a few retries) the Quorum Daemon will reboot the system as current cluster semantics cannot be assured.
- Both SM and PowerD update QuorumD every 30 seconds that they are operational. If

QuorumD does not hear from SM in two minutes, it will reboot. If it doesn't hear from Powerd in 2 minutes, it will only log errors.
- Call CheckPartnerActive(); detailed later.
- See if there are any messages sent by other cluster daemons or management/ monitoring utilities.

}

9.3.3 StopQuorumd() {

- Terminate child quorumD sub−process
- Mark disk state as down so that partner does not shoot that member node that quorumd was running on.
- Notify the Service Manager that we are down. The SM could already be stopped, which is OK. There is no need to do this if the daemon has initiated this, as it would be in response to a Service Manager request.
- Stop power daemon.
- Perform final cleanup

}

9.3.4 checkPartnerActive() {

// This is a low level disk based "Heartbeat" mechanism used to determine if a
// partner node is still alive. Since it is disk based it is inherently of longer duration
// than the comparable network−based Heartbeat schemes.

- Read in partner's status block from shared partition
  - Repeated I/O errors will cause a node to shut itself down
- If the partner's state is stopped, no additional checks are performed.
- Otherwise, the state isn't down. See if the timestamp has changed.
  - If the timestamp hasn't changed, bump a counter. If after a few iterations, it still isn't incrementing, consider the partner failed. In this case, contact Heartbeat to see if it considers the partner to be alive*. If so, give the partner longer before shooting it, call shootPartner() − described later.
  - If the timestamp is changing, mark the partner's state as up. Note: it is therefore not a strict requirement that all cluster member's systems clocks be closely synchronized.

}

* <u>Note:</u> allowing the partner extra time beyond the normal polling frequency to update its timestamp before shooting it is done to avoid prematurely declaring a node down during a spike in I/O or system activity.

9.3.5 shootPartner() {

```
    Check power switch status
    If (OK) {
            Power cycle partner
            Set ON−disk state at partner to DOWN
            revoke any synchronization locks held by the failed node.
```

```
        Notify Service Manager that partner is DOWN
        return (SUCCESS)
    Else {
        // power switch failed
        if (power switch returned error) {
            // do nothing because we can’t safely shoot partner
            // and takeover services
            return (FAIL)
        }
        if (status command timed out) {
            if ((we have "recently" −configurable− successfully
            retrieved status from power switch ) &&
            (heartbeat indicates power down)) {
                // assume partner and its power switch have lost power
                set on−disk state of partner to DOWN
                Notify SM
                return (SUCCESS)
            }
        }
        return (FAIL)
    }
}
```

10 Power Switch Options

10.1 Background

Investigating the options for remote power switches. These efforts included:

- Hardware switch selection. Factors include: price, reliability, amperage/voltage requirements, rack mounting. Next section discusses the RPS−10 Switch that we use.
- We considered UPS/Switch combinations but considered them too expensive, with unneeded features, and usually more switchable outlets than needed.
- Failure scenarios for the switch itself are considered separately in a later section.
- Development of an API that the other cluster software components can call into to do: status reporting (is there a remote switch present and reporting operational status), on, off, toggle, etc., as explained in **Appendix B**.
- There is also a "lower level" into which various remote switch types can be used with the appropriate "driver"−like code. This "driver"−like code provides the device specific abstraction from the upper level API; thereby allowing us to accommodate a range of remote switch types.

10.2 Purpose

I/O Barrier − used to ensure that a failed node does not write data to shared disk. Hung node is the problematic case.

10.2.1 Observations:

Necessary for service takeover from failed nodes

- can safely takeover services which have been cleanly stopped
- upon rejoining, a node can safely takeover services for which it was the previous server

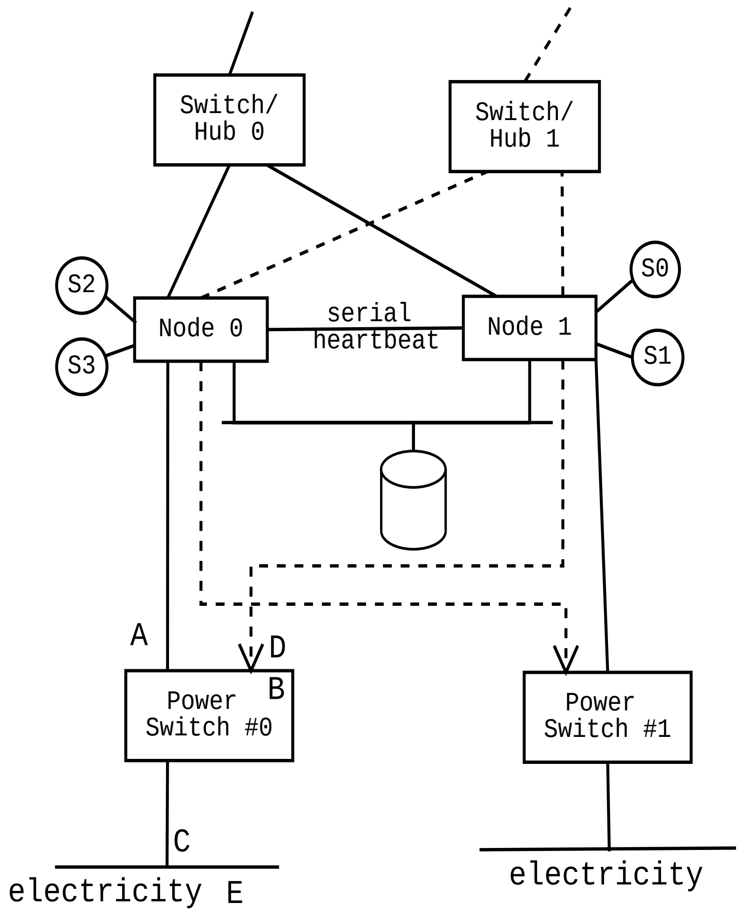


Figure 10.1. Hardware Setup and Failure Points.

10.3 Assumptions

1. It is possible to query the switch to determine its status in detail.
2. There will only be one switch managed by this interface. That switch will control power to one machine.
3. Only one process will ever attempt to communicate with the switch at any given time, i.e. there are no contention issues related to which process controls the switch.
4. Nothing is assumed in the interface about the nature of the communication channel to the switch. The implementation hides the details of the communication channel.
5. Power switch has a single power cycle command for atomicity
6. Power switch has status command + return status from power cycle command.

10.4 Example Implementation: The RPS−10 Switch Behavior

The RPS−10 can be controlled over a serial line. It responds with a string "RPS−10 ready\n", when the corresponding serial port file, e.g. /dev/ttyS0 is first opened. There is a delay before the initial response string is displayed. It is on the order of 10 sec. Normally the switch doesn't respond when the serial port is opened, and must be queried to see if it is alive.

When no power is supplied to the switch, it does not respond to commands over the serial port. There is no other indication. If there is no response to a command or query then this can be taken to be a possible indication that power is not supplied to the switch.

When power is removed from the switch, the condition of the switch after power is re−supplied is dependent upon the position of a dip switch. Depending on the position of the switch, the device will power on in the condition that it was in when powered off, or will power on closed. For our purposes the dip switch will be positioned so that the switch is closed, i.e. supplying power to its load, when power is initially supplied to the switch.

When the serial line is disconnected, the switch remains in the position that it was in at the time of the disconnect. The switch remains in that same position after the serial line is reconnected.

The power toggle delay can be configured to be 5 sec. or 10 sec.

After a command is sent to the switch, the switch will respond with the current switch position, i.e. the position after the command has completed. Then the switch will send "Complete\n" through the serial port.

Since the only indication of failure is that the switch does not respond in a timely way, time−outs must be supported in software.

10.5 Power Switch Related Failure Scenarios

The letters for the scenarios correspond to the failure points in the figure above.

10.5.1 Scenario A

Failure: Node gets unplugged from power switch, or node gets powered off.
Response: Node 1 successfully shoots down Node 0 and takes over services.

10.5.2 Scenario B

Power switch fails. We take three cases under consideration.

10.5.2.1 Scenario B1

Failure: Switch fails such that node 0 gets powered OFF from previously ON, and power switch doesn't respond to serial commands.
Response: Node 1 can't talk to power switch. If the power switch has been recently contacted, Node 1 assumes that Node 0 is without power (not that the serial cable has been pulled) and takes over services. Otherwise, Node 1 cannot safely ascertain the state of Node 0 so it cannot safely take over Node 0's services.

10.5.2.2 Scenario B2

Failure: Switch fails such that node 0 remains powered ON, but power switch does not respond to commands (status or cycle) from serial port.
Response: Node 1 is unable to conclude the state of Node 0 and it will not take over services. No action taken.

10.5.2.3 Scenario B3

Failure: The switch lies! It reports successful power cycle or status when in fact it really didn’t power cycle.
Response: In this case, Node 1 will take over services from Node 0. Should Node 0 come out of a hung state and resume pinging the disk (updating its timestamp), Node 1 will detect that Node 0 is using an old incarnation number and Node 1 will then reboot.

10.5.3 Scenario C

Failure: Power switch gets unplugged from the electric outlet: the node powers down and doesn’t respond to serial commands.
Response: Same as scenario B1

10.5.4 Scenario D

Failure: Serial port gets unplugged, or the serial port controller fails
Response: Verify recent power switch connectivity to know that we haven’t lost the power switch; if so, safely assume partner is dead. If there has been no recent connectivity to the power switch and the partner node appears to be down, services cannot be safely failed over.

10.5.5 Scenario E

Failure: Electricity goes out: Since we assume that the power switches are plugged into separate circuits (power grids), then if a circuit breaker blows, this leaves the other node/switch up.
Response: Same as scenario B1.

10.5.6 Service Availability Disruptions

Note: These service disruptions are the result of double−fault scenarios.
Scenario D: serial cable to power switch fails and a node hangs
Scenario B2: power switch fails to communicate over serial line while stuck in ON position
Both of these result in Node 1 continuing to run the services it had, but it will not take over services from Node 0.
The risk of Scenario D can be reduced by printing error logs anytime we can’t talk to the switch; this clearly does not completely eliminate the risk. Additionally, the system monitoring user−interface will display an indication of the power switch status. By checking to see that we have recently successfully probed the power switch, we increase the probability of knowing that we are not in failure scenario D.

Note: To bound the number of I/O’s that get emitted when a node hangs, check incarnation number of node to see when it un−hangs. In this case, shut yourself down.

10.6 Implementation

1. Node will join cluster even if (powerSwitchStatus() != SUCCESS)

2. Power Daemon periodically monitors ability to communicate with power switch. It records a timestamp of the last successful contact.

3. Quorum Daemon {

```
        if ( other node has stopped updating timestamp ) {
                notifyServiceManager = TRUE;
                // contact Heartbeat Daemon to see if it still sees the other
                // communication channels as alive.  If so, wait before shooting partner.
                if ( shootPartner() != SUCCESS ) {
                        notifyServiceManager = FALSE;
                }

                if ( notifyServiceManager == TRUE ) {
                        // Initiate Service takeover
                        save off incarnation number of failed node
                        set partner’s disk state to DOWN
                        notify Service Manager of partner down
                }
        } /* if ( other node has stopped updating timestamp ) */
}
```

4. Service Manager (Partner Down Notification) {

```
        for each ( service ) {
                if ( service.server == partner ) {
                        // simplified way
                        service takeover;
                }
        }
}
```

11 Dumps and Panics

There is a very unpleasant side−effect of shooting a failed node via power cycle. Specifically, this would likely preclude the ability to take a dump. Consider the following scenario:

- A node panics, thereby inducing a network and storage partition.
- The surviving node shoots the panicking node.
- The panic node gets power cycled which obliterates the kernel state from memory;

thereby precluding its preserving a dump upon reboot.

So how can we improve upon this? The following are some ideas Dave Winchell & Tim Burke cooked up.

- One could use hardware much more exotic than a power switch to incapacitate the other node. In the interest of sticking with commodity components and keeping costs down, there weren't any obvious options.
- An alternative scheme was to use the serial port as a means of communicating to the other node that "I'm in the process of a panic", along with a periodic indication "that I'm still in panic", culminated with a "I'm done panicking and calling reboot now". Usage of the serial port is based on its ability to operate in polled mode (not requiring interrupts); making it callable from the panic code. (Actually, since ‘savecore‘ isn't performed on the panic side, the time it takes to get through the panic code shouldn't be that long. We may be able to achieve the same results by simply sending over a single message like "I just did a panic and am about to reboot − so leave me alone" message. Here, you'd just need the time it takes to declare a node dead be long enough to include the time it takes on average to get through the panic code.)
- The surviving node would notice that the other node is in panic and not power cycle it. There would be some (configurable) time limit to how long it will give the panic sequence to conclude. If the surviving node hears from the other node that it has initiated the reboot, then it can safely assume that its safe to failover the services without power cycling.
- We considered having the panic code modify the shared state partition to reflect the fact that the node was in a panic situation. This approach was dismissed because it requires too many system services which could not be depended upon in a panic context.
- Pros: Allows dumps to work
- Cons: Could increase the service failover time. Added implementation work.

12 Cluster Startup/Shutdown Scripts

12.1 Cluster Start

In order to orchestrate the proper startup sequence, there will be a cluster Startup/Shutdown script in SYSTEM V Init sequence. This script "**`cluster start`**" will then automatically be called appropriately. The order of startup is:

1. Power Daemon
2. Heartbeat
3. Quorum Daemon (has dependency on Power daemon)
4. Service Manager (dependency on Quorum daemon)

Behavior in the case of failure to start any of the daemons is dependent on what daemon actually dies. The following table briefly explains the result of any failure.

Table 12.1. Behavior of Cluster in event of a Failure when starting Daemon(s).

Daemon	***Behavior***
Power Daemon	Services that have been cleanly stopped can be run on this node. However in the event of failure of the partner (unclean stop), no services will be failed over as data integrity cannot be ensured.
Heartbeat	Services will continue to start and be failed over. Failover times will decrease.
Quorum Daemon	The node will never declare itself a cluster member. Consequently no services will start.
Service Manager	No services will be started

12.2 Cluster Stop

Cluster shutdown proceeds in the opposite order as startup. It begins with the Service Manager attempting to cleanly stop all services.

Stopping the cluster daemons is not a simple matter of sending them a kill signal. This is due to the fact that the daemons must be cleanly shutdown in order to stop all running services and mark the node’s state as down. Failure to shutdown cleanly will result in the other cluster member to detect the instability and shoot the other node.

The stop sequence is as follows:
1. Call a utility (stopcluster) which sends a message (via the cluster messaging service) to the service manager. Upon receipt of this message, SM will cleanly stop all services.
2. After SM has cleanly stopped all services it sends a message to the Quorum Daemon telling it to cleanly shutdown. To do this, Quorum Daemon marks its state on the shared disk partition as being down.
3. Quorum Daemon sends a message to the power daemon telling it to shutdown.
4. Heartbeat gets killed off.

12.2.1 Additional Information and Particularities

When one runs the "**`cluster stop`**" command to initiate cluster shutdown, it initiates the chain of messages by calling a utility **`stopcluster`**; and also killing the Heartbeat daemon. After going through those steps, the **`cluster`** script completes. Meanwhile, off in the background, the SM is busy stopping services (which could take a while depending on the type and number of services). Consequently, one may run "**`cluster stop`**", and before that process exits , if he/she runs "**`cluster status`**", then in this instance, it is likely that some daemons remain for the duration it takes for the clean shutdown sequence to complete.

If the SM is unable to cleanly stop all services, then it would not be safe for any running services to be taken over by the other cluster member. For this reason, if SM is unable to stop all of its services it does not send the terminate message to the Quorum Daemon.

12.3 Cluster Restart

The current cluster can be ’restarted’ without an intervening reboot. We intend to revisit whether there are any possible negative implications for service availability.
If one wants to accomplish the equivalent of a cluster restart, the following steps should be taken:

1. Stop the cluster services and daemons (via ‘`cluster stop`‘)
2. Verify that all the cluster daemons have stopped (via ‘`cluster status`‘). Do not attempt to start the cluster daemons if the stop has failed to terminate all the daemons. (Should you find yourself in this situation, a system reboot is warranted).
3. Start cluster daemons and services (via ‘`cluster start`‘).

13 Failure Scenarios

The previous sections may not be completely clear on the desired behavior in the case of a failure. So to help illustrate the point, the following use case scenarios attempt to describe the high level sequence of events in response to various node failures.

13.1 Scenario: Two Nodes Up − Pull SCSI Cable from One Node

1. Quorum Daemon can’t update its own timestamp and attemps to shutdown cleanly by executing a "shutdown −r now".
2. meanwhile, other node’s Quorum Daemon times out the other based on disk inactivity:
   - sees that other node’s state as in cluster
   - Quorum Daemon on survivor calls Heartbeat when it sees no disk activity prior to shooting it
   - QD shoots other node. N.B.: if we can ping over network, wait longer before shooting to allow clean shutdown
   - Heartbeat detects NET_DOWN
   - Quorum Daemon notifies SM that partner is down
   - SM takes over services

Shutting down a node will probably be interrupted by the other node shooting it. This is a tradeoff relating to failover times. If you grant longer failover times, there will be a clean shutdown. In practice though, given a SCSI outage the cluster stop sequence is likely to stall while attempting to stop services (on the now inaccessible shared disk). It is unlikely that the service failover constraints would ever be long enough to perform a complete clean shutdown. The end result is that in the face of SCSI inaccessibility a node will get as much shutdown initiated prior to being shot with the understanding that a node doesn’t get far in the shutdown for short failover times.

13.2 Scenario: Two Servers Up − Pull Cluster Shared State Disk

It is possible −and very likely− that the shared state partition is not the only partition on the disk. With this setup, pulling the disk may lead to loss of access to data or to services altogether, running on the other partitions.

To eliminate this scenario as a SPOF, the quorum partitions should be mirrored at the RAID controller level.
The following behavior will occur upon complete inaccessibility of the quorum partitions.

1. both nodes fail to update their timestamp; both exec "shutdown −r now"
2. nodes reboot
   - cluster startup fails due to inability to access disk. Nodes fully boot but don't start any HA services

Note: (v.1.0) Cluster activity has stopped and will require manual intervention to start. It isn't polling while waiting for SCSI cable to be plugged in and then automatically startup cluster services. At this point, both nodes will be up, but neither will be running the cluster daemons and providing cluster services. It is up to the system administrator to notice the outage and take the necessary corrective action (such as plugging the disk back in and rebooting).

13.3 Scenario: Split−Mirror Scenario: Mirrored Shared State Partition

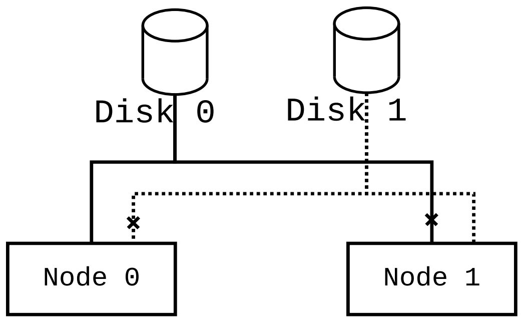


Figure 13.1. Scenario that Leads to Split−Mirror.

We assume in this scenario two SCSI buses as shown in figure 13.1. Every SCSI bus connects both nodes and a disk. In this scenario we lose connectivity between Node 1 and Disk 0, and Node 0 and Disk 1 respectively. This is not an SPOF.

- This double outage will cause the nodes to shoot each others
- Upon reboot they could both become the server

Ideas and issues: consider dual ported raid box for shared state partition redundancy
Improvement: check for network connectivity; if so, it would be smart for them to both get out of the cluster and require manual intervention

13.4 Scenario: Network Cable Pull(s) When Running Cluster

```
if (a subset of the redundant network links survive) {
    nothing happens, no external events generated
}
else {
```

// net partitioned
Heartbeat does the following:

- Node 0: sets internal network state about node 1 to partitioned.
- Node 1: sets internal network state about node 1 to partitioned.

}

Note that this state is not published and no action/message exchange happens between HB and any other daemon or process. When Quorum Daemon cannot see the shared state disk partition, it will ask HB for its view of the network. Heartbeat in this case will reply that the network is partitioned. Quorum Daemon would notify the Service Manager that would take appropriate action.

If HB has set the network to partitioned but QD can see the quorum partition, then nothing happens and services continue running uninterrupted.

13.5 Scenario: Two Node Boot: Full Network Outage or All Hardware Fully Operational

The fact that there's a network outage makes it no different than a normal startup sequence. The following table depicts a timeline where entries on the same horizontal line represent concurrent activity.

Table 13.1. Behavior of nodes at Boot time.

Node 0	***Node 1***
Start Cluster Script	Start Cluster Script
Start Quorum Daemon Start Heartbeat	Start Quorum Daemon Start Heartbeat
DiskService State, reset Service Status	DiskService State, reset Service Status
Check Power Switch	Check Power Switch
Call SM	Call SM

Basically, it relies on the disk locking mechanism: one starts services before the other; both are still cluster members. The node would stop all services for which it is not the preferred node. The preferred node will start those services.

13.6 Scenario: Server Hangs

Say: server 0 hangs

- Node 1 notices node 0's disk timestamp stopped (Quorum Daemon)
- Node 1's Heartbeat also reports node 0 is DOWN
- Node 1 shoots node 0 (Quorum Daemon)
- Node 1 modifies node 0's on−disk state as DOWN
- Node 1's Quorum Daemon calls SM to tell it that node 0 is DOWN
- SM restarts services on 1 after resetting disk service state

13.7 Scenario: Server Panics

- Node 0 panics
- Node 1 shoots it.

Refer to the "Dumps and Panics" section for detail.

13.8 Scenario: Cluster Daemon Dies.

There are multiple daemons that can die. This is different from table 12.1 describing the startup procedure and the behavior when a daemon fails to start. The behavior and remedial in each case are as follows:

Table 13.1. Daemon Failure Scenarios

Deamon	***Behavior***
Power Daemon	The system will continue to run and provide the same services it was already running. Furthermore, if the other node goes down cleanly, it can takeover those services. However when powerd is down, the system is unable to power cycle the other node which implies that it is unable to takeover services in the event of a node failure.
Heartbeat	The cluster service survives. In this case, Quorum Daemon will not lengthen the duration allowed before shooting the partner in the event that the partner stops updating its disk timestamp.
Quorum Manager	Disk timestamp stops. Partner shoots it.
Service Manager	No additional services will be stopped or started on this node.

13.9 Scenario: Site Power Outage

Both nodes lose power at the same time, and we assume that they get power restored at the same time and boot up simultaneously.

Table 13.2. Node Behavior During Power Outage.

Node 0	***Node 1***
Start cluster daemons −starts Quorum Daemon	Start cluster daemons −Starts Quorum Daemon
See other node up	See other node up

This works. Now, in case node 0 starts up before node 1:

Table 13.3. Node behavior During Power Outage (continued).

Node 0	***Node 1***
Start cluster daemons	Busy in fsck
− starts Quorum Daemon − sees other node’s disk state as up − can access power switch	Still busy in fsck...
− observes that QM inactive − shoots partner	Start cluster daemons − starts Quorum Daemon
− sets partner’s disk state to stopped	Gets shot
− takeover services	Starts fsck over again Boots and joins cluster

13.10 Scenario: Planned Maintenance (TBD)

The administrator decides to take down a cluster server for a planned activity. The SM will be responsible for initiating a service stop on a per−service basis. Alternatively, the administrator could manually initiate the relocation of an individual service. For each service:

- The SM performs the application specific stop script (i.e. unmounting filesystems). It waits for the completion of the stop script prior to calling the Quorum Daemon.
- The Quorum Daemon marks in the shared state that it is no longer the server for this particular service. (However it keeps its activity timer updating even if it currently isn’t serving any services.)
- Upon return of a success status from the Quorum Daemon, the SM of the former server contacts the SM on the other node to initiate a service start.

13.11 Scenario: Clean Shutdown

During a normal system shutdown, the Service Manager’s stop function will be called (triggered by SYSTEM V Init−based stop scripts).

- For each service, SM will perform a clean shutdown:
  - First, the application specific stop script (i.e. unmount, stop database)
  - Second, the Disk Library’s Stop service function will be called for each service to set the shared disk state to indicate that it is not the server.
- The Stop script would next cause Quorum Daemon to stop, which marks the node state as DOWN.
- Quorum Daemon on other node notices that the partner node state is DOWN and notifies the Service Manager.
- The SM running on the other cluster member would then pickup the services after they have completed the stop on the original server.

13.12 Scenario: Storage Outage − Data Disk

This case describes the behavior when a disk associated with a shared service becomes inaccessible. Assumption: access to the shared state disk is intact.

Here the application encounters I/O errors, but the Quorum Daemon (and consequently the SM) are oblivious to the error. For this reason, we recommend mirroring at the RAID controller level.

For the first "release" we aren’t doing anything to address this problem. This is another case of a tradeoff between High−Availability and hardware expense. For subsequent releases, we could get more into service monitoring. In that case we could be "pinging" all the devices associated with a service and possibly shutdown the service.

This scenario would benefit from tighter integration whereby the Quorum Daemon could register for error notifications from the low−level (SCSI) device drivers. Since that support is not available in the current SCSI drivers we’re not pursuing this route.

14 Reference

[1] High−Availability Linux Project at http://linux−ha.org.
[2] Timeout Devices, Inc. At http://www.timeoutdevices.com.
[3] Linux Virtual Server at http://LinuxVirtualServer.org.
[4] Swig information and documentation available at http://www.swig.org.
[5] Sorensen, Karla. Kimberlite Cluster Installation and Administration. June 2000. Available at http://oss.missioncriticallinux.com.

15 Appendix A. Messaging Subsystem API

The API provides the following functions.

Function calls and Description

`msg_handle_t msg_open(msg_addr_t dest, int nodeid)`
 	
Description:
Open a communication path to the daemon associated with dest on node 'nodeid.'
Arguments:
`dest` Address to which the connection is made. Currently, one of:
`PROCID_HEARTBEAT`
`PROCID_SVCMGR`
`PROCID_QUORUMD`
`PROCID_ADMIND`
`nodeid` Node where the daemon you wish to contact lives. This value corresponds to that returned from get_clu_cfg(), in the lid field.
Return values:
Upon success, a valid file descriptor is returned. −1 is returned upon failure.

`void msg_close(msg_handle_t handle)` ***Description:*** Close an open msg_handle_t. It is _required_ that you call this, as the msg svc keeps internal tables of file descriptors and associated states. ***Arguments:*** `handle` Handle returned by msg_open or msg_accept which you will no longer use. ***Return Values:*** None.
`msg_handle_t msg_listen(msg_addr_t my_proc_id)` ***Description:*** Create a msg_handle_t which is ready to accept incoming connections. ***Arguments:*** `my_proc_id` Address on which to listen. PROCID_XXX corresponds to a TCP/IP port on which this daemon will listen. ***Return Values:*** On success, a valid file descriptor is returned. On error, −1 is returned.
`msg_handle_t msg_accept(msg_handle_t handle)` ***Description:*** Call accept on a file descriptor returned from msg_listen. ***Arguments:*** `handle` Valid handle returned from msg_listen. ***Return Values:*** On success, a valid file descriptor is returned which describes the new communication channel. On error, −1 is returned.

Function calls and Description
`msg_handle_t msg_accept_timeout(msg_handle_t handle, int timeout)` ***Description:*** Call accept on a file descriptor. If no connections are pending within timeout seconds, the function returns. ***Arguments:*** `handle` valid handle returned by msg_listen. `timeout` time in seconds to wait for a connection. ***Return Values:*** If a connection is pending, a valid file descriptor for that connection is returned. If no connections are pending, 0 is returned. On error, −1 is returned.
`int msg_send(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Send a message over the communications channel described by handle. ***Arguments:*** `handle` Valid handle returned from msg_open or msg_accept[_timeout]. `buf` Pointer to data to be sent. `count` Number of bytes to send. ***Return Values:*** On success, the number of bytes successfully written is returned. On error, −1 is returned, and errno is set according to write(2).
`int __msg_send(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Send a message over the communications channel described by handle. This call differs from msg_send in that it does no sanity checking of internal file descriptor tables. Use this call if you hand craft a connection, and would like the message service to take care of the communications. **Arguments:** `handle` Valid file descriptor. `buf` Pointer to data to be sent. `count` Number of bytes to send. **RETURN VALUES** On success, the number of bytes successfully written is returned. On error, −1 is returned, and errno is set according to write(2).

Function calls and Description
`ssize_t msg_receive(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Receive a message off of the communications channel described by handle. ***Arguments:*** `handle` Valid handle returned by a call to msg_accept or msg_open. `buf` Buffer into which the received data is copied. `count` Number of bytes to read from msg_handle_t. ***Return Values:*** On success, the number of bytes successfully read is returned. On error, −1 is returned, and errno is set according to read(2).
`ssize_t __msg_receive(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Receive a message off of the communications channel described by handle. This call differs from msg_receive in that it does no sanity checking of internal file descriptor tables. Use this call if you hand craft a connection, and would like the message service to take care of communications. ***Arguments:*** `handle` Valid file descriptor. `buf` Buffer into which the received data is copied. `count` Number of bytes to read from file descriptor. ***Return Values:*** On success, the number of bytes successfully read is returned. On error, −1 is returned, and errno is set according to read(2).
`size_t msg_receive_timeout(msg_handle_t handle, void *buf, size_t count, unsigned int timeout)` ***Description:*** Receive a message on a given communications channel. If no message is available within timeout seconds, the call returns. ***Arguments:*** `handle` Valid file descriptor returned from a call to msg_open or msg_accept[_timeout]. `buf` Buffer into which the received data is copied. `count` Number of bytes to read from handle. `timeout` Time in seconds to wait for a message to arrive. ***Return Values:*** If there is data to be read within timeout seconds, the number of bytes read is returned. If the timeout expires before data is ready, 0 is returned. On error, −1 is returned and errno is set according to one of select(2) or read(2).

Function calls and Description
`ssize_t msg_peek(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Check to see if there is data to read from the socket, but do not retrieve the data. For a more detailed explanation, see man 2 recv and search for MSG_PEEK. ***Arguments:*** `handle` Handle returned by msg_open or msg_accept. `buf` Buffer into which the available data is copied `count` Number of bytes to read. ***Return Values:*** On success, the number of bytes available for reading is returned, and buf is filled with that number of bytes from the connection. 0 is returned if there is nothing available for reading. On error, −1 is returned.
`ssize_t __msg_peek(msg_handle_t handle, void *buf, ssize_t count)` ***Description:*** Check to see if there is data to read from the socket, but do not retrieve the data. For a more detailed explanation, see man 2 recv and search for MSG_PEEK. This call differs from msg_peek in that it does no sanity checks on the input parameters. ***Arguments:*** `handle` Handle returned by msg_open or msg_accept. `buf` Buffer into which the available data is copied `count` Number of bytes to read. ***Return Values:*** On success, the number of bytes available for reading is returned, and buf is filled with that number of bytes from the connection. 0 is returned if there is nothing available for reading. On error, −1 is returned.

15.1 Server−Side Routines

A typical server listens for connections, when a connection arrives, it reads the available data, does some processing, perhaps sends a response, and then closes the connection. The message service API was designed with this model in mind. To create a communications endpoint on which to listen, a server makes a call to **`msg_listen()`**, passing its process identifier. This is not a PID, just an internal representation of the services that currently exist. Associated with this identifier is an address which will be used by the message layer, which in our case is an IP address. The return value of this function is a valid file descriptor. Any operations that can be performed on file descriptors can thus be performed on the returned value.

Next, the message service provides a wrapper to the **`accept()`** system call, as well as a wrapper that will do a **`select()`** first. These functions are **`msg_accept()`** and **`msg_accept_timeout()`**, respectively. The return value for **`msg_accept()`** is the same as that for the **`accept()`** system call. The return value of

`msg_accept_timeout()` is a bit different. It will be the same as those values returned from **`accept()`** if there is a pending connection, otherwise it will be the return value of **`select()`**. So, if a connection is pending, a file descriptor will be returned to the caller. If not, the return value of select is propagated. Generally, zero will be returned, unless an error occurs, in which case the caller will be returned a negative one.

After a successful call to accept, the caller can read data with the **`msg_receive()`** call. This call mimics the read system call in every respect. The **`msg_write()`** routine, thus, mimics the write system call. All return values are identical to the corresponding system calls, and that includes `errno` values as well.

To close a connection, the **`msg_close()`** routine is provided. It simply removes the give handle from the message service's internal data structures and issues a close on the file descriptor.

15.2 Client−Side Routines

A typical network client will establish a communications channel to a server, send a request, read a response, and close the connection. The calls to perform this are **`msg_open()`**, **`msg_send()`**, **`msg_receive()`**, and **`msg_close()`** respectively. **`msg_send()`**, **`msg_receive()`**, and **`msg_close()`** are documented in the above section, so only **`msg_open()`** is documented here.

The **`msg_open()`** routine takes two arguments. The first argument tells which process or daemon on the system the client wishes to connect to, and the second argument specifies what node in the cluster the process is running on. The node id can be obtained from the **`get_clu_cfg()`** call provided by the clustinfo subsystem.

16 Appendix B. Power Switch API

16.1 Discussion

The issues discussed in this section relate to Figure B.1.

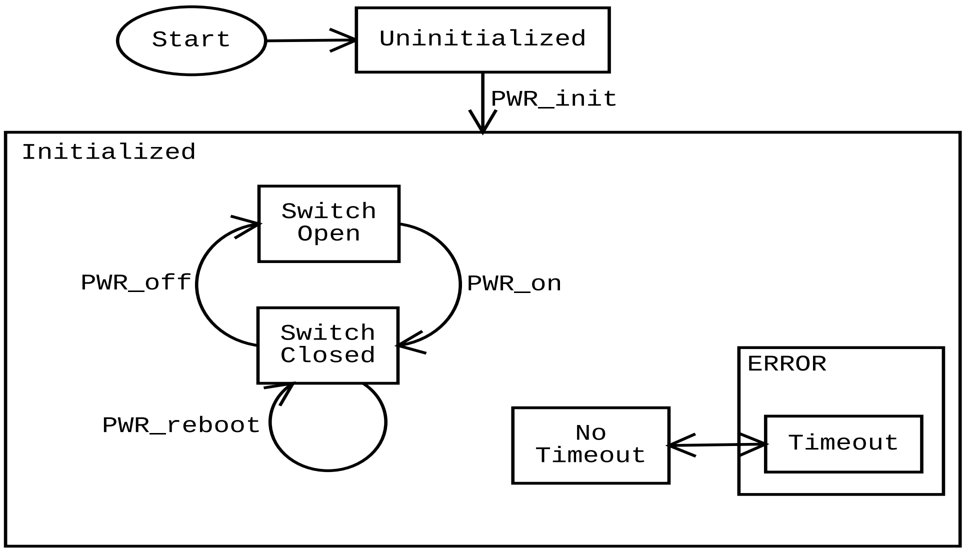


Figure B.1. Power Switch State Diagram.

1. All commands are synchronous, i.e., blocking.
2. Initialization. Assuming that processes will not contend for access to the remote switch, and that there is only one such switch per machine, it is assumed that initialization involves opening the serial device, creating a lock file for the device, setting parameters on the serial device, and sending and receiving an initialization string to/from the switch.
3. Status and results. Certain functions return a PWR_result, which is a set of flag bits whose positions are taken from the PWR_result_codes enumeration. For each flag position, the status is to be taken to be true, available, or on, as appropriate, for that flag if its bit value is one, otherwise the status is to be taken to be false, unavailable, or off.
4. Sequencing. After PWR_init has been called and returned PWR_TRUE, any of the other functions may be called in any order. Redundant calls to functions will return the status after the call regardless of whether the status of the switch changed as a result of the call. For example, a call to PWR_off, followed after some interval by a call to PWR_off will return true in each case if after the call, the switch is in the open condition.
5. Time−outs. Synchronous calls block until they fail, succeed, or time−out. Currently, the default time−out for initialization is 10 sec.; the default time−out for other commands is 1 sec. Time−outs may be altered through configuration file entries.

Note: The boolean type, `enum PWR_boolean { PWR_FALSE, PWR_TRUE }`, and the result, `int PWR_result`, consist of the following bits:

```
PWR_SWITCH  = 1,  Switch is closed=1/open=0.
PWR_ERROR   = 2,  A command has put the interface into an error condition.
PWR_TIMEOUT = 4,  The most recent command sent to the
                  switch timed out without a response.
PWR_INIT    = 8,  1 if interface has been initialized.
```

The following table describes function calls and functionality.

Table B.1. Power Switch Function Call Description.

Function call	***Description***
`PWR_configure`	Configure the power interface. Currently there are three options recognized: 1. power.device: the pathname of the serial port character special device file. 2. power.init_timeout: the time−out, in sec., for initialization. 3. power.timeout: the time−out, in sec., for non−initialization calls through the interface. **PWR_configure()** can be called with: 1. The name of a configuration file to read for parameters. 2. NULL, or 3. a null string.
`PWR_init`	Initialize the power switch, and the communication paths to it.
`PWR_release`	Release resources associated with the module, and close the serial device file. PWR_release should be called when the connection to the power switch is no longer needed so that the serial port lock file can be released.
`PWR_status`	Query the switch to determine its current status, and its readiness to carry out additional commands. Returns a PWR_result that contains flags that indicate whether the switch is open or closed, and whether the last command to the switch timed−out.
`PWR_reboot`	Send a command to the switch that will cause the supplied system to reboot. Normally, this means that the switch will be opened for some period of time, after which it will be closed again. If the switch is already open when the reboot command is issued, the switch remains open for 5 or ten seconds, then closes.
`PWR_off`	Send a command to the switch that will cause the switch to open, removing power from the affected system.
`PWR_on`	Send a command to the switch that will cause the switch to close, supplying power to the affected system.

17 Appendix C. Disk Locking Subsystem

17.1 Overview

The disk locking subsystem provides a means for ensuring the data on the shared raw partition is accessed atomically with respect to both multiple services on one node, and multiple nodes accessing the same shared disk. The lock is needed whenever accessing shared state data, whether for read or write, so that a consistent view of the cluster is shared by all services on all nodes.

17.2 API

The locking API is simple, and should be familiar to those who have experience dealing with spin−locks.

17.2.1 Locking Primitives

At the very basic level, we have a command to take the lock, and one to release it. The lock is essentially a spin−lock. The function **`clu_lock()`** is called to take the cluster lock, and the function **`clu_unlock()`** is called to release the same lock.

17.2.2 Callers

The service manager is the primary caller of the locking routines, as the SM is responsible for maintaining service status information across the cluster. Noteworthy is the case where the SM starts off several threads of execution, each which needs to access the on−disk service database. In this case, several threads will attempt to take the cluster lock. To provide serialization around the on−disk lock, we also implement a node lock, discussed in the section 18.3.

17.3 Locking Algorithms

The locking algorithm is very basic, and works very well for a two node cluster. To serialize access to the disk lock on a per−node basis, we have a node lock. Once a process successfully takes the node lock, it may then attempt to get the disk lock. After the disk lock is released, the node lock is released, and other callers can try for the lock.

17.3.1 Node Lock

The node lock provides serialization on a per−node basis to the disk lock, as only one caller per node can try for the disk lock at any given time. The node lock is implemented using the POSIX.1 **`fcntl()`** system call. There is a pre−defined cluster lock file that is used when taking and releasing the node lock. This lock must be held before the disk lock can be taken, and must be held for the entire duration that the disk lock is held. Once the disk lock is released, we may release the node lock.

17.3.2 Disk Lock

There are two lock blocks on the disk, one for each node. When attempting to get the cluster lock, we simply write a value of ’1’ to our lock block, and follow that by a read of the partner node’s lock block. If its lock bit is not set, we can safely take the cluster lock. If its lock bit is set, we implement a random back−off algorithm. After a random period of time (not exceeding a predefined value), we try for the lock again. There is a lock timeout that, when exceeded, causes the node to shoot down his partner. This is to prevent the case where the partner node fails to release the cluster lock. Most notably, this would be due to a bug in the cluster code, not a hung system. We have other methods for detecting that a system is hung and cleaning up the on−disk data associated with that member.

17.4 Scalability Implications

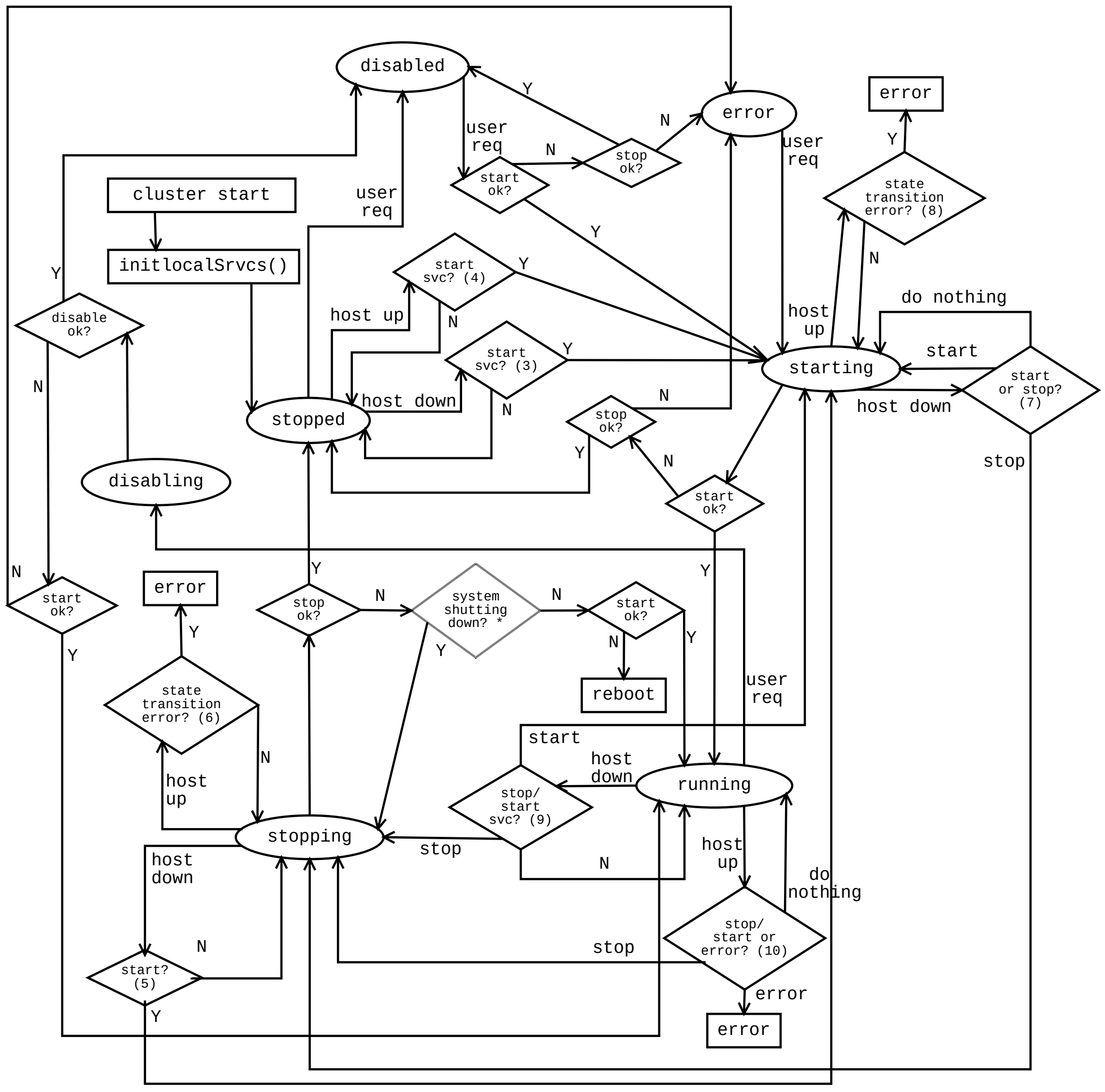
The lock code is presently hard coded to two nodes.

The locking algorithm now performs a synchronous write of 1 into this node’s lock cell and then reads the other node’s cell. If the other node’s cell is zero the lock is obtained otherwise this node’s cell is written synchronously to zero and a random delay ensues before trying again.

For more than two nodes, the algorithm is the same except that in the read cycle all of the other nodes’ lock cells are read in node number order. Then, they must all be zero to obtain the lock.

18 Appendix D. State Diagram for Service Manager

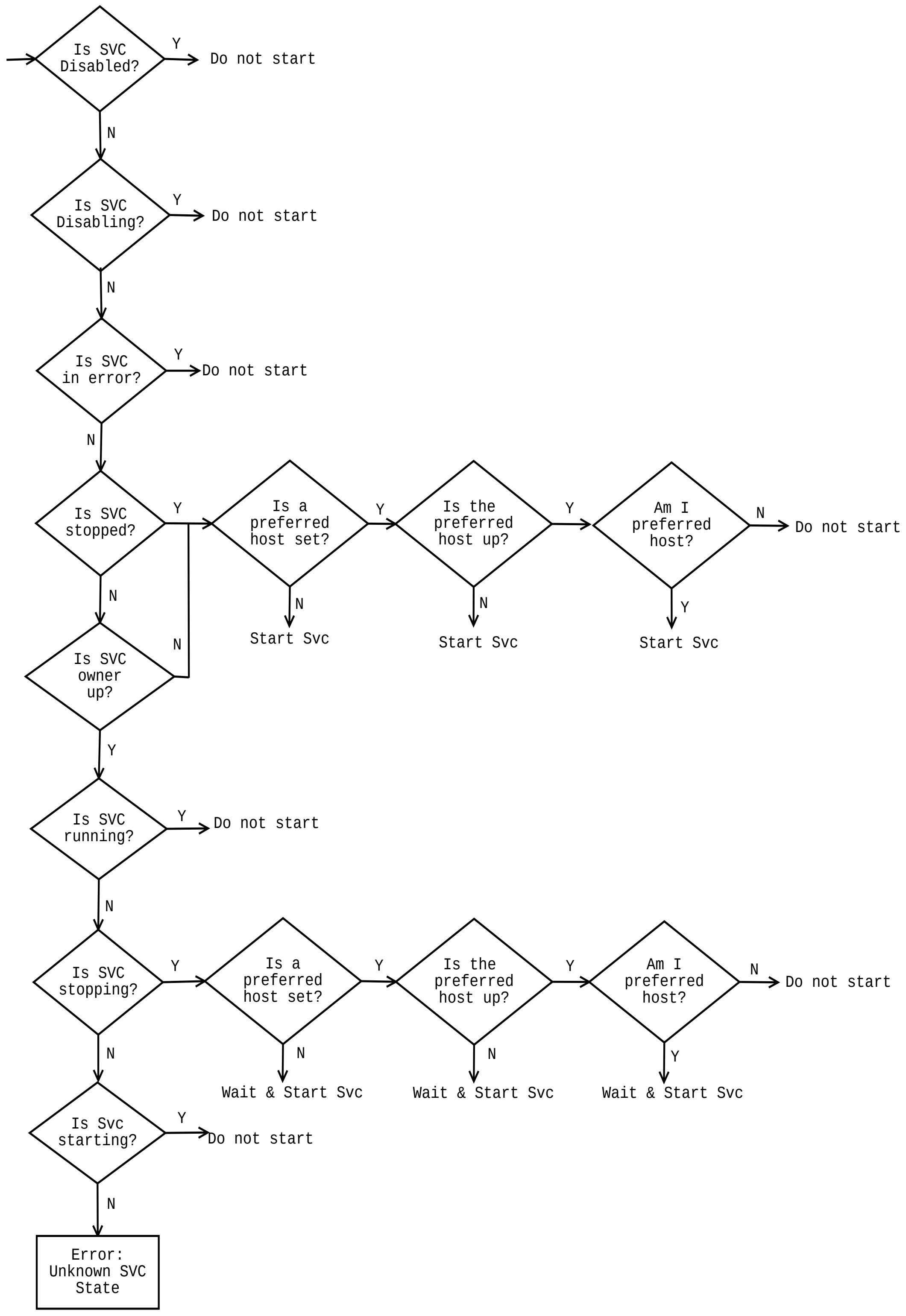
18.1 Complete State Diagram



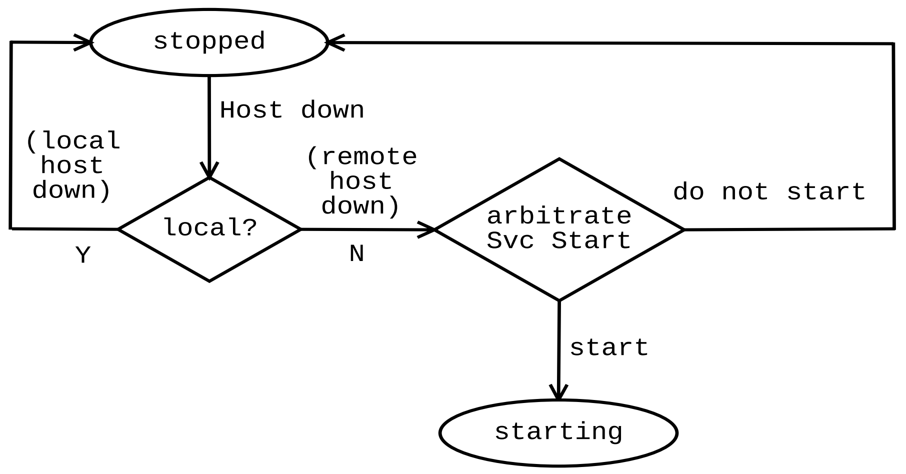
* = Not Implemented Yet

Numbers appearing in the state diagram refer to the blown−up view shown in a later section. The state diagram is blown up to several state diagrams represented in sections 18.2 through 18.10, as referred to in this figure.

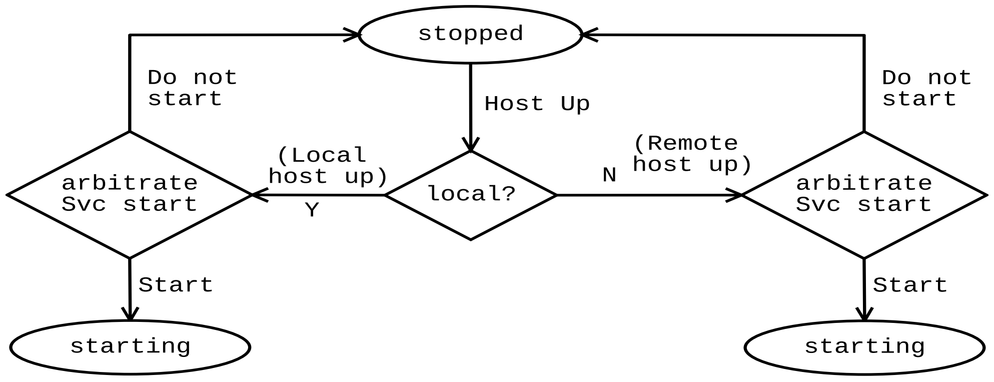
18.2 Service Start Arbitration



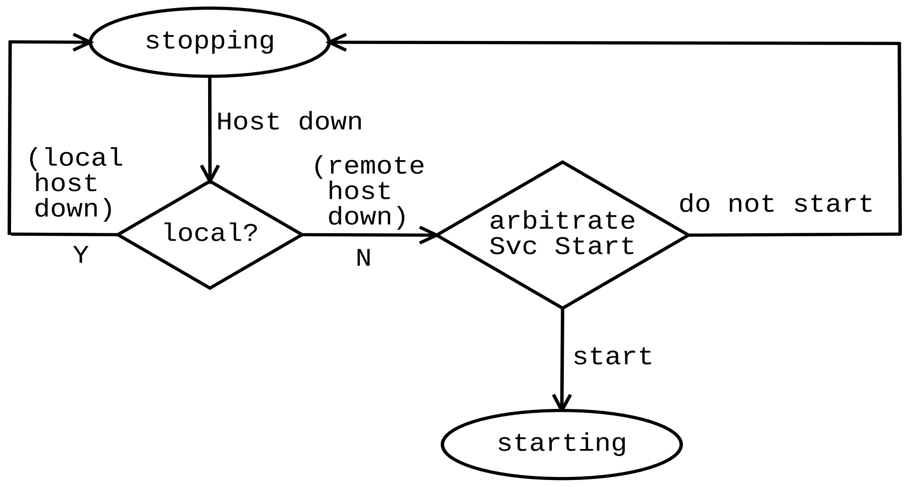
18.3 State: Stopped, Event: Host Down



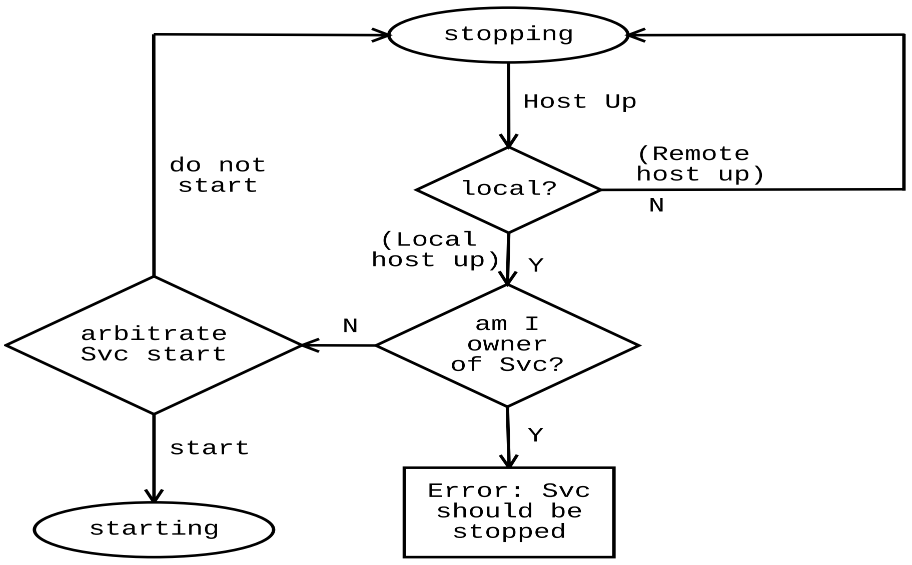
18.4 State: Stopped, Event: Host Up



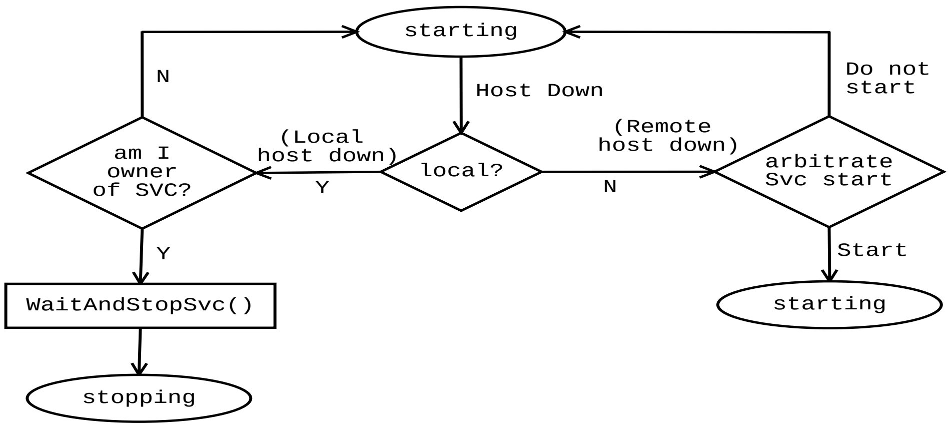
18.5 State: Stopping, Event: Host Down



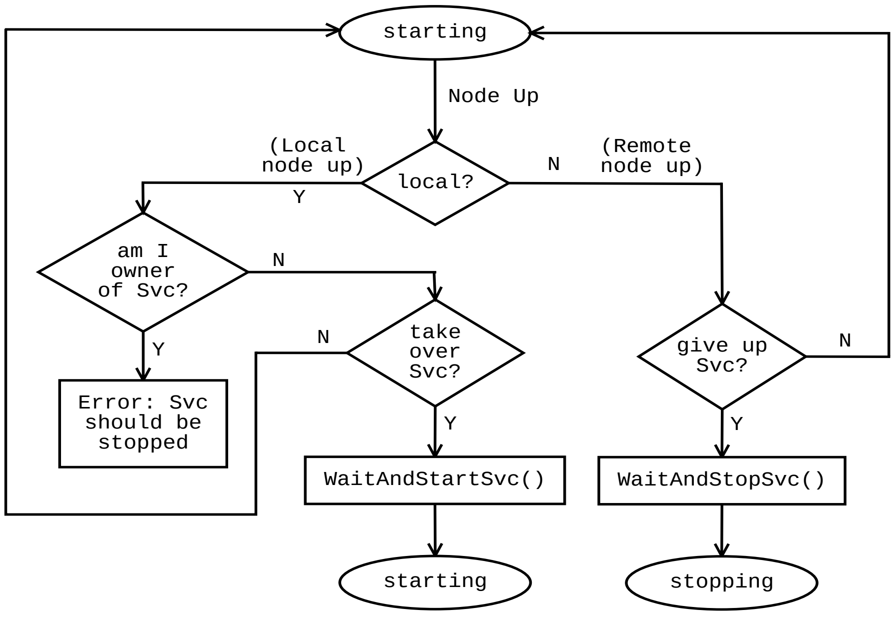
18.6 State: Stopping, Event: Host Up



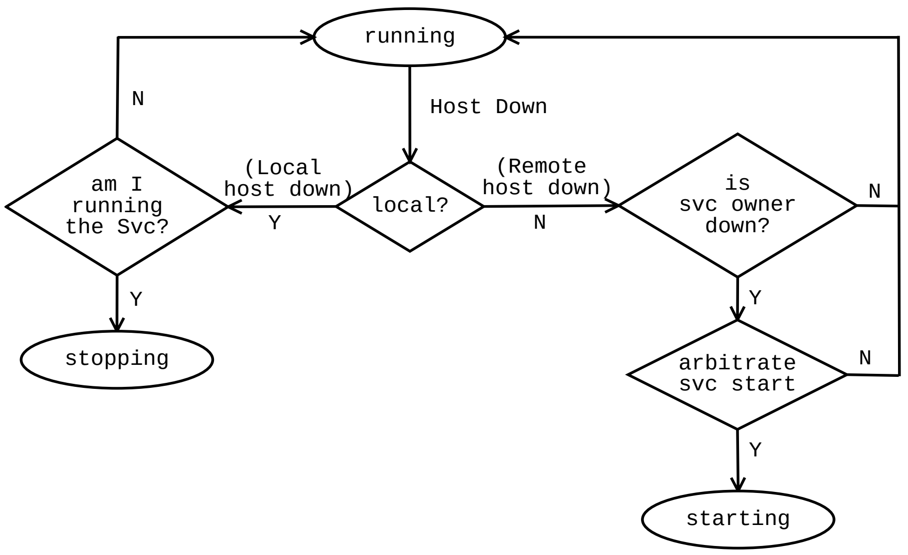
18.7 State: Starting, Event: Host Down



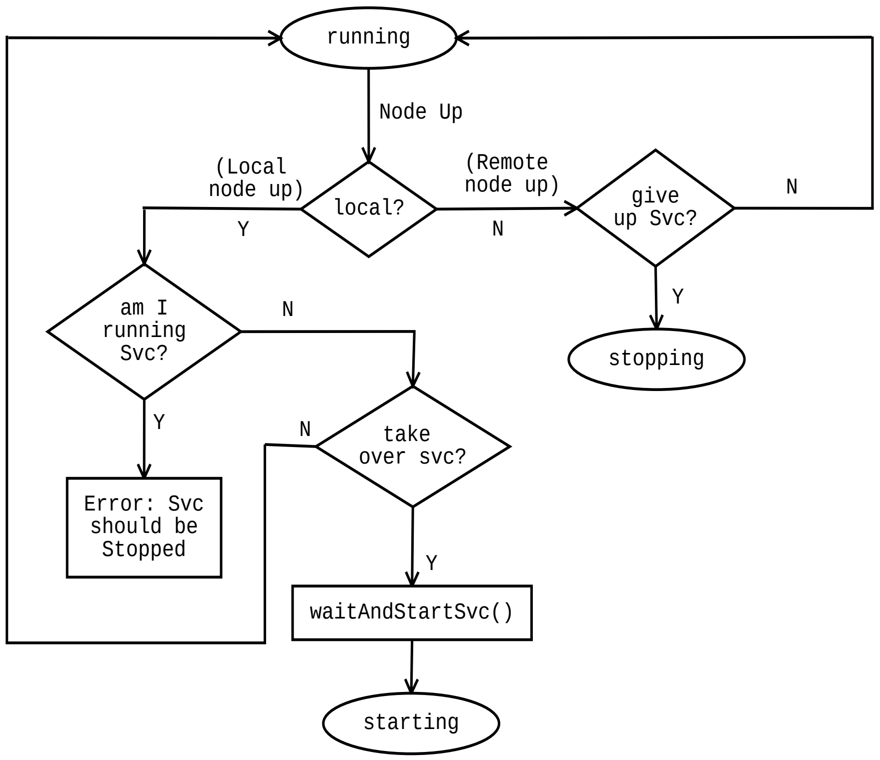
18.8 State: Starting, Event: Host Up



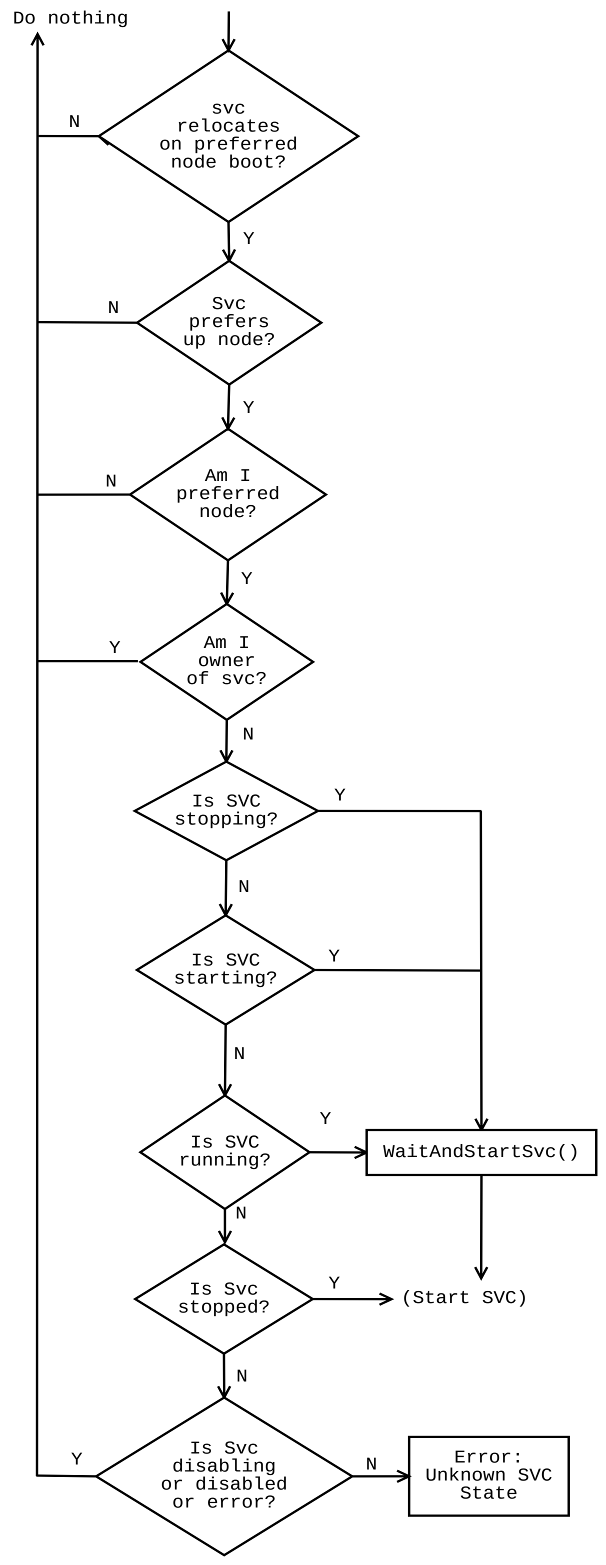
18.9 State: Running, Event: Host Down



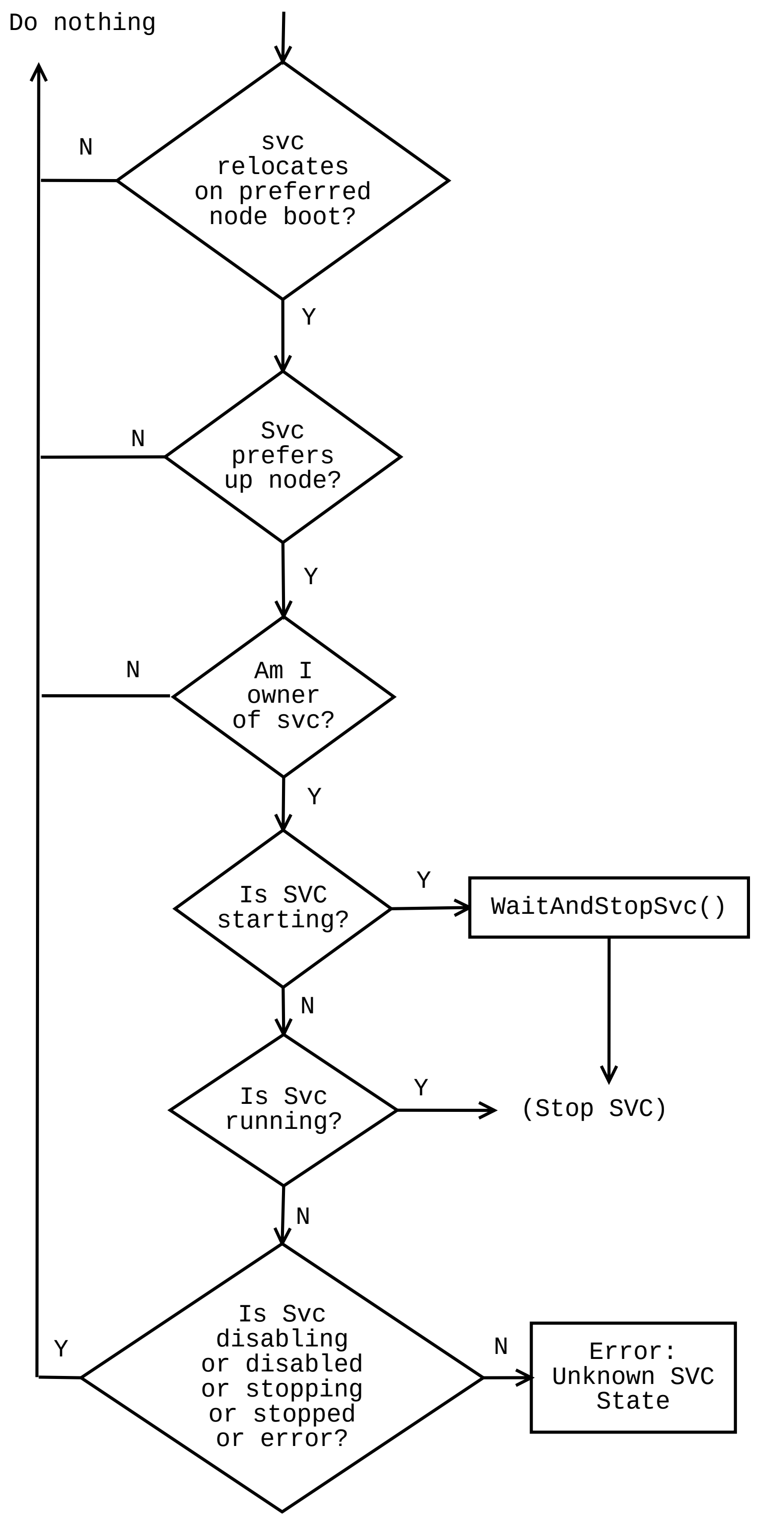
18.10 State: Running, Event: Host Up



18.11 State Diagram for Service Takeover



18.12 State Diagram for Giving up Service



19 Appendix E. Design Notes on Disk−Based Configuration Database

19.1 Problem Statement

All cluster members need access to a description of the cluster configuration information. This configuration information contains:

1. Node specific configuration − for example, node name, what devices are associated with the heartbeat channels and the device special files representing the shared disk partitions.
2. Cluster−wide configuration info. This principally includes the service descriptions.

19.2 Implementation Options

1. Represent all the configuration data in a file on each node’s filesystem. While this is the simplest to initially implement, it requires the administrator to manually keep the files in sync. Beyond manual copying we could implement infrastructure to try to automatically copy the files between cluster members. This can quickly run into problems in cases where all the cluster members are not up. In a worst case ping−pong scenario with configuration modifications in−between you never really know who has the most recent copy.
2. Since we already have a shared disk for cluster state and runtime service descriptions we can extend that model to having the configuration description residing on the shared disk partition. The principal benefit is that there is only a single copy of the configuration information which is shared among cluster members and therefore we don’t have to worry about keeping two separate copies in sync. Additionally, it also affords an opportunity to make configuration modifications to a down node (whereas if the configuration file was on a file in the local filesystem of the down node you likely do not have access to it).

19.3 Implementation Approach

We have decided to implement approach #2; to have all cluster configuration information represented on the shared state disk. (Actually, it’s not really ALL the information as the specification of the shared state disk partitions must reside outside of the disk partitions themselves.)

19.4 Low−Level Implementation

At the low−level, there are routines in the diskstate library (which supports all reading/writing to the shared cluster partition) which provide the following support:

- Ability to WRITE the configuration description in its entirety. There are no facilities for writing only a portion of the configuration description.
- Ability to READ the configuration description in its entirety. There are no facilities for reading only a portion of the configuration description.
- Locking primitive to ensure that only one process in the cluster at a time is modifying the configuration information.

Both checksumming and a shadow copy are transparent to the callers of the configuration description READ/WRITE APIs.

19.5 Higher−Level Implementation

All of the cluster daemons and management utilities interact with the cluster configuration information through the **`metaconfig`** library. This provides a centralized location to abstract the fact that the configuration is now on a raw disk partition rather than in a file in the local filesystem.

The following schemes can be used by **`metaconfig`** and the management utilities to safely access the configuration information:

- Take out the lock, read in the configuration information through the low−level API.
- Then release the lock if there is no modification involved.
- If there is modification involved, the lock must be released quickly:
  - If you have the new values in hand (i.e. specified on the command line) then modify the memory−resident configuration description and then write out the new configuration description. Then release the lock.
  - If you do not have the new values in hand (i.e. they are being prompted for) then stash off a copy of the original configuration in memory and release the lock. Now the user (gui) can modify a memory resident configuration description at their leisure. When the user prompting is done (i.e. "Commit" button has been pressed) then take out the lock, read in the configuration description from disk. Compare it to your original snapshot; if it differs then there were 2 people attempting to modify the configuration at the same time. In this (presumably rare) situation, tell the user about the conflict and require them to start the configuration process all over (rather than implementing a complex merging algorithm). If there are no conflicts, write the new config and release the lock.

19.6 Usage Instructions

To use it, one needs to link in with the libdiskstate.a. The relevant source file implementing the APIs is in diskstate/diskconfigdb.c. The following excerpts describe the interface.

There will be a separate configuration file used to represent ’bootstrap’ parameters. This will include the definition of the cluster shared disk partitions. The remainder of the cluster configuration parameters will reside on the shared disk.

Function Call	***Functionality***
`ssize_t getDatabaseLength()`	Returns the length of the current "data" in the configuration database. *Return values:* −1: Unable to read a database header describing the length. This occurs if the shared disk partition has never been initialized or an I/O error occurs. 0: the database is currently initialized, but is empty. >0: the length of actual database contents
`ssize_t writeDatabase(char *data, ssize_t length)`	Write the service block out to disk. This routine also provides the ability to delete (clear out) the contents of the database by passing in a length parameter of 0. *Return values:* −1 on I/O error, −2 on parameter error, the number of bytes written on success
`ssize_t readDatabase(char *data, ssize_t max_length)`	Read in the configuration database off the shared disk and populate the user's buffer with the data. This of course requires that the user's buffer is big enough to hold the contents of the database. In order to facilitate this, the user can first call getDatabaseLength(). Just to avoid any buffer overflows, the max_length parameter describes the user buffer size. *Note*: the database is read in as a single "blob", there are no facilities for retrieving a portion of the database (i.e. Records). *Return values:* −1 on I/O error, the number of bytes red on success.

20 Appendix F. Design Alternatives

Briefly, here are some alternative design approaches that we considered and ultimately rejected.

- **Linux−HA Heartbeat** [1] − heartbeat to other cluster member over redundant communication channels (Ethernet and serial).
  - Pros: takes multiple communication channel outages to declare another host dead. Uses commodity hardware.
  - Cons: Unable to distinguish between a remote node down from a network partition. This will result in the same service being run concurrently on several nodes.
- **Software Watchdog** − using /dev/watchdog, reboot the system in response to a cluster node becoming inactive.
  - Pros: No extra hardware
  - Cons: In the event of a system hang the softclock could become suspended in time so a node may not know it was even hung. Very weak and unpredictable I/O barrier semantics.
- **Pseudo Driver** − insert a pseudo driver above all I/O operations to shared cluster storage. Have a background kernel thread which queries the PC's hardware clock chip to see if too much time has elapsed; if so reboot. The pseudo driver would not initiate any new I/O operations if the elapsed time has been too long.
  - Pros: no extra hardware
  - Cons: low level I/O's already queued to the (SCSI) driver could be immediately issued upon clearing of the hang.
- **Hardware Watchdog**
  - Reference [2] discusses several such devices − this is basically a power strip with a serial connector. When armed, if you don't heartbeat it in the programmed time interval, it will power cycle.
    - Pros: Effective I/O Barrier
    - Cons: Difficult for the surviving cluster member to know that the other node has taken itself out. Therefore its hard to know when its safe to startup the service on the surviving node.
  - Another form of hardware watchdog I found is a PCI card. Arm it when starting up cluster services and probe its I/O space address. Failure of timely probes will assert an external pin on the card. Wire up this external pin to the system's reset switch. Some systems don't have easy wire accessibility for their reset switch which may require soldering this lead. In addition, device drivers and PCI slot space are a consideration.
- **SCSI Reservations**
  - Pros: Allows service level granularity. Effective I/O barrier.
  - Cons: Linux support is immature. Historically we found this often required a narrow set of qualified disks & controllers to be operational. SCSI reservations were a constant sore−spot. If we go this route it would likely require a dedicated SCSI developer and would unlikely meet the targeted timeframes.

21 Appendix G. Lexicon

QD:	Quorum Daemon
SM:	Service Manager
SPOF:	Single Point Of Failure