Load-Balancing HA Clusters with No Single Point of Failure

Fábio Olivé Leite
<olive@conectiva.com.br>

Conectiva S.A.
Rua Tocantins, 89
Curitiba, PR, Brazil
CEP 80050-430

September, 2002

Abstract

High availability, load balancing clusters are a very active area of Linux development, and there are several efforts worldwide that aim to achieve such goals. Most of these efforts involve the concept of a redirector machine, that receives the clients' connections and distributes them among the working nodes (a server farm) using several different distribution policies, and that sometimes also route the server's answers. This redirector is a single point of failure that can render the whole cluster useless if it fails.

This work presents a new model for load balancing clusters that completely eliminates the need for a redirector, where all nodes are equal and each decides, using a distributed heuristic, whether it is the one that should accept a new connection. The key aspects here are: making sure all connections can reach all servers, and that exactly one server decides it should take an incoming connection.

1 Introduction

With the exponential growth of the Internet and usage of Internet technologies on corporate networks, traffic on TCP/IP based networks and servers is increasing dramatically. The load on any given server is likely to only grow in the future, specially when considering modern networking phenomena like *slashdotting*, when small companies strike news in that popular *News for Nerds* website, and the migration to "portal"-based intranets.

When you have a bigger problem to solve in your hands, you can usually go three directions [1]: you can compute faster, you can compute smarter, or you can call your friends.

There is not a lot that can be done on the server side about *computing faster* in the case of TCP/IP communication, and any serious server is usually already finetuned to its application. The simplistic approach of upgrading hardware is fine only until the next upgrade is necessary and its cost is high, not to mention the poor scalability. The same can be said about *computing smarter* in this sense. But *calling your friends* on a server network is certainly a very interesting solution, which is called *clustering*.

Building scalable cluster servers is a very interesting solution for this problem, since it's cheap and easy to add a few more servers to a server farm and thus increase the maximum workload by a large factor. There are several methods that can be used to build such clusters.

A very popular method is that of using a Round-Robin DNS server to map a server host name into several different IPs. The interesting side of this approach is that you can easily achieve global distribution of the workload, since there is no need for those IPs to be in the same network, or even in the same country. Disaster resiliency is also easy to achieve with such method. But then there are quite a few serious disadvantages to it.

With such a setup you are forced to use very small TTL (Time To Live) values for the DNS record, otherwise the load distribution will be coarse grained and bulky. But even then it *does* get coarse grained and bulky, since clients will cache the DNS mapping and use the same IP for a relatively long set of operations. Some caching DNS servers won't even respect small TTL values [3], and the ones that do respect them will create a high DNS traffic that will compete with application traffic.

This method is that of hiding several different server IPs under the same hostname. Another possibility is that of hiding several actual servers under the same IP. That IP is mapped to a hostname and is the only IP clients ever use. This is the Virtual Server approach, which is very well documented in the Linux Virtual Server website [2].

The Virtual Server approach uses a redirector machine, called a *load balancer*, that receives all clients' requests and redirects them to the *real* servers, which handle the actual work. This works very well and is very scalable, since the load on the redirector is low and has to do only with routing packets. The work being presented in this article goes a step further and intends to eliminate the redirector, which is a *Single Point Of Failure* that can render the whole cluster useless should it fail.

2 Related work

There are at least two other known attempts at achieving this goal, but both are based on slicing the connection space based on the client addresses and distributing different slices to different servers. This not only does not ensure proper load balancing, but also leads to entire network address ranges being unserved should a server fail and the slice not be taken by another server. These attempts also cause the servers to loose their network identity, as they should all assume the same physical and network address, which makes remote administration impossible.

One such attempt is by Bert Hubert, of Linux Advanced Routing and Traffic Control fame, in his web page entitled "Some evil ideas about decentralised load-balancing/clustering, with no single point of failure" [4]. This approach basically configures all servers' network cards with the same MAC addresses, adds an alias in every server for the cluster's IP and then configures several IP Tables rules to separate the connections based on the clients' IP addresses and to try to control the networking mess generated by having several machines with the same IP and MAC address.

This approach fails with network cards where you can't set the MAC address, which are quite common. It also does not ensure even load balance among the servers, since a certain client will always use the same server for all its connections. And last but not least, it needs to be used on a hub, or subvert a switch to act as a hub, since a switch won't be delivering the same packet to several ports unless it is a broadcast or multicast packet (or it *thinks* it is, which is the trick used in the proposed method).

There is also a comercial solution by VitaeSoft (`www.vitaesoft.com`), which is also based on splitting the client address space, but adds an interesting feature that allows one server to serve several client address slices, should servers in the cluster fail. This has the problems mentioned above, and has an additional "misfeature": it patches the kernel so that the ethernet drivers will accept packets for any MAC address [5]. This is clearly something that should be done better, and this article intends to exchange those problems for a single small TCP/IP subversion, which is explained in the next section.

3 The proposed model

Let's assume a common network where you have some clients and servers and routers to other networks and the Internet. Suppose there are two servers with which we want to create a cluster for HTTP, whose IP addresses are 10.0.17.27 and 10.0.17.28. Those two servers have each their own HTTP server listening on their own IP, and work fine when accessed individually by their own IP addresses. We want to have those two servers answer to the cluster's virtual IP address of 10.0.17.26.

For this we want two things: we want all packets going to 10.0.17.26 to reach both machines, and we want each machine to accept every other connection and not to mess with traffic that is not meant for it.

In order to get all traffic to the cluster IP to both machines we resort to a simple ARP hack: *map the cluster IP to a multicast MAC address, and have both machines join the corresponding IP multicast group.* This will ensure that switches deliver packets destined to that multicast MAC address to both machines. This hack is local to the servers' LAN, of course.

Now we have both machines receiving all packets to the cluster IP in the link level, but we still need to get the packets up the IP layer. The machines do not have the cluster IP configured in their network interfaces (so that they don't mess up the MAC address mapping for the cluster IP address), so we have to do Destination NAT from the cluster IP to each machine's own IP.

Now that all machines receive the packets to the cluster and think it is destined to themselves, we have to sort out two things: that the machines don't kill each other's traffic, and that only one machine accepts any given connection. This is easy with IP Tables, since you can use the state match to accept all `ESTABLISHED` and `RELATED` connections, and drop the rest. Now the only trick left is having only one machine accept a connection, and this is the place where all kinds of interesting things can be done.

A simple approach, and the one that is being used for testing by the author, is just have each machine accept every other connection (for a two-node cluster), or every third connection (for a three-node cluster), and so on, making sure all machines count connections initially offset by a certain number. This is done by the `nth` module for IP Tables, which is not in the stable distribution but can be applied if one suffers through patch-o-matic. `;)`

Another more interesting module would be one that uses that idle IP multicast address and does proper membership management, so that the cluster can dinamically adapt to failing or joining nodes, thus achieving High Availability. Such a module is planned to be implemented as soon as the author has time to do so. `:)` A more elegant solution would actually use such membership management to exchange load information among the servers, so that it could even provide *dynamic load balancing*, which is a lot more than just *load balancing*.

3.1 Goals

The model presented for load-balancing, high availability clusters with no single point of failure aims at obtaining a cluster where:

- *nodes in the cluster do not loose their network identity* by having to use the same MAC and/or IP address, thus making remote management possible;
- *behaviour alteration is kept to a minimum* so that the nodes in the cluster can offer services both using the cluster IP and their own IP, and that they can also offer any other service not being controlled by the cluster;
- *system intrusion is kept to a minimum* so that there are not a lot of preparations involved in setting up the nodes and the necessary code is modular.

The next section explains the technical details of how we get there.

3.2 Technical details

The first problem to solve is how we get the cluster's IP address to be mapped to a multicast MAC address. No machine has that IP configured, so no one is going to answer if a client sends an ARP request for that. Solving that is simple, we code a small application that will answer the ARP queries for the cluster IP with the correct MAC address. By running this application on all nodes in the cluster, the clients (the local clients, since the clients one router away just don't care about the MAC address for that IP) will get multiple ARP responses, but that's not a problem. Figure 1 presents example code extracted from such an application (error handling was ommited to save space).

After the "ARP answering machine" is running, the nodes will start getting the packets for the cluster's IP. And now there's a little price to pay for messing

```
...
        ETHER_MAP_IP_MULTICAST(&my_mcast, my_ha);

        /* Get sockets */
        arp_sock = socket(AF_PACKET, SOCK_DGRAM, htons(ETHERTYPE_ARP));
        mcast_sock = socket(AF_INET, SOCK_DGRAM, 0);

        /* Add multicast membership */
        mreq.imr_multiaddr = my_mcast;
        mreq.imr_address.s_addr = INADDR_ANY;
        mreq.imr_ifindex = 0;
        e = setsockopt(mcast_sock, SOL_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));

        /* Start answering ARP requests */
        while (!quit) {
                e = recvfrom(arp_sock, &req, sizeof(req), 0, (struct sockaddr *)&from, &fromlen);
                /* Check it's a valid request for our IP */
                if (req.arp_hrd == ntohs(ARPHRD_ETHER) &&
                                req.arp_pro == htons(ETHERTYPE_IP) &&
                                req.arp_hln == 6 &&
                                req.arp_pln == 4 &&
                                req.arp_op == htons(ARPOP_REQUEST) &&
                                !memcmp(req.arp_tpa, &my_ip, 4)) {
                        /* Fill in reply and send */
                        req.arp_op = htons(ARPOP_REPLY);
                        memcpy(req.arp_tha, req.arp_sha, ETH_ALEN + 4);
                        memcpy(req.arp_spa, &my_ip, sizeof(my_ip));
                        memcpy(req.arp_sha, my_ha, sizeof(my_ha));
                        e = sendto(arp_sock, &req, sizeof(req), 0, (struct sockaddr *)&from,\
                                fromlen);
                }
        }
...
```

Figure 1: Example code to send ARP replies

```
...
static unsigned int
target(struct sk_buff **pskb,
       unsigned int hooknum,
       const struct net_device *in,
       const struct net_device *out,
       const void *targinfo,
       void *userinfo)
{
        (*pskb)->pkt_type = PACKET_HOST;
        return IPT_CONTINUE;
}
...
static struct ipt_target ipt_packethost_reg
= { { NULL, NULL }, "PACKETHOST", target, checkentry, NULL, THIS_MODULE };
...
static int __init init(void)
{
        if (ipt_register_target(&ipt_packethost_reg))
                return -EINVAL;

        return 0;
}
...
```

Figure 2: PACKETHOST target to IP Tables

with the TCP/IP gods: Linux's ethernet layer will differentiate between unicast packets, directed to this host, and multi/broadcast packets, which are not directed, by setting the `pkt_type` field of the `skb`. Who cares about that? TCP. Figure 2 shows some extracted code from an IP Tables target that sets this field back to the value a unicast would have generated (`PACKET_HOST`), and that's it. An example `iptables` line would be: `iptables -t mangle -A PREROUTING -p tcp -d 10.0.17.26 --dport 80 -j PACKETHOST`. That means, change back to a unicast type the packets sent to the web server at the cluster's IP.

After this, it's a matter of DNATing the cluster IP to each machine's IP, with the `REDIRECT` target: `iptables -t nat -A PREROUTING -p tcp -d 10.0.17.26 --dport 80 -j REDIRECT`. Since we're just messing with HTTP packets here, all other traffic to the cluster's IP will be ignored. An interesting possibility is also redirecting ping packets, so that you can ping the cluster IP and easily know how many nodes are up: `iptables -t nat -A PREROUTING -d 10.0.17.26 -p icmp --icmp-type ping -j REDIRECT`.

After that, we should select the traffic, so that the nodes don't step on each other's toes:

```
iptables -t filter -A INPUT -p tcp --dport 80 -m state --state ESTABLISHED \
        -j ACCEPT
iptables -t filter -A INPUT -p tcp --dport 80 --syn -m nth --every 2 --start 1 \
        -j ACCEPT
iptables -t filter -A INPUT -p tcp --dport 80 -j DROP
```

Notice that those lines above use the `nth` module for IP Tables to provide for a simple connection selector. This obviously does not provide High Availability, since if a machine goes down the other still answers only every other connection. For a proof of concept, though, this works just fine.

That's it, with those configurations the cluster is working. Wasn't that simple? As soon as those IP Tables rules are run in both nodes (with the `--start` parameter being 0 in the other node), the cluster can start serving connections, and each server gets its share of the clients' requests.

4 Modules developed

Up to now there is the "ARP answering machine", which is a userlevel application, and the module that sets the packet type to the correct value. The `nth` module is available with Netfilter, although it may need to be enabled by hand.

The author has had little time to develop a selector module which handles node membership, but at the time this article is read there should be some initial code available.

5 Tests

This section shows what the packets that the client (10.0.17.20) sees look like when talking to the above mentioned cluster, consisting of machines 10.0.17.27 and 10.0.17.28, with a cluster IP of 10.0.17.26. First let's see some MAC addresses:

```
# ip n
10.0.17.27 dev eth0 lladdr 00:c0:df:01:c2:ca nud reachable
10.0.17.28 dev eth0 lladdr 00:50:04:71:a5:9e nud reachable
```

If we then ping the cluster's IP address, this is the ARP exchange seen on the wire:

```
# tethereal -p -n host 10.0.17.26
Capturing on eth0
  0.000000 00:c0:df:01:c2:b6 -> ff:ff:ff:ff:ff:ff ARP Who has 10.0.17.26?
        Tell 10.0.17.20
  0.000188 00:c0:df:01:c2:ca -> 00:c0:df:01:c2:b6 ARP 10.0.17.26 is at
        01:00:5e:00:01:14
  0.000220   10.0.17.20 -> 10.0.17.26   ICMP Echo (ping) request
  0.000388   10.0.17.26 -> 10.0.17.20   ICMP Echo (ping) reply
  0.000542   10.0.17.26 -> 10.0.17.20   ICMP Echo (ping) reply
  0.000596 00:50:04:71:a5:9e -> 00:c0:df:01:c2:b6 ARP 10.0.17.26 is at
        01:00:5e:00:01:14
```

The fact that the second ARP reply came later than the pings is not relevant, but the interesting bit is that we can see that the MAC addresses of the machines that answered correspond to the nodes in the cluster. This is how our ARP table looks like now:

```
# ip n
10.0.17.26 dev eth0 lladdr 01:00:5e:00:01:14 nud reachable
10.0.17.27 dev eth0 lladdr 00:c0:df:01:c2:ca nud reachable
10.0.17.28 dev eth0 lladdr 00:50:04:71:a5:9e nud reachable
```

We can see that the client accepts the multicast MAC address just fine. Now let's fetch an HTML page from `http://10.0.17.26/` and see what the packets look like (switched from tethereal to tcpdump to get the MAC address always there):

```
0:c0:df:1:c2:b6 1:0:5e:0:1:14 0800 74: 10.0.17.20.2635 > 10.0.17.26.80:
        S 2828703909:2828703909(0) win 5840 <mss 1460,sackOK,timestamp
        219104929 0,nop,wscale 0> (DF)
0:c0:df:1:c2:ca 0:c0:df:1:c2:b6 0800 74: 10.0.17.26.80 > 10.0.17.20.2635:
        S 2825654836:2825654836(0) ack 2828703910 win 5792 <mss 1460,sackOK,
        timestamp 178155 219104929,nop,wscale 0> (DF)
...
```

If we fetch it again we can see that it came from a different machine by the different MAC address in the responses (which need not be the multicast MAC address used by the client):

```
0:c0:df:1:c2:b6 1:0:5e:0:1:14 0800 74: 10.0.17.20.2631 > 10.0.17.26.80:
        S 2629657960:2629657960(0) win 5840 <mss 1460,sackOK,timestamp
        219085489 0,nop,wscale 0> (DF)
0:50:4:71:a5:9e 0:c0:df:1:c2:b6 0800 74: 10.0.17.26.80 > 10.0.17.20.2631:
        S 2632925614:2632925614(0) ack 2629657961 win 5792 <mss 1460,sackOK,
        timestamp 158168 219085489,nop,wscale 0> (DF)
...
```

This simple test shows that the cluster works, and that it does balance the connections from one node to the other. Of course more complete tests need to be shown, but unfortunately it will have to be for a later article. This same cluster has already been hit by several thousand requests without any problems, but the author will only publish such results after the selector module is implemented, so that the results and their analysis are more relevant.

6 Future work

This initial "framework" can be extended in very interesting ways. For example, it is interesting to mark the packets with the `MARK` target for IP Tables before using the `PACKETHOST` target, so that when filtering packets you can be sure to send to the selector only the packets actually addressed to the cluster IP. If you don't do that and a client connects to the IP of a server, the other servers won't see that packet and it will be counted only in the server who received it.

The selector module is the one that can be hacked the most: for example, it can be implemented so that a certain node can willingly leave the cluster and no longer accept any new connections, while still keeping its current established connections intact. An administrator could use that for node maintenance without loosing a single connection, by having the machine leave the cluster, wait until the current client connections terminate normally, do whatever he/she wants to the node, and then reinsert it into the cluster.

Using one selector "instance" per protocol would allow a cluster manager to direct the nodes into joining and leaving "service groups", thus dynamically adapting to varying load. For example, a cluster with four nodes could dedicate just two servers to SMTP and the other two to HTTP, but having more nodes join the HTTP group when the slashdot folks come knocking on the door, or the SMTP group when the spammers decide to entertain your users.

There certainly is a lot to enhance in the model proposed, but the good thing is that it works and can be easily extended. The author expects to be able to contribute valuable code and ideas in this interesting area.

References

[1] PFISTER, Gregory. **In Search of Clusters**. Prentice-Hall, 1998.

[2] Linux Virtual Server Project. `<http://www.linuxvirtualserver.org>`.

[3] Freemyer, Greg. Answer to a mail on the Linux-HA mailing list about "keep it simple high-availabilty". `<http://marc.theaimsgroup.com/?l=linux-ha&m=102804557604744&w=2>`.

[4] Hubert, Bert. **Some evil ideas about decentralised load balancing/clustering, with no single point of failure**. `<http://lartc.org/autoloadbalance.php3>`.

[5] VitaeSoft. **Quantizer v.0.2 — Load Balancing Software**. `<http://www.vitaesoft.com/registered/Balancing.html>`.