Linux−HA Heartbeat System Design

Alan Robertson − *SuSE Labs* − <alanr@suse.com>

ABSTRACT

One of the most commonly identified features which is felt to be necessary for Linux™ to be considered "enterprise−ready" is High−Availability. High−Availability (HA) systems provide increased service availability through clustering techniques.

HA clusters minimize availability interruptions by quickly switching services over from failed systems to working systems, providing the customer with an illusion of continuous availability. As such, high−availability features, are vital to mission−critical systems. Although there are many components to a high−availability system, two of the key components are heartbeat services and cluster communication services. Heartbeat services provide notification of when nodes are working, and when they fail. In the Linux−HA project, the *heartbeat* program provides these services and intracluster communication services.

This paper describes the design of the *heartbeat* program which is part of the High−Availability Linux Project with particular emphasis on the rationales behind key design choices, and the results obtained.

Introduction

As Linux™[1] grows into handling larger server systems satisfactorily, it will have to provide many of the same services which these larger servers by Sun, Compaq, IBM, and others have traditionally provided. One of the key features which these larger and more mission−critical servers have provided customers is high−availability (HA) clustering.

A high−availability cluster is a group of computers which work together in such a way that the failure of any single node in the cluster will not cause the service to become unavailable. Given this definition, it seems obvious that it is necessary for the cluster to detect when servers fail, and when they become available again. This task is performed by code which is usually called "heartbeat" code. In the case of Linux−HA, this function is performed by a program called *heartbeat*. Heartbeat programs typically send packets to each machine in the cluster to indicate that they are still alive.

Another of the most basic functions which any High−Availability system must perform is cluster communications. It is often the case that these communications need to communicate between all cluster members at once in a broadcast or multicast sense.

The Linux−HA *heartbeat* program takes the approach that the keepalive messages which it sends are a specific case of the more general cluster communications service. In this sense, it treats cluster membership as joining the communication channel, and leaving the cluster communication channel as leaving the cluster. Because of this, the heartbeat messages which are its namesake are almost a side−effect of cluster communications, rather than a separate standalone facility in the *heartbeat* program. It should be emphasized that h*eartbeat* should not be understood as a complete cluster management solution, but a basic component providing certain well−defined low−level services. These services are outlined in more detail below.

Heartbeat Design Philosophy

The *heartbeat* component of the Linux−HA project **[Rob00]** is in some senses a simple program. It is one of the the lowest−level components of the system, and has the purpose of being reliable, so it is important that it be simple and straightforward. It should be designed to run continuously for years without memory leaks, or bugs. It needs to be easy to understand, easy to debug, and extremely robust. For this reason, when design alternatives were considered, the simplest, most straightforward, and easiest to debug were often chosen.

Even though this low level subsystem is reasonably simple, there are some non−obvious design decisions and synergies which were made which appear to be worth understanding. It is the intent of this paper to explore some of these elements of the design, and talk about how it may be extended in the future.

1 Linux is a trademark of Linus Torvalds.

Definitions

The following definitions will prove useful in the discussion of the design of the Linux−HA *heartbeat* program.

Heartbeat Subsystem
A subsystem which monitors the presence of nodes in the cluster through a series of keepalive, or heartbeat messages.

Cluster Manager
A subsystem which manages the cluster, deciding which resources are on which nodes, and taking appropriate action during cluster transitions. In this document, this term is used to refer to an entire range of cluster functions and services which are direct or indirect clients of heartbeat's API.

Cluster Transition
A cluster transition occurs whenever a cluster member enters or leaves the cluster. This event triggers recovery and reconfiguration actions by the cluster management software.

Low−Level Services in High−Availability Systems

Every high−availability system needs two basic services: to be informed of cluster members joining and leaving the system, and to provide basic communication services for managing a cluster. As is discussed below, there is considerable synergy between these two functions. In the discussion below, application data is specifically excluded from consideration. All that is under discussion here is cluster control data, that is, data used for managing and controlling the configuration of the cluster.

Low Level Cluster Membership

The lowest level concept of cluster membership is typically founded on the idea of reachability: Can we communicate with node X? If so, it is considered to be a member of the cluster. If not, it is considered "dead". Towards this end, one conventionally creates a heartbeat subsystem to determine if nodes are alive or dead. In such a system, each machine which is actively a part of the cluster creates heartbeat messages on a periodic basis, and sends them out to all cluster members. Any machine whose heart can no longer be heard to beat (i.e., from whom messages are no longer heard) is considered to be "dead", or have left the cluster. Such events (members leaving or joining the cluster) trigger cluster transitions.

Typically, this means sending out messages from every machine to every machine on a periodic basis (perhaps once per second or so). For large clusters, there is considerable advantage in using multicast or broadcast techniques to avoid the traffic associated with the $O(N^2)$ operation of sending messages from every machine to every machine. In a stable high−availability system (one not in the midst of a transition) these messages are virtually the only messages being sent. Indeed, these messages constitute the overwhelming majority of the communications traffic in a cluster system. Therefore, although these messages are simple, in large clusters, become quite voluminous if not handled carefully. Ultimately, this lowly heartbeat function can limit the scaleability of a high−availability system, particularly if not handled efficiently.

Note that this form of low−level (or local) cluster membership is not sufficient for a complete cluster management infrastructure, it needs to be combined with the idea of consensus or agreement wherein the members of the cluster communicate with each other and exchange their views of cluster membership, resulting in a cluster−wide view of cluster membership. This is discussed further in the *Future Directions* section later in this paper.

Communications in High−Availability Systems

Every high−availability cluster has needs to communicate in order to operate. This communication varies in content and size depending on the particular cluster management system being used in the cluster. As an example, the current heartbeat code has a message type to request that a particular resource group be relinquished, and a corresponding response message. Other resource−related messages also commonly occur during cluster transitions. Although this depends greatly on what cluster management model the cluster has adopted, the majority of (non−heartbeat) cluster management messages are associated with cluster transitions.

Since cluster management is typically transaction oriented, with the transactions spanning the entire cluster, it is common to communicate with all nodes in the cluster simultaneously. For example, one node might note an event which should trigger a cluster transition. This node would then notify all nodes in the cluster to enter a cluster transition. In this process, it would typically send a message to all cluster members, and then await messages from all cluster members acknowledging this message.

This is typical of cluster management messages. Th ey typically have a transactional nature across the entire cluster. The most common pattern is to send a

message to all nodes, and then await responses from all nodes participating in the event. This is the model IBM follows in their Phoenix HA system. A similar model is followed by TweedieCluster barrier services. If you combine this observation with the information that heartbeat information is typically broadcast to the entire cluster, it becomes readily apparent that by far the majority of high−availability packets are sent to the entire cluster. To give some sense of how rarely unicast packets occur, it would be surprising if as many as 1 packet in 100 in a normally−operating HA system is a unicast packets.

Communication Reliability

Cluster communications are the backbone around which one builds a high−availability system. If they are not reliable, then it is quite obvious that the cluster itself will will not be reliable. Indeed, there is a whole class of problems around cluster partitioning (also called split−brain syndrome) which are made significantly less likely by good cluster communications.

It is normally considered desirable to detect a cluster node failure within seconds. For example, in Microsoft's WolfPack system, the time to discover a node is dead is less than 5 seconds. In practice, this means that heartbeat messages should be delivered in significantly less than this amount of time, *even in the presence of single communication failures*. Unfortunately, this is difficult with routing protocols, which are generally tuned to less stringent requirements, and aren't keyed to providing this information to an integrated system. Routing protocols also only deal with IP packets, which don't include "raw" serial ports. Since the code avoids relying on routing protocols for failover, potentially complex interactions between the configuration of a cluster's internal communication channels and the configuration of the surrounding network are minimized.

Moreover, in HA systems, one also wants to know that the backup links are also still working, and report this information to the cluster administration system. This significantly reduces the chances of multiple failures from which the cluster cannot recover. For example, if someone unplugs a backup communications cable and a month or two later, the primary communications link fails, the system will be unable to communicate. This failure to communicate would have been completely avoidable by reporting the failure of backup links as well as the failure of active communication links. In this way, the backup link can be repaired before the primary link also fails.

Many cluster systems also implement heartbeat mechanisms which work on independent communication methods (for example, raw serial ports) in addition to supporting ethernet for heartbeats. This is considered a best current practice, and is strongly recommended by the Linux−HA HOWTO **[Milz99]**. The reasons for this are several fold:

- Failures in the IP communication subsystem are unlikely to affect the serial subsystem
- Serial ports do not require complex external equipment or external power
- Serial ports are simple devices and very reliable in practice.
- Serial ports can be easily dedicated to cluster communications, and are not subject to significant variability in message delivery latency due to exponential backoff algorithms required by CSMA/CD media like ethernet.
- It is difficult to accidentally unplug a properly screwed−in serial connector.

However, in spite of these advantages, it should be understood that serial port communication is less well−suited for large clusters, and should be viewed as simply another tool in the tool box, to be used where it is appropriate. Large clusters may need very high speed UARTs, or nearest−neighbor heartbeat techniques which minimize required bandwidth.

Summary of Cluster Communications Observations

- Heartbeats (keepalives) are the overwhelming majority of non−application cluster control messages.
- The overwhelming majority of all cluster messages go to all cluster members
- Cluster transition messages commonly consist of a cluster−broadcast message, with a set of unicast results.
- Reliable communications are essential in a cluster
- Single communications failures should not disrupt cluster communications even momentarily
- Backup communications methods should be frequently verified for correct operation, and failures reported through the administration interface.
- Multiple, independent communication paths should be supported by the software, serial ports being the most common alternative choice
- In general, simple methods are preferred to complex methods

These are the primary considerations which led to

the communications design which *heartbeat* imple−ments.

Major Heartbeat Design Decisions

- Support many types of communication proto−cols, including non−IP protocols like raw serial ports
- Send all messages across all communications paths all the time. Report link failures, even when redundant links are still working.
- Send all messages to all nodes all the time. Ig nore messages for other nodes.
- Support reliable multicast messaging
- Use heartbeat messages as keepalives on the communications links in the design of the re−liable messaging.

Although it is apparent at this point why most of these design decisions were made, the last design de−cision is not yet obvious from the text presented above. To fully understand the what this means and why it was chosen, a more full exposition of the mul−ticast protocol must be given. This will be given in the next section.

The particular type of serial port implementation chosen configures the serial ports in a bidirectional ring, similar to a FDDI ring. In this configuration, the number of nodes in the cluster are limited by the speed the serial ports can be run. For current message sizes, and links running at 56 Kbits/sec, this limits the maxi−mum size of clusters that rely on serial ports to something over 30 nodes. This is a fairly respectable cluster size. Of course, care must be taken with sys−tem placement or the wiring of such a serial ring will become unmanageable long before this limit is reached.

Heartbeat's Reliable Multicast Protocol

Protocols have a number of characteristics which can be used to describe them. The current heartbeat protocol has the following characteristics:

- Multicast−aware
- Guaranteed packet delivery
- Packet ordering is not guaranteed
- Flow or congestion control is not provided

Although the reasons for the first two decisions are readily apparent from the nature of a high−availability cluster as discussed above, the latter two are not so immediately obvious. At this point, packet ordering is not required because of the strict request/response na−ture of the cluster management functions which might be put on top of it. This also obviates the need for flow control, since further packets are not typically sent until responses are received from previous pack−ets.

There are basically two techniques described in the literature **[Weiss00**], **[Dan94]**, **[Ram87]** for designing multicast protocols:

- Sender−initiated
- Receiver−initiated

In sender−initiated multicast protocols, the receivers of data typically send acknowledgments for packets. The sender then maintains timers and retransmits packets for which acknowledgments are not received within some fixed period of time. This class of proto−cols is subject to flooding senders with acknowledgments with every single packet sent. P. B. Danzig **[Dan94]** designates this flooding phenomenon as sender (or ACK) implosion. The fact that it occurs consistently with every packet sent is burdensome and makes this technique unsuitable for use in *heartbeat*, since it occurs at every machine with every packet, and could double the number of packets sent. Because of timing considerations, media collisions are very likely to be generated in large quantities as well.

In receiver−initiated protocols, the receivers of communications are responsible for detection of er−rors. In this scheme, sequence numbers are used to detect packet loss. When a lost packet is detected, the receiver requests that the sender retransmit the packet. In this method, senders are subject to being flooded with NACKs (NACK implosion) if the packets do not reach any receivers. This can lead to a high load on the sender, and excessive retransmissions. In 1987 Ramakrishnan et al. proposed a scheme to avoid NACK implosion **[Ram87]**. In this scheme, retrans−missions are limited by timers.

A variant of this scheme is implemented in *heart−beat*. In *heartbeat*, each receiver requests a packet's retransmission no more than once per second, and in turn, each sender will re−transmit (by cluster broad−cast) each packet no more than once per second as well. In this way, NACK implosion is strictly limited. HA control protocols do not have many of the require−ments of some of the secure multimedia streaming protocols such as RTP **[Weis00]**.

In an HA cluster, non−heartbeat control messages are rare. In some cluster management structures, no control messages are sent until a failure (cluster tran−sition) occurs. This means that it could be many months or even years between messages. This can make detection of lost packets by sequence numbers problematic. As a result, it is quite useful to transmit the heartbeat messages using the same communication channel as the control messages, and share the same sequence numbers. This comes from the fact that heartbeat messages are transmitted on a frequent,

regular schedule. In this way strong bounds are placed on the amount of time which might elapse before a lost packet is detected. If the heartbeat messages did not ride across the same channel as normal messages, this would be more complex to guarantee.

This provides significant synergy between heartbeat communication and higher−level messages, and simplifies the implementation of the protocol. Indeed, this approach has been quite effective, and has limited the corresponding protocol code in *heartbeat* to a few hundred lines of code.

Heartbeat's Authentication Scheme

Since cluster members must be able to trust each other completely, a cluster communication system needs to have either physically secure communications, or communications which cannot readily be spoofed. Depending on the cluster manager, it is easily possible that one node in the cluster might ask another node to stop serving a particular IP address, or shut down completely, or reboot, or perform various actions with serious consequences. In some respects, allowing a node to masquerade as a cluster member is tantamount to letting it be root on all of the cluster members. A cluster communications channel then becomes a way to crack a machine, and effectively become root. *Heartbeat*'s design intent includes the idea of avoiding adding another route to crack the systems in a cluster. To address this issue *heartbeat* digitally signs every packet, and implement a few precautions in the protocol code to deter replay attacks.

It comes supplied with the following default signature algorithms:

- 32−bit CRC (for physically secure networks only)
- MD5
- HMAC−SHA1 (believed to be the most secure of the three)

As described previously, an HA cluster is a continuously−running multicast system. Unlike many multicast systems, it is largely symmetric, with every node sending approximately the same number of messages, and most of them going to the entire set of members. One of its unique features is that the communication layer is intended to run indefinitely, hopefully for many years without interruption. This includes the need to be able to do various kinds of maintenance operations without ever taking down the entire cluster. These things should include:

- Changing keys
- Changing authentication methods
- Adding new authentication methods

These properties can be problematic, and to the author's knowledge are not all solved by any standard multicast protocol. The next section elaborates on *heartbeat*'s solution to these problems, on the details of the authentication protocol, and how, together with the low level protocol, various kinds of attacks are made more difficult. It is worth noting that one of the reasons for publishing the details of this protocol is that it is hoped that feedback will be received which will help further improve the security and authentication in *heartbeat*.

The Problem of Changing Keys

As noted above, it is necessary to be able to change keys, key types, and even add new authentication methods without taking the entire cluster down. In order to implement this, a key file called *authkeys* is used which contains the following information:

- signature method and key to sign outgoing packets with
- signature method**s** and key**s** which will be accepted on incoming packets

Note that each packet is signed with a single signature type, but each node will authenticate incoming packets which are signed with any of set of signature types. This allows one to change keys gracefully in a system by following the steps below:

1. Initial state. Every node signs with the "old" key, and only accepts the "old" key.
2. Distribute a new authkeys file to each machine which signs with the "old" key, but accepts the old and new keys
3. Activate the new authkeys on each machine, and wait for it to be activated on each machine.
4. Distribute a new authkeys file to each machine which signs with the "new" key, but accepts the old and new keys
5. Activate the new authkeys file on each machine, and wait for things to "settle out".
6. Distribute a new authkeys file to each machine which signs with the new key and accepts only the new key.
7. Activate the new authkeys file.

There is only one detail not well explained by the above description. What is the settling interval, and why is it there? The settling interval occurs because packets signed with the old key may still be present in the communications system, and unless sufficient time is allotted, they may be (re)transmitted and not be accepted by the other members of the cluster. How long this interval should be depends on the details of how the reliable multicast protocol works. In the current

implementation, 100 seconds is sufficient for this step in the worst possible case. This settling interval could be eliminated by resigning all packets saved for possible retransmission with the new key.

Adding new authentication methods can be added to *heartbeat* without recompiling the binary, because the authentication methods are now dynamically linked into the code, and will be loaded when it needs to do so without restarting the program.

Authentication Information in Packets

Each packet is signed with a field called the "auth" field. This field contains two subvalues: a number representing the authentication method along, with the signature value (a string) The number representing the authentication method does not have a fixed mapping to a particular authentication method. Attackers who see the packet cannot readily tell which authentication method is being used. If more methods are added, this will add more difficulty to the attackers job. If an individual site wishes to add security−through−obscurity, one can always add some undocumented methods to make the job of brute force attack of the key space somewhat more difficult for the amateur cryptographer.(or script kiddee). As an additional benefit, the authentication methods also detect packets which have been corrupted by simple media errors, allowing the *heartbeat* protocol to reliably run across media like serial ports which do not verify data integrity themselves.

Replay Attacks

If an attacker can mount an active attack (sniffing and injecting packets) on the heartbeat subnet, then they can initiate a replay attack, in which properly authenticated packets which were previously sent are resent, with potentially serious effects. In a replay attack, the attacker sniffs packets from the heartbeat subnet, then resends them at an opportune time later, so that those packets are taken as genuine. This can cause the cluster to give up resources, or shut down, or take any number of actions which might have been appropriate at some point in the past, but which should not be carried out now under the control of an attacker.

Initially, after putting in the authentication code, the possibility of replay attacks was ignored for the following reasons:

- Very few messages have the possibility of having any impact if replayed. (they occur only rarely)
- Sniffing the local subnet was thought to be difficult in practice for a cluster subnet.

However, others made convincing arguments why this protection was too weak:

- It is possible through various attacks to cause one of the cluster member to crash using one of the known denial of service attacks, creating a situation where messages with serious consequences are generated on demand.
- It is desirable to add multicast heartbeats to *heartbeat*, in which case there are potentially many more places to sniff the traffic than there are in the broadcast case.

These arguments were found to be convincing. Som ething needed to be done. In the original code, resetting sequence numbers was simply assumed to be a link reset, so an attacker could capture packets around a link reset (which almost always involve serious actions) and replay them. As a result, a change was devised to the packet management code which is believed to make replay attacks impossible. Each time a sequence number reset is performed, the new version of *heartbeat* increments an instance number. The handling of packet sequence and instance numbers fall into the following categories:

- Packets with a higher sequence number than seen before and a current instance number: treated as received
- Packets known to be missing: treated as received
- Packets with new instance numbers: treated as a protocol restart.
- Recent, duplicate packets: ignored
- Packets with "old" sequence numbers or instance numbers: ignored and flagged as a possible replay attack

Heartbeat Message Format

It is the purpose of a high−availability system to be available without interruption for years, through hardware and software upgrades. Consequently, one of the tasks that a cluster system must undertake is on−the−fly upgrades. Towards this end, it is necessary that old versions of the software should be able to accept messages from new versions of the software, and ignore fields in the messages which they do not understand. This is an undertaking of moderate difficulty, requiring careful thought on the design of new messages to be sent in the cluster and how they will be interpreted by the old software.

In this design, a communications system like *heartbeat* can be of some help. Some message formats are designed to easily allow software to ignore fields they don’t understand, and yet still find the information

they need in the fields they do understand. Towards this end, messages in the heartbeat system are designed in a fashion similar to the environment strings which UNIX™[2] supplies to processes.

In this format, each message consists of a set of ASCII (name, value) pairs. New message formats most commonly add new (name, value) pairs. In some cases, it is desirable to effectively change the semantics of a given name. When this situation occurs, there are various techniques available to handle it. One of the most common ones is to have the new version of the software continue to supply the old name semantics through the old name, and redundantly supply a new name, and the new semantics associated with it through the new (name, value) pair.

Although there are various techniques to deal with this situation, this message format has the advantage of being very simple, easy to understand, and yet very flexible. As a bonus, it is a simple matter to provide the contents of a message to a shell script.

The message format *heartbeat* adopted is simple, and easy to understand, but has a few disadvantages. ASCII data is bulky, and having names in every message makes it more so. This extra bulk is primarily of concern for heartbeat messages, which constitute the majority of data in the cluster communication. This has been made a little less problematic by choosing short field names for the fields found in heartbeat messages. This trades message bulk off against a little obscurity for heartbeat messages. With strong authentication, current heartbeat messages are approximately 150 bytes in size. It is believed that this size of message is small enough to justify the design chosen.

Heartbeat API

The initial implementation of *heartbeat* had a very simple prototype API for communication with management layers of the cluster. In this API, each cluster message caused the invocation of a process to which the message was given. Simple rules were used for filtering out heartbeat messages in which no state changed (i.e., weren't associated with a node entering or leaving the cluster). This interface is extremely effective and easy to put together scripts to manage a 2-node cluster. This prototype communication mechanism proved adequate for some simple purposes, and allowed easy assembly of a very basic (some might say crude) cluster manager for two nodes.

However, a more sophisticated cluster management functions need more state information, and need to be continuously running in order to negotiate with other nodes. This is a critical need for clusters with greater than two nodes. The prototype API fell short in many respects. These include:

- Ability to support more than one client process
- Ability to monitor and query the current state of the cluster
- Ability to adequately isolate the client process from the implementation details
- Ability to communicate with a continuously running process.
- Ability to write an independent communications debugger
- fork/exec is unreliable under heavy load

Since the prototype API was designed as a vehicle for demonstrating *heartbeat*, none of these limitations came as any surprise. At the time of this writing, this new heartbeat API has been implemented and is now in the testing phase. This API provides the following basic services:

- Obtain node and link status
- Observe changes in node and link status
- Send reliable messages to other nodes
- Receive reliable messages from other nodes
- Keep multiple users of the API from interfering with each other.

It is expected that this API will allow *heartbeat* to become useful in a wide variety of circumstances, ranging from normal HA operations, through integrating with other cluster managers, through integrating with cluster file systems.

STONITH Implementation

On the linux-ha mailing list, the term STONITH has been used to describe a technique for I/O fencing which allows one to guarantee exclusive use of a set of shared resources. STONITH stands for **S**hoot **T**he **O**ther **N**ode **I**n **T**he **H**ead. This technique allows cluster systems to safely use shared disk arrangements. An API has been defined for this technique, and an implementation was created. Although it is useful now, it will be more useful in the future, with a new cluster management infrastructure. At this point, the API and implementation will appear in *heartbeat* and several other open source cluster managers.

Future Directions

There are many possibilities on the horizon for *heartbeat*. These possibilities have been considerably broadened by the recently introduced API. These include.

True multicast

Currently, *heartbeat* does all of its UDP communi-

2 UNIX is a trademark of SCO.

cation using broadcast packets. This limits the cluster to being all on one subnet, and causes unnecessary interrupts on machines which are on the subnet but not part of the cluster. It would be good to implement multicast communications in order to alleviate these concerns.

Automated key distribution and management

At the present time, it is necessary for all the members of the cluster to have their authentication data updated manually. It would be desirable to have a way to distribute keys across the cluster, perhaps using openssh or some similar mechanism.

Integration with LinuxFailSafe

SGI and SuSE have made SGI's FailSafe product available on Linux as an open source software package **[Vas00]**. Although LinuxFailSafe currently has heartbeat and membership mechanisms of its own, there are enough things that *heartbeat* does better, that it is expected to prove desirable to replace portions of FailSafe with *heartbeat*.

GUI tools

There are at least two efforts underway to add a GUI configuration and status front end to *heartbeat*. Conectiva has written a linuxconf module for *heartbeat*, and David Martinez has prototyped a GUI configuration tool for it [**Mar00**].

Cluster Consensus Membership

Heartbeat provides only a local view of cluster membership. That is, it only knows about or is aware of each node's own individual view of the cluster membership. This is adequate for some purposes, but inadequate for many. If the local medium has no communication asymmetries, and the same configuration, then there is no difference between the local and the global view.

However, if communication is impeded due to hub or switch bugs or routed multicasts, then it is possible for different members of the cluster to have differing views of what the membership of the cluster actually is. It is important that every member of the cluster have the same view of cluster membership as every other member. It is necessary to have a consensus membership layer which then shares cluster membership information across the cluster, so that each machine has the same view of the cluster membership.

Barrier API implementation

Certain kinds of clusters use the model of barriers in their implementation. The TweedieCluster is an example of such a cluster **[Twe00]**. According to Tweedie, a barrier provides a guaranteed synchronization point in distributed processing which is valid over all nodes in the cluster: it strictly divides time into a pre−barrier and a post−barrier phase. The barrier may not necessarily complete at exactly the same time on every node, but there is absolute guarantee that the barrier will not complete on any node unless all other nodes have begun the barrier. The barrier API he describes could easily be implemented on top of the heartbeat API.

Phoenix n−phase transactions

IBM's Phoenix Cluster technology has the concept of providing a tool kit to implement generalized n−phase transactions to achieve synchronization across the cluster **[Pfi98]**. These are similar in function to the barrier API described above, but are more general. In this model, a transaction is begun in which a machine names a next state. Each node participating in the transaction then sends a message nominating the next state. When all nodes have reported the next state, then the transaction proceeds to the next phase. If the membership changes or any node nominates a differing next state during this time, the transaction is aborted. If this does not happen, the transition to the next state is accomplished, and the process is repeated until a final state is achieved. The idea of a generalized synchronization mechanism seems like a very good idea, and one worth emulating in the Linux−HA environment. This mechanism could be implemented on top of a barrier API, or using the heartbeat API directly.

Dynamic loading of modules

Much of the code in *heartbeat* consists of modules which are invoked through table lookups. These modules provide services which it would be desirable to add dynamic loading so that new modules could be added to *heartbeat* without restarting the services. In order to ensure reliable heartbeat services in the presence of ill−behaved applications, *heartbeat* runs locked into memory, and at high priority. Because of this, it is desirable to minimize the amount of memory which *heartbeat* used. Replacing the static linking which *heartbeat* uses with dynamic linking would help in this effort.

It is anticipated that *heartbeat* would benefit from dynamically loading at least the authentication methods, heartbeat media drivers, and STONITH implementations. At this point in time, the implementation of dynamic loading is being tested.

Ping membership

For two−node systems, it is often desirable to have a resource which can be used as a quorum resource, so that one can guarantee that only one of two partitions of a cluster can be active at a time. Since Linux−HA systems are intended to achieve very low deployment costs, it is undesirable to add a third node solely for the purpose of breaking ties. As a result, it would be desirable to allow relatively unintelligent devices which are already present on the customer's site to act as pseudo−cluster members. For these devices to par−

ticipate as members, a new class of membership would be added to *heartbeat*: ping membership. Whenever the local system would send out its heartbeat, a ping packet would be sent to the pseudo−member. Whenever a ping response is received, it would be translated into a heartbeat message. The result of this is that such devices could effectively act as tie−breakers in the case of 2−node quorum systems. Candidates for such devices include switches, routers, and ethernet−accessible intelligent power controllers or other pingable devices. There are problems potentially associated with this approach: It is possible (through ARP cache problems for example) for each of two nodes to be able to communicate with a common endpoint, yet be unable to communicate with each other. At this time, an implementation of this technique has been prototyped and further research will be done to see if there is a variant on this technique which would be helpful in solving these difficulties.

External Cluster Manager

Heartbeat's current cluster manager was originally prototyped to demonstrate *heartbeat*. It is unsophisticated and very limited in its function. However, the introduction of the heartbeat API opens up the possibility of providing a "real" cluster manager, or a series of them. It is expected that *heartbeat* will drop its primitive cluster management facility in favor of external cluster management. As a result, this current implementation is not discussed here.

Design Retrospective

The previous sections describe the positive reasons why the design decisions were made without significant references being made to the corresponding costs or disadvantages. This section will explore these costs, and comment on the design of these key areas. Some of the areas addressed here were pointed out by the readers of the linux−ha−dev mailing list.

Communication using PPP

Initially, the *heartbeat* code was implemented directly on top of the serial device. Later on it was suggested by Stephen Tweedie and Alan Cox that PPP might be a better choice for using the serial ports. This seemed worth trying, so it was implemented. Unfortunately, there were several problems which resulted. These problems include: PPP unreliability, slow startups, code complexity, and PPP hangs. In a small percentage of cases, PPP would not start at all, and had to be killed and restarted, sometimes on both ends. Even when it does start correctly, PPP can take up to seven seconds to start. This makes hangs slow to detect, and delays starting up the cluster. Sometimes, for no apparent reason, PPP would stop transmitting in one direction for no apparent reason and with no obvious symptoms. These bugs and behaviors in PPP make it a poor choice for use in highly reliable systems. These problems exist today, and have existed for more than a year in the Linux PPP implementation. The workarounds for all these problems make the PPP transport module in *heartbeat* about twice as large and complex as the corresponding code for any other medium, in addition to being unreliable.

Code Size

Although the *heartbeat* code has proven quite robust, various people have asked questions about its size. On i386 Linux, the object size is approximately 128K including all *heartbeat* media, authentication methods, STONITH methods, the API, and the code for the prototype cluster management function. This breaks down to lines of commented source like this: 4000 lines for core function, 800 lines for messaging functions, 600 for authentication, 1000 for parsing configuration files, 2000 lines for handling different transport media (over half of which is PPP code), 1400 for the heartbeat API. At this point, it seems to be eminently maintainable. The recently−added ability to dynamically load modules will minimize memory usage while allowing for more growth in capabilities.

Flexibility

It has proven to be quite simple to add new *heartbeat* media types, authentication types, message types and features to *heartbeat*. Much of the code is table−driven, and is overall quite straightforward and simple to extend and change. Indeed, one of the unexpected problems associated with writing this paper has been that many things have changed and been implemented while it was being written, causing it to have to be updated continually.

Bandwidth usage

One of the concerns which been expressed concerning the design of *heartbeat* is that its combination of ASCII message format makes for verbose heartbeat messages. The concern is specifically that it might become a bandwidth hog as cluster sizes grow.

The formula for computing bandwidth used by *heartbeat* for N nodes on an unswitched network is as follows:

$$B_{hb} = S_{hb} * 8_{\text{bits/byte}} R_{hb} * N$$

Where B_{hb} is the bandwidth consumed in bits/sec, S_{hb} *is the size of a heartbeat's packet in bytes, R_{hb} is the rate of heartbeat per cluster node, and N is the number of nodes in a cluster. A common heartbeat rate is 1 packet/sec, heartbeat packets average around 150 bytes. If a cluster has 1000 nodes with these other characteristics, then the B_{hb} for such a system is 1.2 x 10^6 bits/sec. This is approximately 1.2% of the bandwidth available on an unswitched 100 Mbit network. It is clear from this calculation that fears about *heart*−

beat bandwidth consumption are unfounded for realistic−sized clusters. On the other hand, it is also clear from this calculation that this same bandwidth would significantly overwhelm a normal PC serial connection.

Reliability

Although released versions of *heartbeat* have had a few bugs, they have largely exhibited themselves consistently on system startup or not all. The result of this is that *heartbeat* has proven itself to be quite reliable in actual field usage.

Protocol Limitations

The current heartbeat protocol guarantees that packets are delivered to every active node, or that the client is notified of the death of the nodes to which it isn't delivered subject to resource limitations. However, these packets are not guaranteed to be delivered in any particular order, nor is any flow control provided. The packet delivery order problem is relatively easily solvable should that prove desirable. The flow control issue is not expected to be a problem in the domain for which this protocol is designed (high−availability control messages). However, if a client were sufficiently ill−behaved it is possible that it could exhaust packet retransmission facilities, resulting in a packet not being delivered. In practical tests with well−behaved clients and extraordinary packet loss rates (90% on transmission, and 90% on reception), this behavior could not be induced in many hours of testing.

Security

Because it was designed from the ground up to operate in the cracker−rich internet environment as an open source Linux package, it is one of the more security−conscious high−availability implementations available. Although it makes no attempt to encrypt data because of repressive US laws on encryption, it does make good use of strong authentication protocols.

Perhaps too successful

One of the most interesting and curious criticisms of *heartbeat* is that it has been too successful. What was meant by this was that *heartbeat* solved a wide−enough class of problems sufficiently well that there was little motivation to create an alternative solution, even though *heartbeat* itself was quite limited. In particular, the original heartbeat API was incapable of supporting a sophisticated cluster management infrastructure, so that until the new API was added, it was impossible to go beyond two nodes in the cluster. Now that this API is available, it is possible to write a better cluster management services. Now, *heartbeat*'s past success can now serve as a springboard for future enhancements and greater usefulness rather than serving as a source of criticism.

Is *heartbeat* misnamed?

Because of the fact that it treats providing heartbeat services as a subcase of cluster communications, some have said that *heartbeat* is misnamed. It is not simply a heartbeat mechanism, but is best thought of as a robust low−level cluster communications mechanism which provides notification when nodes join and leave the communication channel. In this sense, the heartbeat could perhaps even be thought of as a side−effect of the reliable communications protocol.

Conclusion

Heartbeat's technique of providing heartbeats as an integral part of a cluster communications channel has proven straightforward and to implement and maintain. its use of ASCII message formats has proven quite flexible, and added to the ease of debugging. The addition of an API is pushing it into a new stage of growth and usefulness in roles where it provides only cluster communication and membership services. This is expected to make it more useful than ever, and allow many different applications and cluster management functions to interface with it.

Acknowledgments

Thanks to the members of the Linux−HA and linux−ha−dev mailing lists who have used and critiqued and questioned aspects of *heartbeat*. In this regard, the author offers special thanks to Jerome Etienne. In many ways, Jerome inspired this paper by persistently asking questions about *heartbeat*, its design decisions and questioning their wisdom. Without Jerome's persistence and energy, this paper would likely not have been written. Additionally, Jerome commented on extensively on several drafts of this document. The following people also provided useful comments on drafts of this document which improved it considerably: Peter Badovinatz, Simon ("horms") Horman, Neal McBurnett, John Schmaus and Stephen Tweedie. The author would also like to thank Neal McBurnett of AVAYA (formerly Bell Labs) for his help in designing the authentication scheme used by *heartbeat*. Additionally, many, many people have contributed to *heartbeat*, by writing code for it, or testing it, or providing documentation for it. Without them, *heartbeat* itself would not be nearly the program it is now.

To Learn More

The Linux−HA web site can be found at **[Rob00]**. *Heartbeat* can be downloaded (in source or RPM format) from the Linux−HA web site download page at: http://linux−ha.org/download/. Information on subscribing to the various Linux−HA mailing lists can be found on the contact page at: http://linux−ha.org/contact/. The Linux FailSafe project is described in detail in **[Vas00]**.

References

[Dan94] Danzig, P. B.: "Flow Control for Limited Buffer Multicast", IEEE Transactions on Software Engineering, Vol 20, No. 1, January 1994, pp. 1−12

[Milz99] Milz, Harald: "The Linux High Availability HOWTO". http://metalab.unc.edu/pub/linux/ALPHA/linux−ha/High−Availability−HOWTO.html

[Mar00] Martinez, D.: http://linux−ha.org/screenshots.html

[Phi98] *In Search of Clusters*, by Gregory F. Pfister, 2nd Edition 1998, Prentice Hall PTR

.[Ram87] Ramakrishnan, S., Jain, B. N.: "A Negative Acknowledgment with Periodic Polling Protocol for Multicast over LANs". In: Proc. IEEE INFOCOM ’87, March 1987, S. 502−511.

[Rob00] Robertson, A. L.,: "The High−Availability Linux Project". Http://linux−ha.org/

[Twe00] Tweedie, S. C.,: "Barrier Operations". http://linux−ha.org/PhaseII/WhitePapers/sct/barrier.txt

[Vas00] Vasa, M.,: "The Linux Fail Safe Project". http://oss.sgi.com/projects/failsafe/

[Wei00] Weis, R., Geyer, W, Kuhmünch, C., "Architectures for Secure Multicast Communication", In: Proc. SANE 2000 System Administration and Networking Conference, May 22−25, 2000., pp. 63−91.