Linux Virtual Server Clusters

Feature Story

Written by Wensong Zhang and Wenzhuo Zhang

Saturday, 15 November 2003

With the explosive growth of the Internet and its increasingly important role in our daily lives, traffic on the Internet is increasing dramatically, more than doubling every year. However, as demand and traffic increases, more and more sites are challenged to keep up, literally, particularly during peak periods of activity. Downtime or even delays can be disastrous, forcing customers and profits to go elsewhere. The solution? Redundancy, redundancy, and redundancy. Use hardware and software to build highly-available and highly-scalable network services.

Started in 1998, the Linux Virtual Server (LVS) project combines multiple physical servers into one virtual server, eliminating single points of failure (SPOF). Built with off-the-shelf components, LVS is already in use in some of the highest-trafficked sites on the Web. Need 24x7 service? Try 7-11 for that gallon of milk. Try LVS for your machine room.

As more and more companies move their mission-critical applications onto the Internet, the demand for always-on services is growing. So too is the need for highly-available and highly-scalable network services. Yet the requirements for always-on service are quite onerous:

* The service must scale: when the service workload increases, the system must scale up to meet the requirements.

* The service must always be on and available, despite transient partial hardware and software failures.

* The system must be cost-effective: the whole system must be economical to build and expand.

* Although the whole system may be big in physical size, it should be easy to manage.

As you might infer, a single machine, no matter how large (read: expensive), is rarely able to meet the demands of a highly-available service. A single server is usually insufficient to handle aggressively increasing loads; upgrading a single server is usually quite complex; and higher-end servers are proportionally more expensive. And ultimately, one server remains a single point of failure.

Clusters of servers, interconnected by a fast network, are emerging as a viable architecture for building a high-performance and highly-available service. This type of loosely-coupled architecture is more scalable, more cost-effective, and more reliable than a single processor system or a tightly-coupled multiprocessor system. However, there are challenges, including transparency and efficiency.

From Many, One

The Linux Virtual Server is one solution that meets the requirements and challenges of providing an always-on service. In LVS, a cluster of Linux servers appear as a single (virtual) server on a single IP address. Client applications interact with the cluster as if it were a single, high-performance, and highly-available server. Inside the virtual server, LVS directs incoming network connections to the different servers according to scheduling algorithms. Scalability is achieved by transparently adding or removing nodes in the cluster. High availability is provided by detecting node or daemon failures and reconfiguring the system accordingly, on-the-fly.

For transparency, scalability, availability and manageability, LVS is designed around a three-tier architecture, as illustrated in *Figure One.* The three-tier architecture consists of:

A load balancer, which serves as the front-end of the whole cluster system. It distributes requests from clients among a set of servers, and monitors the backend servers and the other, backup load balancer.

A set of servers, running actual network services, such as Web, email, FTP and DNS.

Shared storage, providing a shared storage space for the servers, making it easy for the servers to have the same content and provide consistent services.

The load balancer, servers, and shared storage are usually connected by a high-speed network, such as 100 Mbps Ethernet or Gigabit Ethernet, so that the intranetwork does not become a bottleneck of the system as the cluster grows.

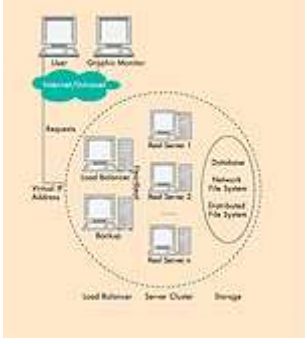

Figure One: The architecture of a Linux Virtual Server

The load balancer is the single entry-point of server cluster systems. Clients connect to a single, known IP address, and the load balancer redirects the incoming connections to the servers that actually perform the work. The load balancer can run *IPVS* -- IP load balancing techniques inside the Linux kernel, described in the "IPVS" sidebar -- or *Kernel TCP Virtual Server* (KTCPVS) that implements application-level load balancing inside the Linux kernel.

IPVS

IPVS modifies the TCP/IP stack inside the Linux kernel (versions 2.0, 2.2, 2.4, and 2.5), to support IP load balancing technologies. The system implementation of IPVS is illustrated in *Figure Three.*

The IPVS Schedule & Control Module is the main module of IPVS. It hooks two places in the kernel to grab and rewrite IP packets to support IP load balancing. It looks up the IPVS Rules hash table for new connections, and checks the Connection Hash Table for established connections.

The IPVSADM user-space program administers virtual servers, can write the virtual server rules inside the kernel through `setsockopt()`, and can read the IPVS rules through `getsockopt()` or the */proc* file system.

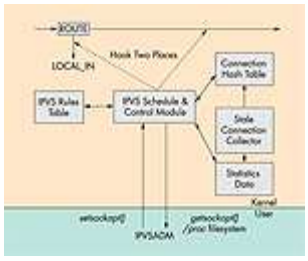

Figure Three: How IPVS works

The connection hash table is designed to hold millions of concurrent connections, and each connection entry only occupies 128 bytes of effective memory in the load balancer. So, a load balancer with 256 MB free memory can maintain two million concurrent connections.

The hash table size can be customized according to the applications. The client is used as the hash key so that hash collision is very low. A slow timer is ticked every second to collect stale connections.

IPVS also implements ICMP handling for virtual services. The incoming ICMP packets for virtual services are forwarded to the real servers, and outgoing ICMP packets from virtual services are altered and sent out. This is important for error and control notification between clients and servers, such as the MTU discovery.

Different kinds of IP load balancing techniques can be used for different kinds of server clusters. Different techniques can also be used in the same cluster at the same time: some packets can be forwarded to some servers via the VS/NAT method, some via VS/DR, and still some others via VS/TUN to geographically distributed servers.

When IPVS is used, each server is required to provide the same services and content, and the load balancer forwards a new request to a server according to the scheduling algorithm and the load of each server. No matter which server is selected, the requesting client gets the same result. (IPVS supports three three load balancing techniques: *VS/NAT, VS/TUN,* and *VS/DR*. See the sidebar "Three Ways to Balance Load" for more information.)

Three Ways to Balance Load

IP load balancing techniques are quite scalable, and IPVS supports three different load balancing techniques: *Virtual Server via NAT* (VS/NAT), *Virtual Server via Tunneling* (VS/TUN), and *Virtual Server via Direct Routing* (VS/DR).

Virtual Server via NAT (VS/NAT)

Due to security considerations and the shortage of IP addresses in IPv4, more and more networks use private IP addresses that aren't allocated on the Internet. *Network address*

translation is needed when hosts in internal networks want to access the Internet, or need to be accessed from the Internet. NAT can also be used to build a virtual server: parallel services at different IP addresses can appear as a virtual service on a single IP address. The architecture of Virtual Server via NAT is illustrated in *Figure Four.* The load balancer and real servers are interconnected by a switch or a hub.

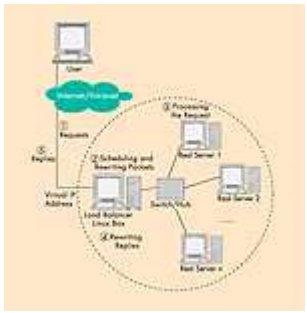

Figure Four: Architecture of Virtual Server via NAT

The workflow of VS/NAT is as follows:

1. When a user accesses a virtual service provided by the server cluster, a request packet destined for the virtual IP address (the IP address to accept requests for virtual service) arrives at the load balancer.

2. The load balancer examines the packet's destination address and port number. If they match a virtual service in the virtual server rule table, a real server is selected from the cluster by a scheduling algorithm and the connection is added to hash table that records connections. Then, the destination address and the port of the packet are rewritten to those of the selected server, and the packet is forwarded to the server. When an incoming packet belongs to an established connection, the connection can be found in the hash table and the packet is rewritten and forwarded to the right server.

3. The request is processed by one of the physical servers.

4. When response packets come back, the load balancer rewrites the source address and port of the packets to those of the virtual service. When a connection terminates or timeouts, the connection record is removed from the hash table.

5. A reply is sent back to the user.

Virtual Server via IP Tunneling (VS/TUN)

IP tunneling (also called *IP encapsulation*) is a technique to encapsulate IP datagrams within IP datagrams, which allows datagrams destined for one IP address to be wrapped and redirected to another IP address.

This technique can also be used to build a virtual server: the load balancer tunnels the request packets to the different servers, the servers process the requests, and return the results to the clients directly. Thus, the service appears as a virtual service on a single IP address. The architecture of Virtual Server via IP Tunneling is illustrated in *Figure Five*.

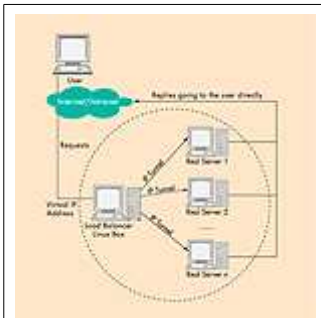

Figure Five: Architecture of Virtual Server via IP Tunneling

In the figure, the real servers can have any real IP addresses in any network, and the servers can be geographically distributed. However, each server must support the IP tunneling protocol, and each must have one of their tunnel devices configured with virtual IP.

The flow of VS/TUN is the same as that of VS/NAT. In VS/TUN, the load balancer encapsulates the packet within an IP datagram and forwards it to a dynamically selected server. When the server receives the encapsulated packet, it decapsulates the packet, finds the inner packet destined for the virtual IP address on its tunnel device, processes the request, and returns the result to the user directly.

Virtual Server via Direct Routing (VS/DR)

The VS/DR balancing approach is similar to the one implemented in IBM's NetDispatcher. The architecture of VS/DR is illustrated in *Figure Six*.

The load balancer and the real servers must have one of their interfaces physically linked by an uninterrupted segment of LAN such as an Ethernet switch. The virtual IP address is shared by real servers and the load balancer. Each real server has a non-ARPing, loopback alias interface configured with the virtual IP address, and the load balancer has an interface configured with the virtual IP address to accept incoming packets.

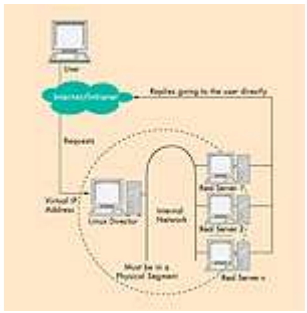

Figure Six: Architecture of Virtual Server via Direct Routing

The workflow of VS/DR is similar to that of VS/NAT or VS/TUN. In VS/DR, the load balancer directly routes a packet to the selected server (the load balancer simply changes the MAC address of the data frame to that of the server and retransmits it on the LAN). When the server receives the forwarded packet, the server determines that the packet is for the address on its loopback alias interface, processes the request, and finally returns the result directly to the user.

The logical interfaces of the real servers that are configured with virtual IP address should not respond to ARP requests; otherwise, there would be an ARP conflict between the interface that accepts incoming traffic for virtual IP and the interfaces of real servers in the same network.

Comparing Technique

Which technique is best? For most Internet services (such as web hosting) where request packets are often short and response packets usually carry large amount of data, a VS/TUN load balancer can easily schedule over 100 real servers and won't become the bottleneck of the system. The load balancer just directs requests to the servers and the servers reply to the clients directly. While VS/TUN requires servers to support the IP tunneling protocol, the IP tunneling protocol is becoming a standard of all operating systems, and VS/TUN should be usable with servers running operating systems other than Linux.

When KTCPVS is used, servers can have different content, and the load balancer can forward a request to a different server according to the content of request. Since KTCPVS is implemented inside the Linux kernel, the overhead of relaying data is minimal, so that it can still have high throughput. However, KTCPVS is not very mature, so IPVS is preferred.

The number of server nodes can be changed based on demand. When all of the installed servers are overloaded, more servers can be readily added to the cluster to handle the increasing workload. For most Internet services such as serving Web pages, the requests are usually not highly interrelated, and can be run in parallel on different servers. Therefore, as the number of server nodes increases, the performance of the whole cluster scales up almost linearly.

Shared storage can be made up of database systems, networked file systems, or distributed file systems. Any data that server nodes need to update dynamically should be stored in a database, because database systems can guarantee the consistency of concurrent data access when server nodes read or write data in parallel. Static data is usually kept in networked file systems such as NFS or CIFS, so that data can be shared by all the server nodes. However, the scalability of a single networked file system is limited. For example, a single NFS/CIFS can only support concurrent data access from between four to eight servers. For large-scale cluster systems, distributed, scalable cluster file systems, such as *GPFS, Coda,* and *GFS,* can be used for shared storage.

High on Availability

One of the advantages of a clustered system is that it has *hardware and software redundancy*. High availability can be provided by detecting node or daemon failures, and then reconfiguring the cluster accordingly so that the workload can be taken over by the remaining nodes in the cluster.

Several software packages, including *Heartbeat* (described in the feature "Highly-Affordable High Availability," pg. 16), *Red Hat Piranha, Keepalived,* and *Ultra-Monkey*, can be used to provide high availability in conjunction with LVS.

Usually, there are service monitor daemons running in the load balancer to periodically check the health status of the servers. If there's no response to service access requests or ICMP ECHO REQUESTs from a server for a specified length of time, the service monitor considers the server "dead," and the server is removed from the list of available server lists. A dead server doesn't get any requests until it's deemed "alive" again.

In effect, the load balancer masks the failures of service daemons and hardware. Furthermore, administrators can use system tools to add new servers to increase the system throughput, or can remove servers for system maintenance, without bringing down the service as a whole.

You might notice in *Figure One* that the load balancer itself is now a single point of failure. To prevent that failure, the load balancer also needs a backup. A *heartbeat* daemon runs both on the primary and the backup, periodically exchanging messages like "I'm alive" to each other through serial lines and/or network interfaces. When the heartbeat daemon of the backup fails to receive heartbeat messages from the primary for a specified length of time, it assumes the virtual IP address and responsibilities of the load balancer. When the (former) primary load balancer comes back to work, there are two solutions: it becomes the backup, or resumes its role as the primary.

Besides coordination of roles, the primary and backup load balancer machines also have to share *state*. In particular, whatever machine is primary must maintain a record of which server each incoming connection was forwarded to. If the backup load balancer takes over without this connection information, the clients have to resend their requests. To make load balancer failover transparent to client applications, connections are synchronized using IPVS: the primary load balancer synchronizes connection information to the backup load balancer(s) through UDP multicast.

The Test Drive

Let's construct a highly-available VS/NAT web cluster with two load balancers and three web servers. The topology is illustrated in *Figure Two*.

In this configuration, virtual IP address and gateway IP address are 10.23.8.80 and 172.18.1.254 respectively, which are floating between the two load balancers (LD1 and LD2).

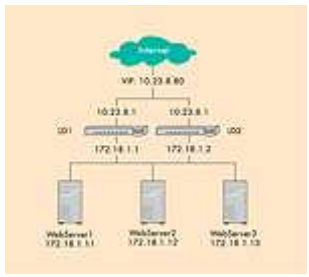

Figure Two: An example LVS cluster

First, let's build an IPVS enabled kernel. (Some Linux distributions and cluster managers provide pre-built kernel packages with IPVS enabled. However, it's always good to know how to build an IPVS-enabled kernel in case you want to upgrade to the latest release of the kernel or the newest IPVS patch.)

To enable IPVS in the Linux Kernel, start from a vanilla kernel source tree and an appropriate *.config* file for your hardware. The IPVS kernel code exists as a single kernel patch, which can be downloaded from the LVS web site. This example starts with the current stable kernel release v2.4.21 and IPVS patch *linux-2.4.21-ipvs-1.0.10.patch.gz.*

```
% cd linux-2.4.21
% gzip -cd linux-2.4.21-ipvs
  1.0.10.patch.gz | patch -p1
```

Here are the IPVS configurations in the 'make menucong' menu:

```
Networking Options -->
  IP: Virtual Server Configuration -->
    <M> virtual server support
      (EXPERIMENTAL)
    [ ] IP virtual server debugging
    (12) IPVS connection table size
      (the Nth power of 2)
    --- IPVS scheduler
    <M> round-robin scheduling
    <M> weighted round-robin scheduling
    <M> least-connection scheduling
      scheduling
    <M> weighted least-connection
```

```
      scheduling
    <M> locality-based least-connection
      scheduling
    <M> locality-based least-connection
      with replication
    scheduling
    <M> destination hashing scheduling
    <M> source hashing scheduling
    <M> shortest expected delay scheduling
    <M> never queue scheduling
    --- IPVS application helper
    <M> FTP protocol helper
```

This kernel configuration builds everything as kernel modules. Other relevant kernel configuration options include "TCP/IP networking" and "Network package filtering", and netfilter masquerading options (if you intend to use VS/NAT).

```
Networking Options -->
  <*> Packet socket
  [*] Packet socket: mmapped IO
  <*> Netlink device emulation
  [*] Network packet filtering
    (replaces ipchains)
  [ ] Network packet filtering debugging
  [*] Socket Filtering
  <*> Unix domain sockets
  [*] TCP/IP networking
  [ ] IP: multicasting
  [ ] IP: advanced router
  [ ] IP: kernel level autoconfiguration
  <M> IP: tunneling
  < > IP: GRE tunnels over IP
  [ ] IP: multicast routing
  [ ] IP: ARP daemon support
    (EXPERIMENTAL)
  [ ] IP: TCP Explicit Congestion
    Notification support
  [ ] IP: TCP syncookie support
    (disabled per default)
Networking Options -->
  IP: Netfilter Configurations
    <M> Connection tracking
      (required for masq/NAT)
      <M> Full NAT
      <M> MASQUERADE target support
      <M> REDIRECT target support
```

After configuring the kernel, compile and install it and its modules. This command is typically sufficient.

```
% make bzImage modules modules_install install
```

After installing the new kernel into the boot loader and rebooting into the new kernel, perform the following simple test:

1. Configure the network interfaces and routing tables of the web servers and the LinuxDirectors as per the diagram in *Figure Two.*

2. Setup the web services in the servers.

3. At last, run the following command on LD1:

```
# echo "1" > /proc/sys/net/ipv4/ip_forward
```

```
# iptables -t NAT -A POSTROUTING -s \
  172.18.1.0/24 -j MASQUERADE
# ipvsadm -A -t 10.23.8.80:80 -p 600
# ipvsadm -a -t 10.23.8.80:80 -R \
  172.18.1.11 -m -w 100
# ipvsadm -a -t 10.23.8.80:80 -R \
  172.18.1.12 -m -w 100
# ipvsadm -a -t 10.23.8.80:80 -R \
  172.18.1.13 -m -w 100
```

Use the browser to access the virtual IP 10.23.8.80 from the Internet, and use the command `ipvsadm -ln` on the LinuxDirector to list the virtual server table.

Keeping an Eye on Things

Once you have your cluster up and running, you can kick back and relax -- almost. Even with LVS in place, a *cluster monitor* can help detect and point out problems. Here's a very brief overview of some of the more popular cluster monitors:

* *Ultra Monkey* is a project to create load balanced and highly available services on a local area network using open source components and Linux. Ultramonkey includes *heartbeat* and *ldirectord* from the Linux-HA project. The configuration files for UltraMonkey are the same on LD1 and LD2, and are shown in *Listing One.*

Listing One: The configuration files for UltraMonkey

/etc/ha.d/ha.cf:

```
logfacility local0
keepalive 2
deadtime 10
warntime 10
initdead 10
nice_failback on
udpport 694
bcast eth1
node ld1
node ld2
```

/etc/ha.d/haresources:

```
ld1 IPaddr::10.23.8.80/24/eth1
  IPaddr::172.18.1.254/24/ \
  eth1 ldirectord::ldirectord.cf
```

/etc/ha.d/ldirectord.cf:

```
checktimeout=10
checkinterval=2
autoreload=no
logfile="local0"
quiescent=yes
virtual=10.23.8.80:80
fallback=127.0.0.1:80
real=172.18.1.11:80 masq
real=172.18.1.12:80 masq
real=172.18.1.13:80 masq
service=http
request="index.html"
receive="Test Page"
```

```
scheduler=wlc
persistent=600
protocol=tcp
checktype=negotiate
```

Piranha is the clustering product from Red Hat Inc. It includes the IPVS kernel code, a GUI-based cluster configuration tool and cluster monitoring tool. Again, the configuration file for Piranha is the same on load balancers LD1 and LD2, and is shown in *Listing Two*.

Listing Two: The configuration file for Piranha

```
primary = 10.23.8.1
service = lvs
rsh_command = rsh
backup_active = 1
backup = 10.23.8.2
heartbeat = 1
heartbeat_port = 539
keepalive = 4
deadtime = 12
network = nat
nat_router = 172.18.1.254 eth1:0
nat_nmask = 255.255.255.0
reservation_conflict_action = preempt
debug_level = NONE

virtual web {
  active = 1
  address = 10.23.8.80 eth0:1
  vip_nmask = 255.255.255.255
  port = 80
  persistent = 600
  send = "GET / HTTP/1.0\r\n\r\n"
  expect = "HTTP"
  load_monitor = none
  scheduler = wlc
  protocol = tcp
  timeout = 6
  reentry = 15
  quiesce_server = 0
  server webserver1 {
    address = 172.18.1.11
    active = 1
    weight = 100
  }
  server webserver2 {
    address = 172.18.1.12
    active = 1
    weight = 100
  }
  server webserver3 {
    address = 172.18.1.13
    active = 1
    weight = 100
  }
```

```
}
```

* *Keepalived* checks the health of LVS clusters. It implements a framework of health checking on multiple layers for server failover, and a VRRPv2 stack to handle director failover. In our example, the Keepalived configuration file */etc/keepalived/keepalived.conf* at LD1 looks like *Listing Three.*

Listing Three: The Keepalived configuration file on LD1

```
vrrp_sync_group VG1 {
  group {
    VI_1
    VI_2
    }
}

vrrp_instance VI_1 {
  state MASTER
  interface eth0
  virtual_router_id 51
  priority 100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass 1111
  }
  virtual_ipaddress {
  10.23.8.80
  }
}

vrrp_instance VI_2 {
  state MASTER
  interface eth1
  virtual_router_id 51
  priority 100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass 1111
  }
  virtual_ipaddress {
  172.18.1.254
  }
}

virtual_server 10.23.8.80 80 {
  delay_loop 6
  lb_algo wlc
  lb_kind NAT
  persistence_timeout 600
  protocol TCP
    real_server 172.18.1.11 80 {
      weight 100
      TCP_CHECK {
```

```
        connect_timeout 3
      }
    }
    real_server 172.18.1.12 80 {
      weight 100
      TCP_CHECK {
        connect_timeout 3
      }
    }
    real_server 172.18.1.13 80 {
    weight 100
      TCP_CHECK {
        connect_timeout 3
      }
    }
}
```

The Keepalived configuration file at LD2 is similar to that of LD1, except to change the state of `VI 1` and `VI 2` from MASTER to BACKUP.

Conclusion

LVS clusters provide load balancing and high availability for network services at low cost. The solutions require no modification to either the clients or the servers, and they support most TCP and UDP services. An LVS load balancer is designed to handle millions of concurrent connections.

Furthermore, it's easy to setup highly-available and highly-scalable network services using LVS clusters. Compared to other commercial products, LVS provides many unique features:

Supports three IP load balancing technologies, where VS/ TUN and VS/DR provide very high performance.

Supports multiple scheduling algorithms for dispatching connections to the real servers. Additional, custom schedulers can be added as loadable modules.

Offers many flexible and extensible service monitoring tools to maintain the high-availability of the whole system.

A robust and stable code base, a large user and developer base, an active community of users, and the maturity provided by worldwide peer review.

Proven reliability in real-world applications.

Best of all, LVS is free to everyone.

RESOURCES

The Coda Project

http://www.coda.cs.cmu.edu

Keepalived

http://www.keepalived.org

Ultramonkey

http://www.ultramonkey.org

Piranha

http://www.redhat.com/support/wpapers/piranha

The Linux Virtual Server Project

http://www.LinuxVirtualServer.org

*Wensong Zhang is the lead developer of the Linux Virtual Server. You can reach Wensong at **wensong@linux-vs.org**. You can reach Wenzhuo Zhang at **wenzhuo@zhmail.com**.*

For more information, follow these *Linux Magazine trails*:

Feature Story **November 2003** **Wensong Zhang** **Wenzhuo Zhang**

Close Window