Active-Active Servers and Connection Synchronisation for LVS

Horms (Simon Horman) – horms@valinux.co.jp
VA Linux Systems Japan, K.K. – www.valinux.co.jp
with assistance from
NTT Comware Corporation – www.nttcom.co.jp

January 2004

http://www.ultramonkey.org/

Abstract

The Linux Virtual Server Project (LVS) allows the Linux Kernel to load balance TCP and UDP services. The host running LVS is referred to the Linux Director. All connections for load balanced services are forwarded by the Linux Director.

This system can be made highly available by having a standby linux director that can take over when a failure occurs or system maintenance is necessary. However, when failover occurs existing connections are broken as the standby does not know about them. Connection Synchronisation attempts to resolve this problem by sharing session information between active and stand-by directors.

If linux directors are stable and are taken down for system maintenance infrequently, then the standby is idle for much of the time. This is arguably a waste of resources. Also, the throughput of the cluster is limited by that of the linux director. By having multiple linux directors in an active-active configuration these problems can be alleviated.

This paper presents an implementation of active-active linux directors as well as significant enhancements to the connection synchronisation capabilities of LVS. These imeplementations should be considered as work in progress. It is intended for a technical audience that is interested in building large internet sites.

Introduction

Load Balancing using Layer 4 Switching, as provided by The Linux Virtual Server Project (LVS)[3] and others is a powerful tool that allows networked services to scale beyond a single machine[5]. This is the prevailing technology for building large web sites from small, commodity servers.

Layer 4 Switching can, however, introduce a single point of failure into a system - all traffic must go through the linux director when it enters the network, and often when it leaves the network. Thus if the linux director fails then the load balanced site becomes unavailable. To alleviate this problem it is common to have two linux directors in an active/stand-by configuration.

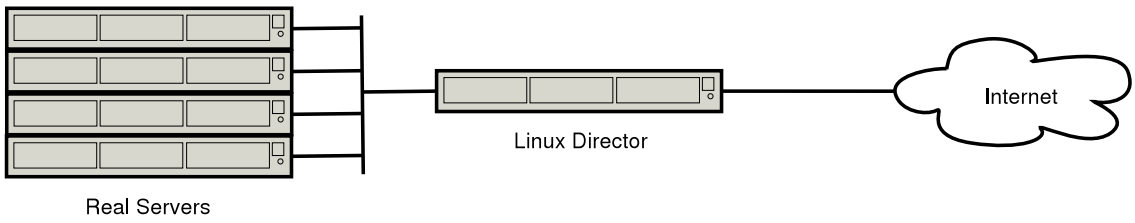


Figure 1: One Director, Single Point of Failure

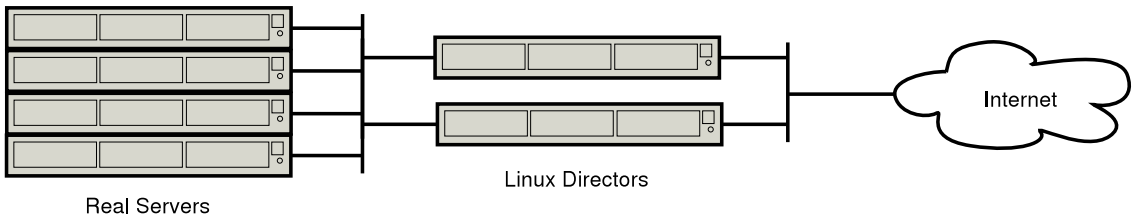


Figure 2: Active/Stand-By Directors

In this scenario the active linux director accepts packets for the service and forwards them to the real servers. The stand-by linux director is idle. If the active linux director fails or is taken off-line for maintenance, then the stand-by becomes the active linux director. However, when such a fail-over occurs, any existing connections will be broken. *Connection Synchronisation* resolves this problem by synchronising information about active connections to the stand-by linux director. Thus, when a fail-over occurs, the new active linux director is able to load balance subsequent packets for these connections.

Part 1 of this paper will discusses how LVS's connection sychronisation implemntation works and improvements that have been made to it which allow connections to continue regardless of which linux director is active or how many times a fail-over occurs.

Another problem with active/stand-by configurations is that while the real servers can be horizontally scaled by adding more real servers. There is no easy way to increase the capacity of the linux director beyond its hardware capabilities. *Active-Active* provides horizontal scalability for the available linux directors by having all of the available linux directors active load balance connections. This allows the aggregate resources of the linux directors to be used to. Thus, using otherwise idle resources to provide additional capacity.

Part 2 of this paper discusses an method to allow active-active linux directors for between one and approximately sixteen hosts to act as the linux director for a service simultaneously. The design is such that adding additional linux directors should give a near linear increase in the net load balancing

capacity of the network. Given that a single linux director running LVS on the Linux 2.4 kernel can direct in excess of 700Mbits/s of traffic, the cost of obtaining the resources to test the combined load balancing capacity of a number of active-active linux directors is somewhat prohibitive. For this reason the implementation and capacity testing of this project will focus on one to three linux directors.

The design also allows load balancing to continue when linux directors fail, as long as at least one or more of the linux directors is still functional. While the discussion of active-active focuses on its use for linux directors it should be applicable to any type of host.

Part I

Connection Syncronisation

1 LVS Connection Synchronisation Overview

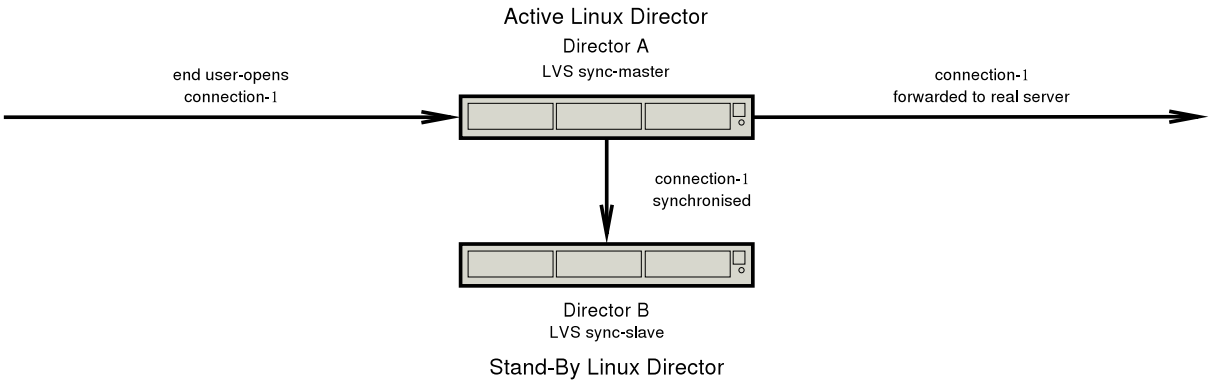
When a linux director receives a packet for a new connection it allocates the connection to a real server. This allocation is effected by allocating an ip_vs_conn structure in the kernel which stores the source address and port, the address and port of the virtual service, and the real server address and port of the connection. Each time a subsequent packet for this connection is received, this structure is looked up and the packet is forwarded accordingly. This structure is also used to reverse the translation process which occurs when NAT (Network Address Translation) is used to forward a packet and to store persistence information. Persistence is a feature of LVS whereby subsequent connections from the same end-user are forwarded to the same real server.

When fail-over occurs, the new active linux director does not have the ip_vs_conn structures for the active connections. So when a packet is received for one of these connections, the linux director does not know which real server to send it to. Thus, the connection breaks and the end-user needs to reconnect. By synchronising the ip_vs_conn structures between the linux directors this situation can be avoided, and connections can continue after a linux director fails-over.

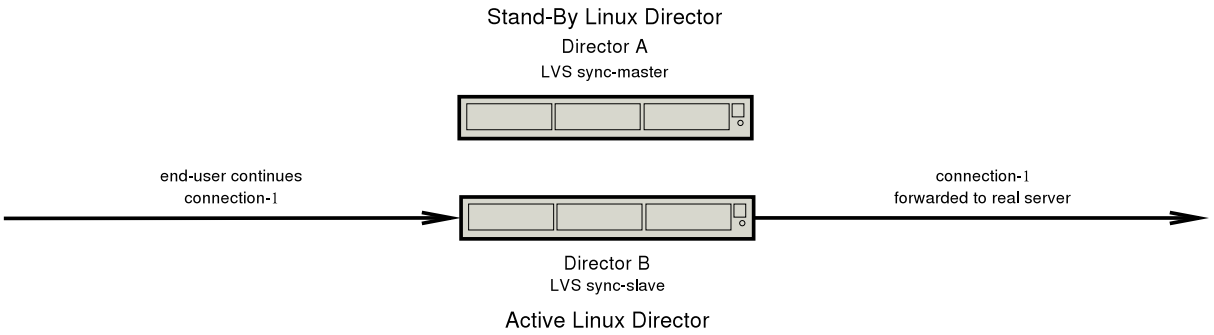
2 Master/Slave Problem

The existing LVS connection code relies on a sync-master/sync-slave setup where the sync-master sends synchronisation information and the sync-slaves listen. This means that if a sync-slave linux director becomes the active linux director, then connections made will not be synchronised. This is illustrated by the following scenario.

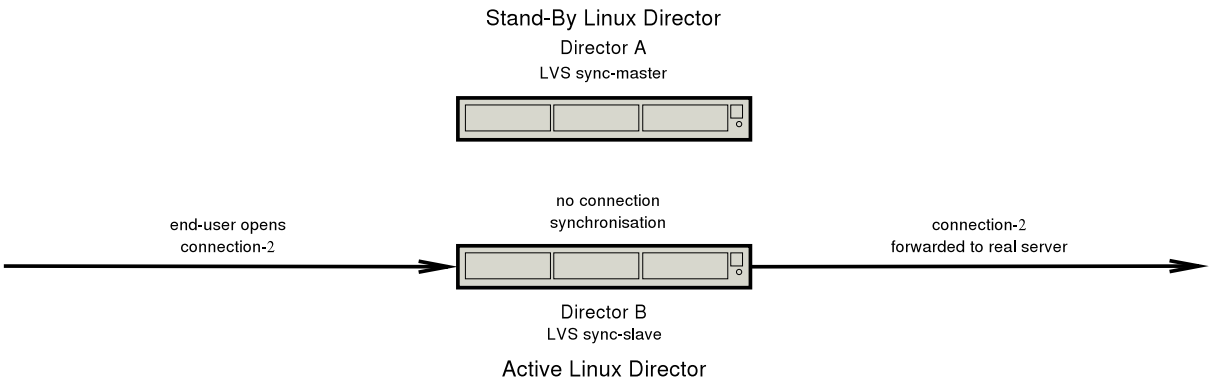
There are two linux directors, director-a and director-b. Director-A is the LVS sync-master and the active linux director. Director-B is an LVS sync-slave and the stand-by linux director.



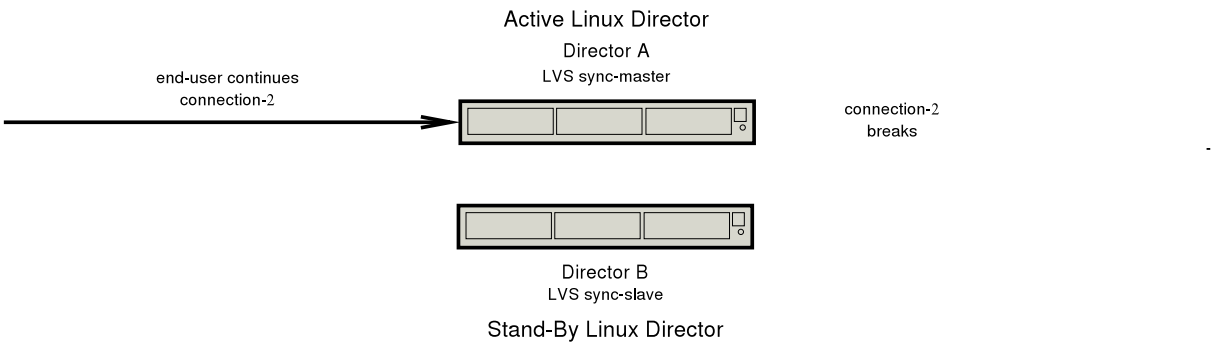
Step 1. An end user opens connection-1. Director-A receives this connection, forwards it to a real server and synchronises it to the sync-slave, director-b.



Step 2. A fail-over occurs and director-b becomes the active linux director. Connection-1 is able to continue because of the connection synchronisation that took place in step 1.



Step 3. An end user opens connection-2. Director-B receives this connection, and forwards it to a real server. Connection synchronisation does not take place because director-b is a sync-slave.



4. Another fail-over takes place and director-a is once again the active linux director. Connection-2 is unable to continue because it was not synchronised.

Patches are available for LVS which allow a linux director to run as a *sync-master* and *sync-slave* simultaneously. This overcomes the above problem. The patches have been integrated into the LVS code for the Linux 2.6 kernel since LVS-1.1.6. They have not been integrated into the LVS code for the Linux 2.4 kernel as of LVS-1.0.10 but the patches can be found on the lvs-users' mailing list[3].

As different approach to this problem, a peer to peer synchronisation system has been developed. In the system developed each linux director sends synchronisation information for connections that it is handling to all other linux directors. Thus, in the scenario above connections would be synchronised from director-b to director-a and connection-2 would be able to continue after the second fail-over when director-a becomes the active director again.

3 Protocol Deficiencies

LVS's existing connection synchronisation code uses a very simple packet structure and protocol to transmit synchronisation information. The packet starts with a 4 byte header followed by up to 50 connection entries. The connection entries may be 24 or 48 bytes long. The author has observed that in practice, most synchronisation connection entries are 24 bytes, not 48 bytes. Thus, the 50 connection entry limit results in most packets having 1204 bytes of data.

There are several aspects of this design that can be improved without impacting on its simplicity: There is no checksum in the packet. This makes it impossible to detect corruption of data. There is no version field in the packet. This may make it difficult to make subsequent changes to the packet structure. On networks with an MTU significantly greater than 1204 bytes larger packets may be desirable.

4 Implementation

The implementation moves much of the the synchronisation daemon logic out of the kernel and into user-space as it is believed that more sophisticated synchronisation daemons can be developed more rapidly in user-space than in kernel space. The main disadvantage of moving the synchronisation logic into user-space is that there is a possible performance impact. In particular, there is some overhead in passing the synchronisation information to user space, especially as the user-space daemon's multicast communication must pass back through the kernel. However, testing showed that the netlink socket is very fast, see appendix Appendix A.

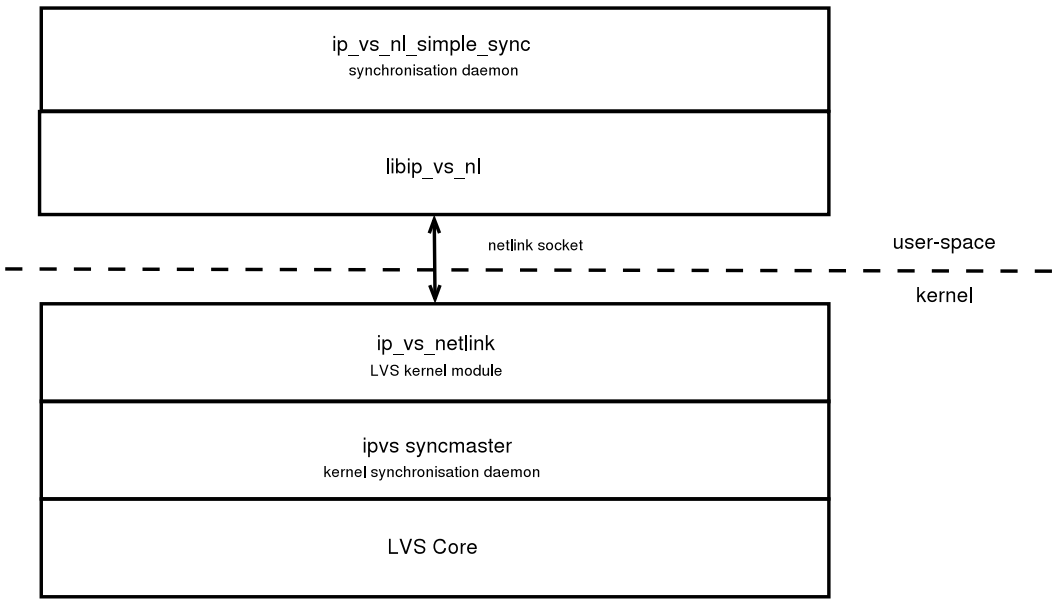


Figure 3: Block Diagram

LVS Core

LVS's existing synchronisation code works by feeding synchronisation information to *ipvs syncmaster* which in turn sends it out onto the network. *ipvs syncslave* listens on for synchronisation packets to arrive via multicast and feeds them into the LVS core. *ipvs syncmaster* and *ipvs syncslave* should be started on the linux directors using `ipvsadm`.

By default a connection is passed to *ipvs syncmaster* once it passes a threshold of 3 packets, and then is resent to *ipvs syncmaster* at a frequency of once every 50 packets after that. The threshold may be manipulated using the existing `net/vs/sync_threshold` proc entry. To allow the frequency to be manipulated, LVS has been patched to add the `net/vs/sync_frequency` proc entry. As synchronisation information for a given connection is resent every 50 packets it is not considered of particular importance if occasionally the synchronisation packets are dropped. This is important, as multicast UDP which does not have an underlying retransmission method for dropped packets, is used for inter-linux director communication.

The LVS synchronisation code was patched to abstract the sending and receiving of synchronisation packets. This allows different methods of sending and receiving synchronisation packets to be used without further modification to this code. The following hooks are provided:

send_mesg	Used by *ipvs syncmaster* to send a packet.
open_send	Used by *ipvs syncmaster* to open the socket used to send packets.
close_send	Used by *ipvs syncmaster* to close the socket used to send packets.
recv_loop	Event loop used by *ipvs syncslave* to receive packets.
open_recv	Used by *ipvs syncslave* to open the socket that is used to receive packets.
close_recv	Used by *ipvs syncslave* to close the socket that is used to receive packets.

The following functions are provided to register functions for these hooks. It is envisaged that the hook functions would be implemented in a separate kernel module. When the module initialises itself *ip_vs_sync_table_register* should be called. When the module is unregistering itself, *ip_vs_sync_table_register_default* should be called.

ip_vs_sync_table_register	Used to register the hooks above. If NULL is supplied for any of the hooks, then the hook will have no effect.
ip_vs_sync_table_register_default	Used to register the default hooks, which gives the existing behaviour of of directly sending and receiving multicast packets. This behaviour is registered when LVS is initialised.

The proc entry `net/vs/sync_msg_max_size` was added to allow the maximum size of messages sent by *ipvs syncmaster* to be modified. In situations where the linux director is under load, this will be the size of most synchronisation packets. The default is 1228 bytes which reflects the old hard-coded value. The intention of this is to allow more connections to be synchronised in a single packets on networks with an MTU significantly larger than 1228 bytes, such as ethernet networks with jumbo 6000 byte frames and netlink sockets which have an MTU of 64Kbytes.

ip_vs_user_space

Using the hooks that were added to the LVS synchronisation code, a method that sends and receives synchronisation packets via a netlink socket was written. This was implemented as a separate kernel module, *ip_vs_user_space*. When this module is inserted into the kernel it registers itself as the synchronisation method and starts the kernel synchronisation daemon *ipvs syncmaster*. The *send_mesg* hook registered passes synchronisation packets to user-space. The *ip_vs_user_space* module listens for synchronisation information from user-space and passes it directly to the LVS core. Thus there is no need for an *ipvs syncslave* process and *ipvs syncmaster* can be run on all linux directors. This is important to allow a peer-to-peer relationship to be established between linux directors in place of a master/slave relationship.

The *ipvs syncmaster* kernel daemon is started when *ip_vs_user_space* is initialised. Thus, unlike the existing LVS synchronisation code, this daemon should not be started using `ipvsadm`.

Netlink Socket

The user-space daemon communicates with to its kernel counterpart using an netlink socket. This requires a small patch to the kernel to add the NETLINK_IPVS protocol. Communication using this socket is somewhat analogous to UDP: Packets of up to 64Kbytes may be sent. Packets may be dropped in situations of high load, though unlike UDP this results in an error condition. Unlike UDP the data in packets received can be assumed to be uncorrupted. Unless there is broken memory in the machine, which should result in other, more dire problems. Importantly only local processes owned by root may communicate with the kernel using the netlink socket. Thus there is some level of implicit authenticity of the user-space client.

Testing of the Netlink Socket as shown in appendix Appendix A shows that there is a sweet spot for throughput at a packet size of around 7200bytes. For this reason it is suggested that the maximum size of synchronisation packet sent by ipvs syncmaster be set to this value using the `net/vs/sync_msg_max_size` proc entry.

libip_vs_nl

This is a library that provides a convenient way for user-space applications to communicate with the kernel using the NETLINK_IPVS netlink socket. This provides calls analogous to libc's send(), and recv() as well as defining a simple packet structure which is also used by *ip_vs_user_space*.

ip_vs_nl_simple_sync

A user-space synchronisation daemon. It uses *libip_vs_nl* to access a netlink socket to listen for synchronisation information from the kernel. It then sends this information to other linux directors using multicast. The packet format used to send these packets has a version field to allow packets from a different version of the protocol to easily be identified and dropped. All nodes running this daemon can send and receive this synchronisation information. Thus there is no sync-master/sync-slave relationship.

This is intended as a bare-bones synchronisation daemon that illustrates how *libip_vs_nl* can be used in conjunction with *ip_vs_user_space* to create a user-space synchronisation daemon. Its key advantage over the existing LVS synchronisation code is that eliminates the sync-master/sync-slave and the problems that can introduce. It is suitable for use in both active/stand-by with two linux directors and active-active configurations with any number of linux directors.

This daemon, like the existing LVS synchronisation code, has no security protection. Thus interfaces listening for synchronisation packets over multicast should be protected from packets from untrusted hosts. This can be done by filtering at the gateway to the network, or using a private network for synchronisation traffic. It is not sufficient to use packet filtering on an exposed interface, as it is trivial for a would-be attacker to spoof the source address of a packet.

5 Sample Configuration

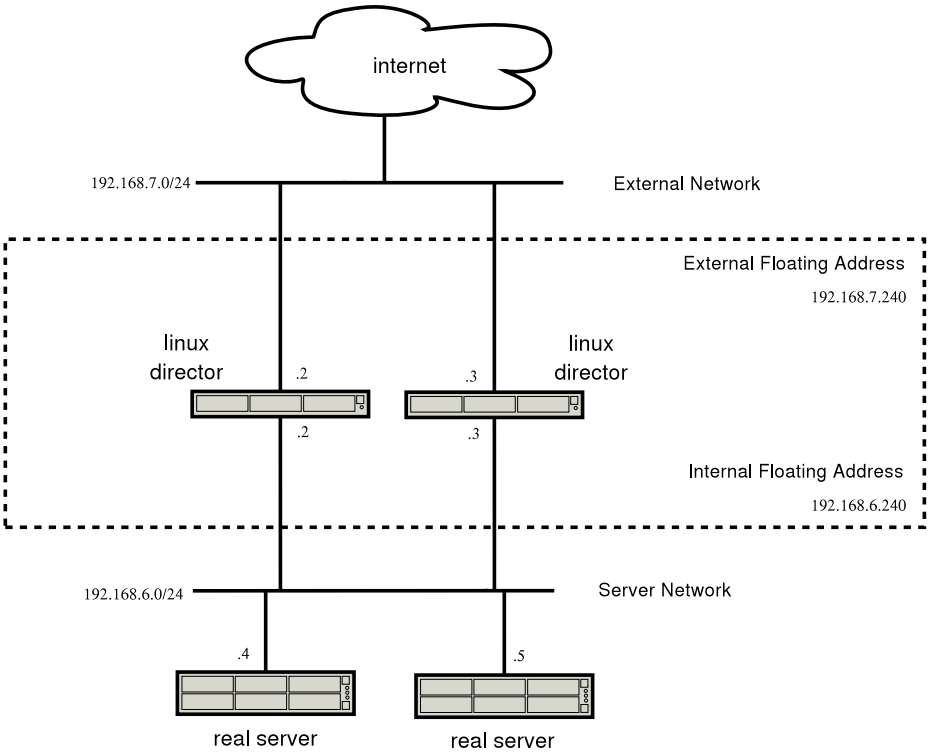


Figure 4: Sample Topology

The topology has two linux directors, in an active/stand-by configuration. There is a Virtual IP Address (VIP) on the external network which is the IP address that end-users connect to and should be advertised in DNS. There is also a VIP on the internal network. This is used as the default route

for the real servers. The VIPs are administered by Heartbeat[2] so that they belong to the active linux director. *ip_vs_nl_simple_sync* runs on both of the linux directors to synchronise connection information between them. NAT (Network Address Translation) is used to forward packets to the two real servers. More real servers may be added to the network if additional capacity is required.

Given the flexibility of LVS, there are many different ways of configuring load balancing using LVS. This is one of them. Details on configuring LVS can be found in the LVS HOWTO[8] and several sample topologies can be found in the Ultra Monkey documentation[4]. The information in these documents, combined with this paper should be sufficient to configure load balancing with connection synchronisation for a wide range of network topologies.

Network Preparation

The documentation that follows assumes that all nodes on the network are set up with correct interfaces and routes for each network they are connected to as per the diagram above. The return path for packets must be through the active linux director. In most cases this will mean that the the default route should be set to the VIP.

Linux Directors

Packet Forwarding

The linux directors must be able to route traffic from the external network to the server network and vice versa. Specifically, in addition to correctly configuring the interfaces and routes you must enable IPV4 forwarding. This is done by modifying the line containing net.ipv4.ip_forward in `/etc/sysctl.conf` as follows. Alternatively the corresponding `/proc` value may be manipulated directly.

```
net.ipv4.ip_forward = 1
```

For this change to take effect run: `sysctl -p`

Heartbeat

Heartbeat runs on the two linux directors and handles bringing up the interface for the VIPs. To configure heartbeat `/etc/ha.d/ha.cf`, `/etc/ha.d/haresources` and `/etc/ha.d/authkeys` must be installed. The node names in `/etc/ha.d/ha.cf` and `/etc/ha.d/haresources` must be set according to the output of the uname-n command on each linux director. The key in `/etc/ha.d/authkeys`, should be modified to something confidential to the site. It is highly recommended that heartbeat be run over at least two links to avoid a single link-failure resulting in a fail-over. More information on configuring these files can be found in the documentation an sample configuration files supplied with Heartbeat.

To start heartbeat run: `/etc/init.d/heartbeat start`

After a few moments heartbeat should bring up an IP alias for the VIP on the first master linux director. This can be verified using the ifconfig command. The output of the following command has been truncated to only show the eth0:0 and eth1:0 interfaces. Depending on the setup of the host it is possible that heartbeat will use difference interfaces.

```
/sbin/ifconfig
eth0:0    Link encap:Ethernet  HWaddr 00:D0:B7:BE:6B:CF
          inet addr:192.168.6.240  Bcast:192.168.6.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:17 Base address:0xef00

eth1:0    Link encap:Ethernet  HWaddr 00:90:27:74:84:ED
          inet addr:192.168.7.240  Bcast:192.168.7.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:18 Base address:0xee80
```

Ldirectord

The monitoring of real servers, and their insertion and removal from the pool of servers available is controlled by ldirectord. To configure ldirectord, `/etc/ha.d/ldirectord.cf` must be installed. Information on customising this file can be found in the `ldirectord`(8) man page and in the sample configuration supplied with ldirectord.

To start ldirectord run: `/etc/init.d/ldirectord start`

Ldirectord should initialise the the current LVS kernel table. To inspect this use `ipvsadm`. For example:

```
/sbin/ipvsadm -L -n
IP Virtual Server version 0.9.16 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port          Forward Weight ActiveConn InActConn
TCP  192.168.7.240:443 rr
  -> 192.168.6.4:443            Masq    1      0          0
  -> 192.168.6.5:443            Masq    1      0          0
```

Netfilter

As NAT is being used to forward packets to the real servers, netfilter on linux director should be configured accordingly:

```
# Flush existing rules
/sbin/iptables -F

# NAT for 192.168.6.0/24 bound for any host
/sbin/iptables -t nat -A POSTROUTING -j MASQUERADE -s 192.168.6.0/24

# Log all packets that attempt to be forwarded
# Useful for Debugging. Questionable for Production
#/sbin/iptables -t nat -A POSTROUTING -j LOG
```

If an FTP Virtual Service is to be used then the `ip_vs_ftp` kernel module needs to be used:

```
/sbin/modprobe ip_vs_ftp
```

Synchronisation Daemon

The synchronisation daemon ip_vs_nl_simple_sync is configured by editing `/etc/ip_vs_nl_sync/ip_vs_nl_simple_sync.conf`.

To start ip_vs_nl_simple_sync run: `/etc/init.d/ip_vs_nl_simple_sync`

Optionally, the the communication overhead between the kernel and user-space synchronisation daemons, can be slightly reduced by increasing the maximum packet size for packets sent by the kernel daemon from 1228bytes to 7200 bytes. To do this, add the following line to `/etc/sysctl.conf` or by manipulating the corresponding `/proc` entry directly.

```
net.ipv4.vs.sync_msg_max_size = 7200
```

For this change to take effect run: `sysctl -p`

Real Servers

The real servers should be configured to run the underlying services for their respective virtual services. For instance, an HTTP daemon. In addition the the "request" URLs as specified in `/etc/ha.d/ldirectord.cf` should be present and contain the "receive" string. The real servers also need to be set up so that their default route is set to the VIP on the server network.

Result

When an end-user make a connection to the external VIP, 192.168.7.240 this should be received by the active linux director. The linux director will then allocate the connection to one of the real servers and forward packets to the real server for the life of the connection. It will also synchronise the connection to the other director using the synchronisation daemon. If the active linux director fails then the stand-by linux director should assume the VIP, then as this director has information about all the active connections, from the connection synchronisation daemon, any active connections will be able to continue.

Part II

Active-Active

6 Overview

The proposed method of providing Active-Active is to have all linux directors configured with the same Ethernet Hardware (MAC) Address[6] and IP Address. While the focus of this project is to distribute traffic to multiple linux directors, it should work equally well for any type of host.

Overview: A Common MAC and IP Address

MAC addresses are 48bit integer values[7], usually represented as six colon delimited octets in hexadecimal. A MAC address is used by Ethernet hardware - usually NICs and Switches - to address each other. When a host wants to sent a packet to another host on the network it resolves the IP address of the host to its MAC address using The Address Resolution Protocol (ARP)[9]. The host then uses this MAC address to send the ethernet frame that encapsulates the packet to the host.

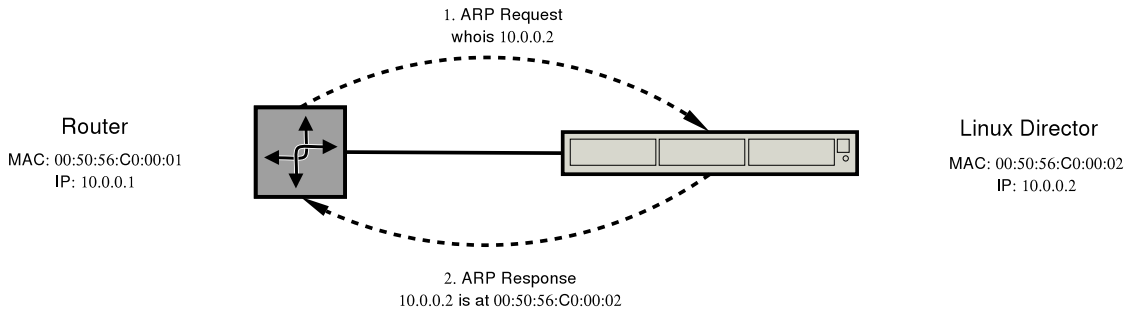


Figure 5: Resolving the MAC Address of a Node Using ARP

MAC addresses are preassigned to ethernet equipment by their manufacturer. Each manufacturer has a designated range of MAC addresses. Thus, each IP address in use on the network is present on a single host, and each MAC address is present on a single host. Thus when an ARP request is sent for the MAC address of an IP address that is present on the network, a single reply is received from the host that owns that IP address, with its MAC address.

Most ethernet hardware allows the MAC address to be changed, so it is possible to configure the NICs of multiple hosts to have the same MAC address. If multiple hosts have the same MAC and IP - this will be referred to as the common MAC and IP address respectively - then an ARP request for the common IP address will result in multiple ARP responses, all with the common MAC address. Although each ARP response will replace the previous one, they will all be the same so this is not important. Ethernet frames will be sent to the common MAC address. These frames should be received by all of the hosts on the network that have the common MAC address. Thus, the packets being sent will be received by all of the hosts with the common MAC address.

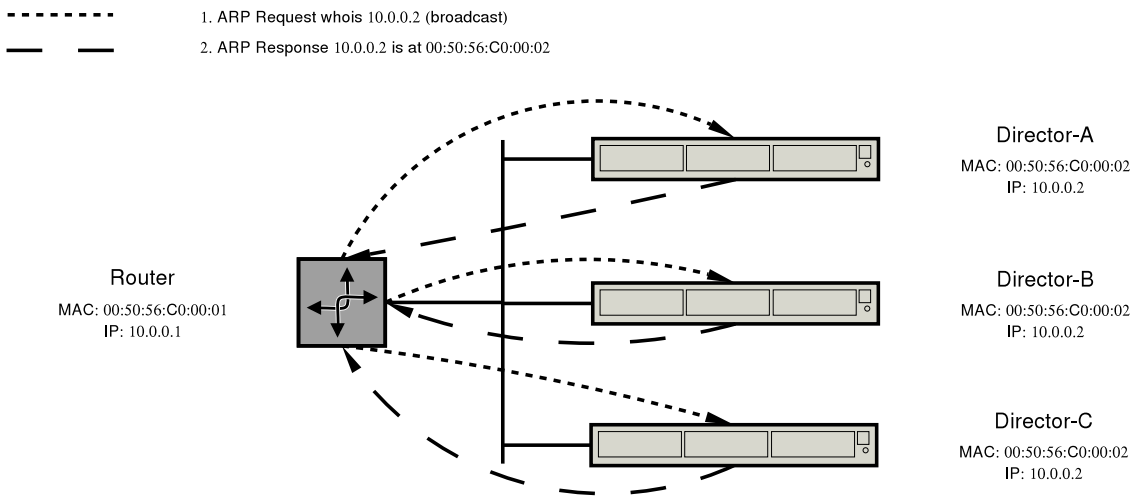


Figure 6: ARP on a Network with Hosts with a Common MAC and IP Address

Overview: Filtering

Although packets are received by every host with the common MAC address, it is undesirable for them to be processed by more than one host - one connection should be terminated by one host. Hubert[6] suggests using static iptables[11] rules to drop traffic based on the source address. For example, if there are two linux directors, Director-A and Director-B. Then Director-A should accept packets from hosts with an even IP address, and Director-B should accept packets from hosts with an odd IP address. This works well and can be scaled to any reasonable number of hosts by dividing up the source address space accordingly. However, this is not dynamic, and if one of the hosts fails then some end-users may not be able to connect.

Another problem with this approach is that it assumes that all source addresses are equal. However, it is quite reasonable to expect that some source addresses may make more requests than others. An example where this may occur are hosts that NAT or proxy large networks. Depending on who is accessing the service in question, this could result in an uneven distribution of load. Clearly, as Hubert himself suggests there is room for improvement.

7 Implementation

Implementation: Common MAC and IP Address

Giving NICs on different hosts the same MAC and IP address is a trivial task that can be done using the `ifconfig` or `ip` commands. Examples can be found in *Putting It Together: A Common MAC and IP Address* later in this paper.

Common MAC and IP Address: `outgoing_mac`

In an unswitched network this should work fine as the ethernet is a broadcast medium, so the NICs on every host on the network will see every frame sent. The NICs on the hosts with the common MAC address will dutifully accept packets addressed to them. However, in a switched environment things are a little bit more difficult.

The switches used during testing stored the MAC addresses being used to send packets by hosts connected to each port. Subsequent packets sent to one of these MAC addresses are only sent to the associated port. If each host has a unique MAC addresses - as they typically do - then this works quite well. But in a situation where a common MAC address is used by several hosts the result is that only one host will receive packets sent to the common MAC address. Clearly this is a problem in implementing the design presented here.

A simple solution to this problem is to use a bogus MAC address as the source when sending packets. This prevents the switch from associating the common MAC address with any particular port. And when the switches used in testing didn't know which port a MAC was associated with it would send the frame to all ports. Thus all of the hosts with the common MAC address receive the frame. This has the slight disadvantage that the switch behaves much like a hub for packets addressed to the common MAC address. But that is largely unavoidable.

Implementing this solution turned out to be a straight forward patch to `eth.c` in the Linux kernel to mangle the source MAC address. This behaviour can be configured through the `/proc` file system.

To globally enable this behaviour set `/proc/sys/net/ipv4/conf/all/outgoing_mac` to 1. Any non-zero value enables the feature. The MAC address may be set on a per-interface basis by modifying `/proc/sys/net/ipv4/conf/<interface>/outgoing_mac`. A value of 0 disables the behaviour while any non-zero value will be used as the MAC address for frames sent from the corresponding interface.

Packets are always received using the MAC address of the NIC. And if the outgoing_mac behaviour is disabled then the address of the NIC is used to send packets. This is the normal behaviour for an interface.

Common MAC and IP Address: Group Mac Addresses

If the first bit of a MAC address is zero, then this is an individual address. If the first bit is 1, then it is a group address. Group addresses are intended for use with multicast and broadcast. It was thought that group addresses should be used for this project. However, RFC1122[1] specifies various restrictions on how packets with a multicast source or destination MAC address should be used that make their use for unicast IP traffic impractical. More specifically:

> Section 3.2.2
>
> *"An ICMP Message MUST NOT be sent as the result of receiving... a datagram sent as a link-layer broadcast..."*
>
> Section 3.3.6
>
> *"When a host sends a datagram to a link-layer broadcast address, the IP destination address MUST be a legal IP broadcast or IP multicast address."*
>
> *"A host SHOULD silently discard a datagram that is received via a link-layer broadcast (see Section 2.4) but does not specify an IP multicast or broadcast destination address."*

Implementation: Filtering

Simple filtering, such as described by Hubert[6] can be set up using a simple ipchains rule on each host. However, the static nature of this approach is highly undesirable. For this reason a method of dynamically filtering traffic is used.

The heart of the filtering is a a netfilter[11] kernel module, *ipt_saru*. While most netfilter modules work such that the packets that they match are configured statically using iptables. The *ipt_saru* module is set as an netfilter match and initialised to a sane default setting by writing a module for iptables, *libipt_saru*. *ipt_saru* then allows the packets that it matches to be configured using a setsockopt from user-space. A daemon *saru* monitors the status of other linux directors using heartbeat and sets the configuration of *ipt_saru* on the fly.

The block diagram shows how these components fit together. A detailed explanation follows.

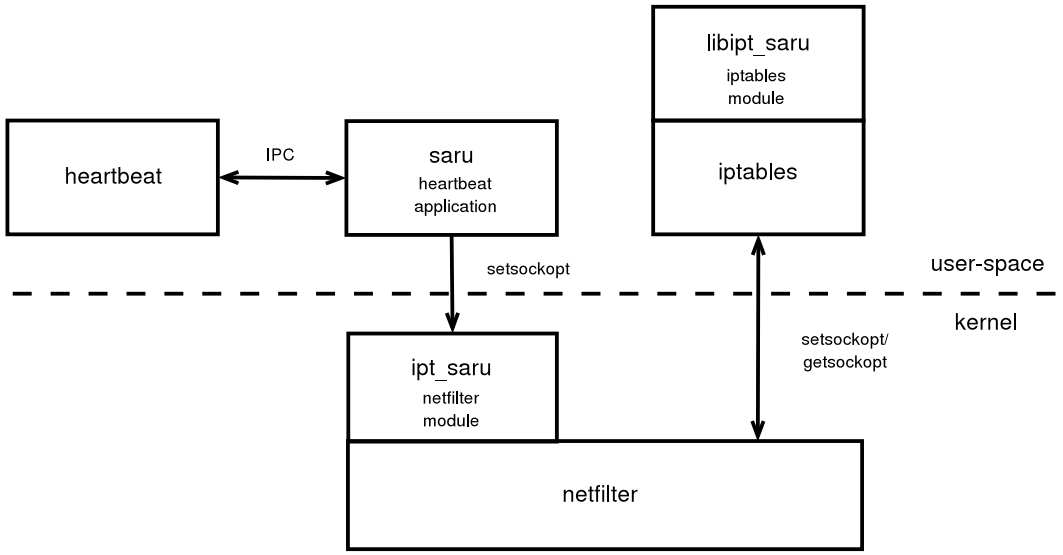


Figure 7: Block Diagram

netfilter:	Firewalling, NAT and packet mangling subsystem in the Linux 2.4 kernel. (existing code)
libipt_saru:	Iptables module that allows ipt_saru to be set as a match for netfilter. (existing code)
iptables:	User-space, command-line configuration tool for netfilter. (existing code)
ipt_saru:	Netfilter kernel module that allows packet matching to be configured on the fly. (new code)
heartbeat:	Monitors the status of nodes using a heartbeat protocol. (existing code)
saru:	Heartbeat application that determines what matches ipt_saru should make. (new code)

Filter: Source and Destination Ports and IP Addresses

The current filtering implementation is somewhat similar. It allows incoming connections to be allocated to a host based on its source or destination port or IP address.

It is thought by the author that filtering on the source port, offers the most even load balancing. Connections for IP based services generally have an ephemeral port as the source port. While the exact selection of these ports varies between different operating systems, there is a large range of them, at least several thousand, often tens of thousands[10]. Given the size of this range, it seems reasonable to expect that connections from each end-user's machine will come in from many different ports.

However, such a simple scheme has its limitations and better methods could be developed. This can be achieved by enhancing *ipt_saru*.

Filter: Blocking Strategies

The number of individual matches for the incoming packet filter is likely to be quite large. In the case of using the source or destination port, there are $2^{16} (\approx 64,000,000)$ possible ports. In the case of using the source or destination IP address there are $2^{32} (\approx 4,300,000,000,000)$ for IPv4. More complex filtering criteria are likely to have even larger numbers of possibilities. Clearly, it is not practical to have an individual filtering rule for each possibility. To alleviate this problem, a blocking scheme is used.

The result space is divided into a fixed number of blocks. For example, if the possible ports are divided into 512 blocks, then each block contains 128 ports: 0-127,128-255,...,65408-65545.

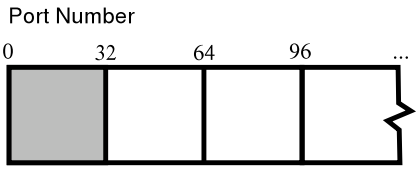


Figure 8: Port Blocks

For IP addresses the result space is divided up by using the modulus of the least-significant 16 bits, thus there are 2^{16} sub-blocks of 32 addresses per block.

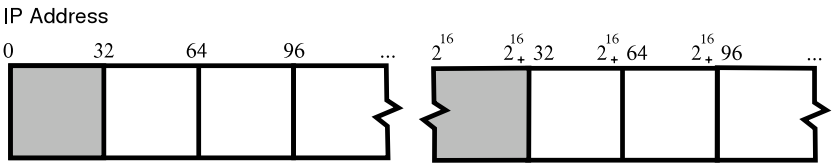


Figure 9: IP Address Blocks

Blocks are allocated to each linux director using a modulus - if there are two linux directors then each linux director gets every second block. Using this scheme, if some areas of the filter space are not used as heavily as others, each linux director should all have some blocks in both the quieter and busier areas. An adaptive scheme that allocates blocks according to the relative load on each linux director may be developed in the future.

Conveniently these block allocations can be represented as a bit-map. For example, using ports again. If there are two linux directors and one is allocated the blocks 0-127,256-383,...,65280-65407 and the other the remaining blocks. Then the bit-map for this in hexadecimal would be 0xAA...A and 0x55...5 respectively.

Although the examples above focus on ports and IP addresses, this should be extensible to the results of any selection method whose result can be expressed as an integer in a known range. Given that the result needs to be generated for each incoming packet, it is envisaged that all selection methods will be simple, iterative processes that generate an integral result.

Filter: Linux Director Status

To allow blocks to be allocated to the linux directors that are available it is important to know the state of each linux director and to be able to allocate blocks accordingly. To do this a helper application, *saru*, was written for Heartbeat[2].

Heartbeat implements a heartbeat protocol. That is messages are sent at regular intervals between host and if a message is not received from a particular host then the the host is marked as being down. Heartbeat can be configured to manage resources - for instance a virtual IP address - according to this information. Heartbeat is able to monitor both other hosts running Heartbeat and any device that can respond to ping requests. The latter can be used to establish a simple quorum device. For instance, if the heartbeat hosts are configured to monitor the IP address of a switch they are connected to, or a router, then they can use this information to decide whether they can take over a resource. It may be desirable to configure things such that if a host is unable to access the switch on the external network, then it will not take over any resources. That is, if the quorum device is unavailable, then that host is not eligible to hold any resources.

Unfortunately, heartbeat's resource manager - the logic that decides when to take over or relinquish resources based on information on the status of monitored nodes - is somewhat limited and does not work at all for more than two nodes. For this reason when using heartbeat for this project it should not manage any resources. Rather, *saru* should receive messages from Heartbeat about the status of nodes and makes its own decisions about what resources it owns. In a nutshell, *saru* should use information from heartbeat to allocate the blocks in the result space to the active linux directors and use this information to configure *ipt_saru* locally.

Filter: Node State

The Node State is used to determine if a node is participating in the cluster and thus participates in elections and has blocks allocated to it.

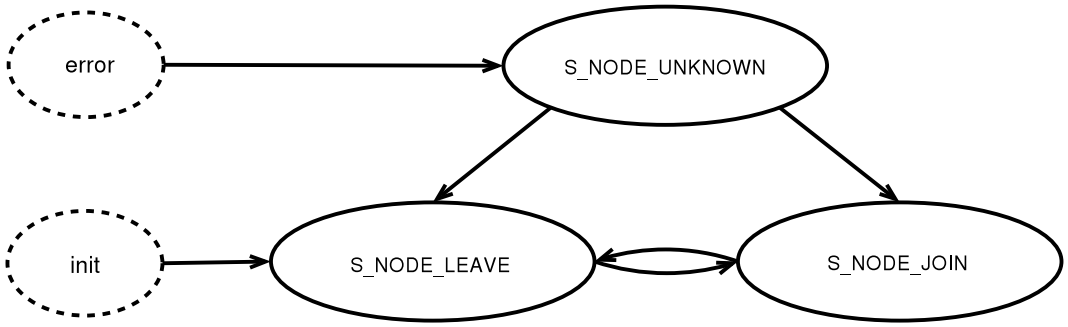


Figure 10: Node States

Node States

S_NODE_UNKNOWN	This is an error state. A node may move itself to the S_NODE_LEAVE or S_NODE_JOIN state from this state by sending a SARU_M_NODE_STATE_NOTIFY message.
S_NODE_LEAVE	The node is not able to participate in the cluster. This occurs when any of the configured quorum devices are unavailable. When all the configured quorum devices are available a node should move from this state to the S_NODE_JOIN state by sending a SARU_M_NODE_STATE_NOTIFY message.
S_NODE_JOIN	The node is able to participate in the cluster. This occurs when all of the configured quorum devices are unavailable. When any the configured quorum devices are become unavailable a node should move from this state to the S_NODE_LEAVE state by sending a SARU_M_NODE_STATE_NOTIFY message.

Node State Messages

M_NODE_STATE_NOTIFY	Sent by a any node to any other node or all nodes to advise its state or in response to a M_NODE_STATE_QUERY message. Should include the state, S_NODE_LEAVE or S_NODE_JOIN.
M_NODE_STATE_QUERY	Sent by master node to a slave node or all nodes in the cluster to determine their state. Nodes should reply with a M_NODE_STATE_NOTIFY message.

Filter: Master Election

If *saru* running on each linux director divides up the blocks in the results space, then it is possible that an inconsistent state may result between linux directors. This may result in some blocks being allocated to more than one linux director or some blocks being allocated to no linux director. Both of these situations are highly undesirable. For this reason it is thought that it is good to have a master node in the cluster.

Note that saru having a master and slave nodes is only a convenience for saru internally. The linux directors are still working in an active-active configuration to load balance network traffic.

When the cluster of linux directors starts up for the first time or the current master leaves the cluster, a master needs to be elected. When any node joins the cluster it needs to find out which nodes is the master so that it can request blocks from it. The following states and messages are used to allow a master to be elected and queried. The messages are sent via Heartbeat.

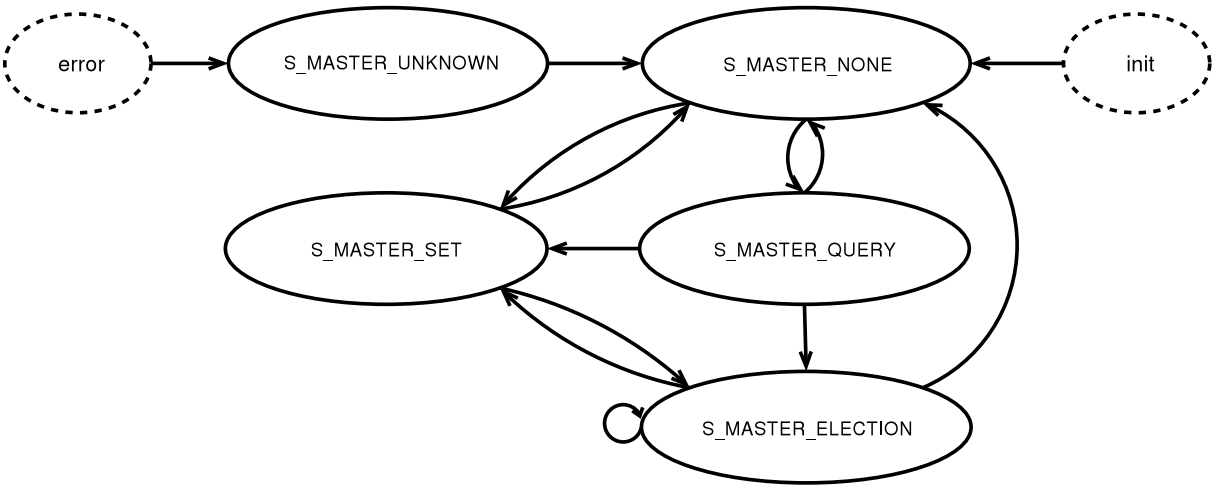


Figure 11: Master Election States

Master Election States

S_MASTER_UNKNOWN	This is an error state. A node may move itself to the S_MASTER_NONE state from this state.
S_MASTER_NONE	The node does not know who the master is. This should be the state of a node if it is not a member of the cluster. That is, the node should move to this state from any other state if it is not a member of the cluster. A node may move to the S_MASTER_QUERY state from this state by sending a M_MASTER_QUERY message.
S_MASTER_QUERY	The node does not know who the master is, has sent an M_BLOCK_QUERY message and is waiting for a reply. If the node receives a M_MASTER_NOTIFY message then it should move to the S_MASTER_SET state. After an internally defined timeout or if a M_MASTER_NOMINATION message is received, the node should send a M_MASTER_NOMINATION message and go to the S_MASTER_ELECTION state.
S_MASTER_ELECTION	The node does not know who the master is and has sent a M_MASTER_NOMINATION message. A node should move to this state from any other state by sending a M_MASTER_NOMINATION message if it receives a M_MASTER_NOMINATION message. On receipt of a M_MASTER_NOTIFY message, the node should tally the election results, obtained from information in any M_MASTER_ELECTION messages received and calculate a winner. If the master in the M_MASTER_NOTIFY message matches the winner calculated then, this winner should be announced in a M_MASTER_NOTIFY message and the node should move to the S_MASTER_SET state. Otherwise the node should send a M_MASTER_NOMINATION and reenter the S_MASTER_ELECTION state. After an internally defined timeout the node should calculate a winner and announce this winner by in a M_MASTER_NOTIFY message and move to the S_MASTER_SET state.
S_MASTER_SET	The node knows who the master is. The node should send a M_MASTER_RETIRE if it is the master and is going to the S_MASTER_NONE state. If the node is the master, then on receipt of a M_MASTER_QUERY it should respond with a M_MASTER_NOTIFY message. Nodes may asynchronously send M_MASTER_NOTIFY messages. On receipt of a M_MASTER_NOTIFY message that does not match the current master, the node should send a M_MASTER_NOMINATION and go into the S_MASTER_ELECTION state.

Note: A node is regarded to have left the cluster if any of the available quorum devices are inactive.

Master Election Messages	
M_MASTER_QUERY	Sent by a node to find out who the master node is. The master node should reply with a M_MASTER_NOTIFY message.
M_MASTER_NOTIFY	Sent by the master node to a slave node or all nodes in the cluster in response to a M_MASTER_QUERY message. Also sent to all nodes in the cluster by any node when a new master is elected. Should include the name of the master node.
M_MASTER_NOMINATION	Sent by a node when entering the S_MASTER_ELECTION state. Includes a metric, presumably the generation of the node, that is used to determine who wins the election. This metric should be regenerated by each node, each election.
M_MASTER_RETIRE	Sent by the master node if it stops being the master.

Note: All messages include the name of the node that they were sent from. This may be used to determine which node to reply to.

Filter: Finding Our Own Identity

Heartbeat associates a node name with each node in the cluster. However, this is somewhat inefficient and by using an identity number in the range from 0 to 255 a bitmap can be used to allocate blocks to the nodes. The id of 0 is reserved to mark nodes whose identity number is unknown. Thus, the effective range of identity numbers is from 1 to 255.

The master node allocates an identity number to each node. A slave node requests its identity number from the master node when it joins the cluster. It also requests its identity number when a the master node changes. To avoid the identity number of a given node in the cluster needlessly changing, all nodes should listen to all identity number notifications and remember the association between node names and identity numbers. In this way, if a slave node subsequently becomes the master node, it can use this information as a base for the identity numbers it will issue.

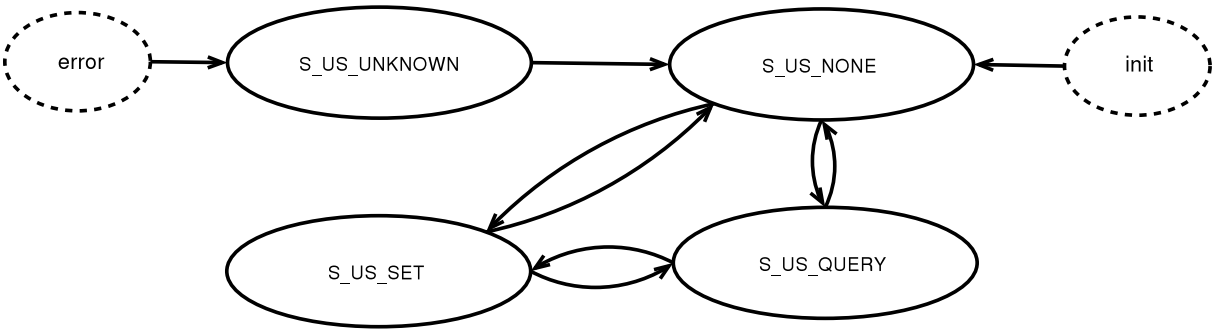


Figure 12: Identity Management States

Identity Management States

S_US_UNKNOWN	This is an error state. A node may move itself to the S_US_NONE state from this state.
S_US_NONE	The node does not know its own ID number. A node should move to this state from any other state if it is not a member of the cluster or if the master node changes. A node may move to the S_US_QUERY state from this state by sending a M_US_QUERY message.
S_US_QUERY	The node does not know its own ID number, has sent a M_US_QUERY message and is awaiting a reply. After an internally defined timeout the node should move to the S_US_NONE state.
S_US_SET	The node knows its own ID number. A node may move to this state from any other state if it receives a M_US_NOTIFY message and sets its ID number accordingly. A node may move to this state from any other state if it is the master node. It should allocate id numbers to all nodes in the cluster.

Identity Management Messages

M_US_QUERY	Sent by a slave node to the master node to find its identity number. If this message is received by a slave node it should be ignored.
M_US_NOTIFY	Sent by a master node to a any other node or all nodes to associate an identity number with a node name. This should be the response to a M_US_QUERY. All nodes that receive this message should update their table of node names and identity numbers accordingly. This message should include the node's id number and the name.

Note: All messages include the name of the node that they were sent from. This may be used to determine which node to reply to.

Filter: Obtaining Blocks

The master node allocates blocks to all the linux directors. Slave nodes may request blocks from the master. These requests can be made via heartbeat. The following states and messages are used to allow blocks to be allocated by the master.

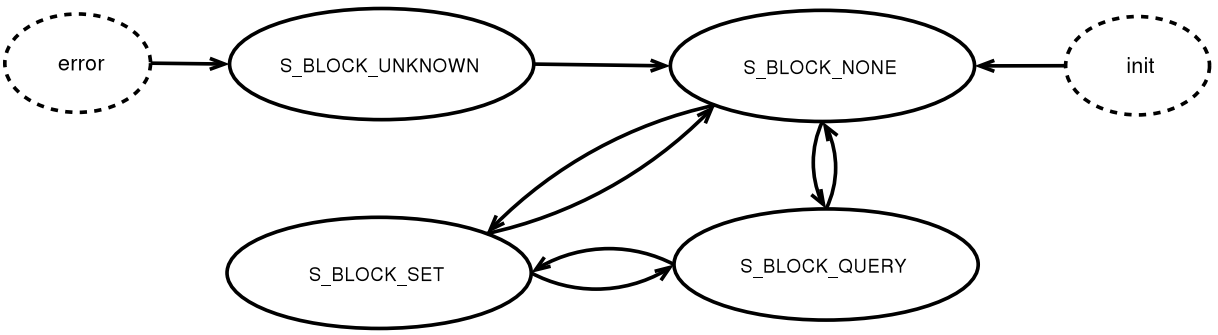


Figure 13: Block Management States

Block Management States

S_BLOCK_UNKNOWN	This is an error state. A node may move itself to the S_BLOCK_NONE state from this state.
S_BLOCK_NONE	The node has no blocks allocated. This should be the state of a node while it is not a member of the cluster. That is, the node should move to this state from any other state if it is not a member of the cluster. A node may move to the S_BLOCK_QUERY state from this state by sending a M_BLOCK_QUERY message.
S_BLOCK_QUERY	The node has no blocks allocated, has sent a M_BLOCK_QUERY message and is awaiting a reply. After an internally defined timeout the node should move to the S_BLOCK_NONE state.
S_BLOCK_SET	The node has blocks allocated. A node may move to this state from any other state if it receives a M_BLOCK_GRANT message and sets its allocated blocks accordingly. A node may move to this state from any other state if it is the master node. It may send a M_BLOCK_QUERY message to all nodes in the cluster and wait for M_BLOCK_USED replies until an internally defined timeout. It should allocate the internal block space and send a M_BLOCK_GRANT to all nodes in the cluster.

Note: A node is regarded to have left the cluster if any of the available quorum devices are inactive.

Block Management Messages

M_BLOCK_GRANT	Sent by a master node to grant blocks. Includes a bitmap of the blocks being granted. May be sent to an individual slave node or all nodes in the cluster. May be sent in response to a M_BLOCK_REQUEST message or asynchronously. In either case all recipient nodes should respond with a M_BLOCK_ACCEPT message.
M_BLOCK_REQUEST	Sent by a slave node to the master node to request blocks. This should be done when the slave node joins the cluster. It may also be done to request additional blocks, which may or may not be granted. The master node should respond with a M_BLOCK_GRANT message.
M_BLOCK_ACCEPT	Sent by a slave node to the master node in response to a M_BLOCK_GRANT message. The master node may resend the M_BLOCK_GRANT message if this is node received within a internally defined timeout.
M_BLOCK_QUERY	Sent by any node to any other node or all nodes in the cluster to ask which block allocations that node is using. It is thought that nodes may have some blocks allocated that they are not using. These may be allocated to other nodes to allow more even load balancing. The node should respond with a M_BLOCK_USED message.
M_BLOCK_USED	Sent to the master node in response to a M_BLOCK_QUERY message. Includes a bitmap of the blocks being used.

Note: All messages include the name of the node that they were sent from. This may be used to determine which node to reply to.

Filter: Multiple Clusters

It is reasonable to expect that users may want to have different clusters of linux directors and to have a single linux director be a member or more than one cluster. For this reason a cluster id is used to uniquely identify a cluster. An unsigned 16bit number is used. This allows 65,000 unique clusters.

Connection Management

Given that the *saru* daemon changes the *ipt_saru* filtering rules on the fly, according to which linux directors are active in the cluster. It is reasonably to expect that the block that a given TCP connection belongs to, may be reallocated to a different linux director if linux directors are added or removed from the cluster. Thus steps should be taken to avoid these connections breaking in such circumstances.

Connection Management: Connection Synchronisation

One approach is to use Connection Synchronisation This synchronises information about active connections between the linux directors in the cluster, and allows a connection to continue even if its linux director changes. This is arguably the best solution for linux director, as it allows the load of existing and new connections to be distributed amongst the active linux directors by *saru*. However, if active-active is being used on real servers, that is hosts that terminate a connection, then this solution is not so attractive as most daemons that handle connections cannot deal with the connection changing machine mid-stream.

Connection Management: Connection Tracking

In this situation an alternative approach that allows established connections to continue being handled by their existing real server is needed. This can be achieved by using the Connection Tracking support of netfilter to accept packets for established connections. While packets for new connections are accepted or rejected by saru.

Unfortunately this solution can only work for TCP and not UDP as the latter does not have sufficient information in the protocol to distinguish between new and established connections. In fact, strictly speaking UDP does not have connections at all.

```
1: iptables -F
2: iptables -A INPUT -p tcp -d 172.17.60.201 \
        -m state --state INVALID -j DROP
3: iptables -A INPUT -p tcp -d 172.17.60.201 \
        -m state --state ESTABLISHED -j ACCEPT
4: iptables -A INPUT -p tcp -d 172.17.60.201 \
        -m state --state RELATED -j ACCEPT
5: iptables -A INPUT -p tcp -d 172.17.60.201 \
        -m state --state NEW \
        --tcp-flags SYN,ACK,FIN,RST SYN -m saru --id 1 -j ACCEPT
6: iptables -A INPUT -p tcp -d 172.17.60.201  -j DROP
```

Line 1: Flushes the iptables rules.

Line 2: Drops all packets that connection tracking detects are in an invalid state. We should not get any packets like this, but if we do it is a good idea to drop them.

Line 3: Accepts packets for established connections.

Line 4: Accepts packets for related connections. This usually means data connections for FTP or ICMP responses for some established connection.

By now we have accepted packets for all existing connections. It is now up to Saru to decide if we should accept a new connection or not.

Line 5: Accepts packets for connections that are:
NEW, that is we have not seen packets for this connection before. But This may be a connection that is established on another machine so we need extra filtering.
SYN,ACK,FIN,RST SYN examines the SYN, ACK, FIN and RST bits in the TCP packet and matches if only the SYN packet is set. This should be sufficient to isolate a packet that is the beginning of a TCP three-way handshake.
Saru accepts. Saru will accept the packet on exactly one of the active hosts in the cluster.

Line 6: Drops all other packets addressed to 172.17.60.201.

Typically the first packet for a connection will be accepted by line 5 and all subsequent packets will be accepted by line 3.

The order of lines 2-5 is not important.

More complex rules can be built up for multiple virtual services by creating a separate chain for lines 2-5 without the Virtual IP address included. And then branching to this chain based on a match on the Virtual IP address and optionally the port.

8 Putting It Together

Putting It Together: A Common MAC and IP Address

On Linux, both the MAC and IP address of an interface can be set using either the `ifconfig` or `ip` command on Linux. The following configures a host's eth0 interface with the MAC address 00:50:56:14:03:40 and adds the IP address 192.168.20.40.

```
ip link set eth0 down
ip link set eth0 address 00:50:56:14:03:40
ip link set eth0 up
ip route add default via 172.16.0.254
ip addr add dev eth0 192.168.20.40/24 broadcast 255.255.255.0
```

The results can be verified by using `ip addr sh eth0`. For example:

```
2: eth0: <BROADCAST,MULTICAST,UP; mtu 1500 qdisc pfifo_fast qlen 100
   link/ether 00:50:56:14:03:40 brd ff:ff:ff:ff:ff:ff
   inet 172.16.4.226/16 brd 172.16.255.255 scope global eth0
   inet 192.168.20.40/24 brd 255.255.255.0 scope global eth0
```

Note that 192.168.20.40 was *added* to the interface. The interface already had another addresses. This is useful, as a linux director can have a unique IP address as well as the common IP address to allow access directly to a particular linux director.

If necessary, set a bogus MAC address to be used as the source MAC address when sending frames. This example sets 00:50:56:14:03:43 as the address to be used when sending frames from eth0.

```
echo 1                  > /proc/sys/net/ipv4/conf/all/outgoing_mac
echo 00:50:56:14:03:43 > /proc/sys/net/ipv4/conf/eth0/outgoing_mac
```

Putting It Together: Filtering

Referring to the Block Diagram in Figure 7 and the explanation above, there is a daemon, *saru* that collects information about the other linux directors in the cluster from heartbeat. This information is used to determine which linux directors should be allocated blocks of ephemeral source ports to accept. This information is used to configure the *ipt_saru* netfilter module.

Iptables uses a module *libipt_saru* to add rules such that packets will be vetted based on the packet match made by the *ipt_saru* netfilter module. The following iptables rules should configure a virtual IP address, 192.168.20.40, to only accept packets as dictated by *ipt_saru* and in turn the *saru* daemon. `--id 1` indicates that the *saru* daemon monitoring cluster with id 1 should be used to manipulate the filter on the fly.

```
iptables -F
iptables -A INPUT -d 192.168.20.40 -p tcp \
-m saru --id 1  --sense src-port -j ACCEPT
iptables -A INPUT -d 192.168.20.40 -p udp \
-m saru --id 1  --sense src-port -j ACCEPT
iptables -A INPUT -d 192.168.20.40 -p icmp \
        -m icmp --icmp-type echo-request \
        -m saru --id 1  --sense src-addr -j ACCEPT
iptables -A INPUT -d 192.168.20.40 -p icmp \
        -m icmp --icmp-type ! echo-request -j ACCEPT
iptables -A INPUT -d 192.168.20.40  -j DROP
```

If LVS-NAT is being used then the following rules are also required to prevent all the linux directors sending replies on behalf of the the real servers. The example assumes that the real servers are on the 192.168.6.0/24 network.

```
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -d 192.168.6.0/24 -j ACCEPT
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -m state --state INVALID \
        -j DROP
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -m state --state ESTABLISHED \
        -j ACCEPT
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -m state --state RELATED \
        -j ACCEPT
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -p tcp -m state --state NEW \
        --tcp-flags SYN,ACK,FIN,RST SYN -m saru --id 1 -j MASQUERADE
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -p udp -m state --state NEW \
        -m saru --id 1 -j MASQUERADE
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -p icmp \
```

```
        -m icmp --icmp-type echo-request -m state --state NEW \
        -m saru --id 1 --sense dst-addr -j MASQUERADE
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -p icmp \
        -m icmp --icmp-type ! echo-request -m state --state NEW \
        -j MASQUERADE
iptables -t nat -A POSTROUTING -s 192.168.6.0/24 -j DROP
```

Given that the *saru* daemon changes the *ipt_saru* filtering rules on the fly, according to which linux directors are active in the cluster. It is reasonably to expect that the block that the source port of a given TCP connection belongs to, may be reallocated to a different linux director during the life of the connection. For this reason, it is important that TCP connections be synchronised from each active linux director to all other active linux directors using connection synchronisation.

Heartbeat should be configured with no local resources and at least one ping node which will act as a quorum device. That is, the *haresources* file should be empty and the *ha.cf* file should include at least one ping directive, a node directive for each linux director and at least one transport. An example, minimal *ha.cf* which uses multicast for communication and has the nodes `fred` and `mary` in the cluster follows.

```
mcast eth0 225.0.0.7 694 1 1
ping  192.168.0.254
node  fred
node  mary
```

The *saru* daemon may be started manually, at boot time using it's init script, or by adding the following line to the *ha.cf* file.

```
respawn root /path/to/saru
```

By default saru waits for a short time after starting before initialising its filter. This is to allow connection synchronisation information time to propagate. This and other options can be configured on the command line or by editing the *saru.conf* file. See the saru man page for details.

Once *saru* is up and running the state of the filtering rules can be inspected by examining */proc/net/saru_map*. Below is an example showing that the filter rule for cluster `id 1` has been set with a bitmap of all '5's. Please note that the bitmap has been truncated for formating reasons.

```
Id   RefCount BitMap
0001 00000001 55555555555555...
```

As discussed in *Connection Management* either Connection Synchronisation of Connection Tracking should be used in conjunction with *saru* to avoid connections breaking unnecessarily when linux directors are added or removed from the cluster.

Conclusion

Connection Synchronisation

Connection synchronisation between linux directors allows connections to continue after a linux director fail-over occurs. This can occur either if the active linux director fails or is taken down for maintenance.

The method discussed in part 1 of this paper establishes a peer-to-peer relationship between the linux directors using multicast. This allows connections being handled by any linux director to be efficiently synchronised to all other available directors. Thus, no mater which director is active, if it becomes unavailable and at least one other director is available, then fail-over should occur and active connections should continue.

The implementation modifies the existing LVS synchronisation code to allow different synchronisation methods to be registered. The method registered forwards synchronisation information to a user space daemon, where it is processed and distributed over multicast. Thus, much of the synchronisation logic was moved out of the kernel and into user-space, allowing a more sophisticated daemon to be built using existing user-space libraries and debugging tools.

Active-Active

Connections can be distributed between multiple active linux directors by assigning all of the linux directors a common MAC and IP address. This is a relatively straight forward process. This should work on both switched and non-switched ethernet networks. Behaviour in non-ethernet environments is beyond the scope of this project.

In order to prevent duplicate packets - which would probably result in TCP connection failure - it is important than only one of the active linux directors accept the packets for a given TCP connection. By utilising the Netfilter packet filtering infrastructure in the Linux 2.4 kernel and the monitoring capabilities of Heartbeat it is possible to build a system that dynamically updates what traffic is accepted by each linux director in a cluster. In this way it is possible to balance incoming traffic between multiple active linux directors.

This solution should scale to at least 16 nodes. Both heartbeat and the connection synchronisation daemon use multicast for communication. Thus, as the number of nodes increases, the amount of inter-linux director communication should increase linearly. It is thought that the real limitation will be either switch bandwidth, or the handling of packets, most of which will be dropped by any given node, by netfilter.

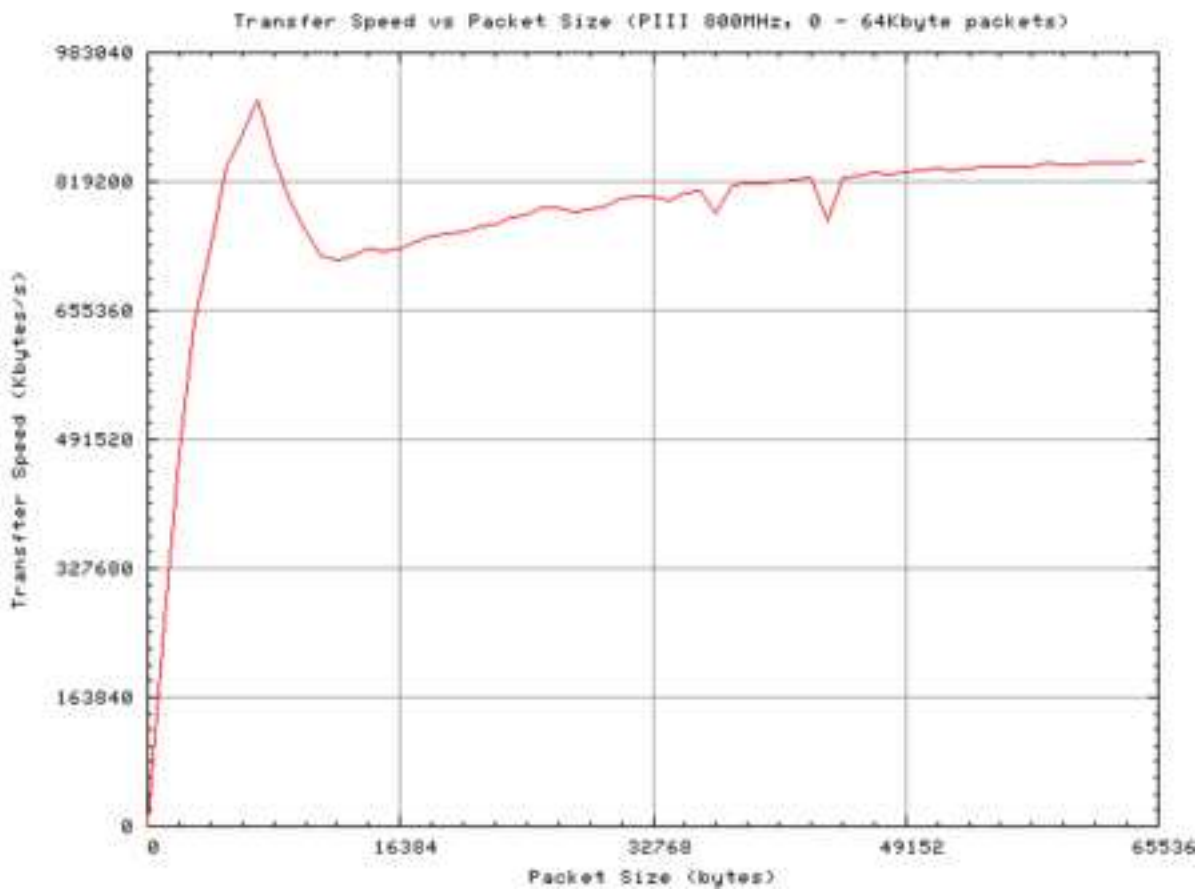
The design allows for fast migration of existing connections and resources to accept new connections in the case of fail-over. Something that I do not believe can be said for DNS based solutions.

I believe that this design provides the best possible active-active solution for linux directors. Interestingly this design could be used to load balance any linux machines, and could if appropriate, be used in place of Layer 4 Switching.

Appendix A Netlink Socket Throughput

The performance of netlink socket communication between user-space and the kernel is quite fast as shown in the charts below. The chart transfer speed against packet size for packets from 0 to 64Kbytes.

The graphs show that on a Pentium III 800MHz transfer rates of in excess of 660000Kbytes/s (5.0Gbits/s) are attainable for packets over 3100bytes in size. The graphs also indicate that there is a sweet spot at 7200bytes. At this point a transfer rate of 920000Kbits/s (7.2Gbits/s) is obtained.



References

[1] Internet Engineering Task Force. Editor: R. Braden. Rfc 1122: Requirements for internet hosts – communication layers. http://www.ietf.org/, October 1989.

[2] Alan Robertson et al. Heartbeat. http://www.linux-ha.org/heartbeat/, 1999–2003.

[3] Wensong Zhang et al. Linux virtual server project. http://www.linuxvirtualserver.org/, 1998–2003.

[4] Simon Horman. Ultra monkey. http://www.ultramonkey.org/, 1999–2003.

[5] Simon Horman. Creating linux web farms. http://www.vergenet.net/linux/, 2000.

[6] Bert Hubert. How to do simple loadbalancing with linux without a single point of failure. http://lartc.org/autoloadbalance.php3, 2001.

[7] IEEE. Ieee std 802.3: Csma/cd access method and physical layer specification. http://standards.ieee.org/, 2000.

[8] Joseph Mack. Lvs-howto. http://www.austintek.com/LVS/LVS-HOWTO/, December 2003.

[9] David C. Plummer. Rfc 826: An ethernet address resolution protocol or converting network protocol addresses to 48.bit ethernet addresses for transmission on ethernet hardware. http://www.ietf.org/, November 1982.

[10] W. Richard Stephens. *Unix Network Programming*, volume 1, pages 42–43. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 1998.

[11] Netfilter Core Team. Netfilter – firewalling, nat and packet mangling for linux 2.4. http://www.netfilter.org/, 2003.