Configuring a high availability load balancing architecture with HAProxy and Keepalived

1. Introduction

This paper explores the ideas of redundancy, reliability and scalability by setting up a high availability load balancing architecture on a cluster of raspberry pis using HAProxy and Keepalived.

The practical is split up into two parts. The first part illustrates the concept of load balancing and redundancy by configuring a reverse proxy (haproxy) to distribute incoming traffic to an application across three different servers, in turn.
In the second part of the practical a second load balancer is introduced in a passive/active configuration to illustrate the idea of high availability.

2. Why High availability?

As the name suggests, high availability is used to describe the uptime, or the guarantee of uptime, of a given system [1].
Designing a system with high availability in mind means guaranteeing reliability and the ability to deal with failure, when it occurs. This can be done by introducing redundancy into the system and eliminating all single points of failure.

In life-critical sectors such as medicine or spaceflight, the lack of highly reliable, highly available, fault tolerant systems is particularly vital. If an application were to become unavailable under these conditions, the consequences could potentially be disastrous, or fatal. In business, financial and retail related applications this could mean the loss of revenue, unwanted lawsuits, or worse. The lack of availability causes a ripple effect for all involved: high availability is fundamental to reduce this possibility [2].

High availability dictates that there will always be a response to a request, without loss of data.

3. How it works

Load balancing with HAProxy

As the HAProxy documentation states, HAProxy is a proxy and load balancer for both HTTP and TCP, which provides two essential features:

- It listens for incoming connections on any address/port combinations provided. Once these connections are accepted, and any desired processing performed on them (for example, modifying the packet headers), it will forward them on to a cluster of backend servers using the chosen load balancing algorithm: for example, by distributing the traffic to each backend server in turn, or by distributing the incoming traffic to the server with the least amount of connections at the time.
- It performs health checks by continuously monitoring the state of the backend servers, and only sending traffic to them if they are valid, to make sure no connections are lost.

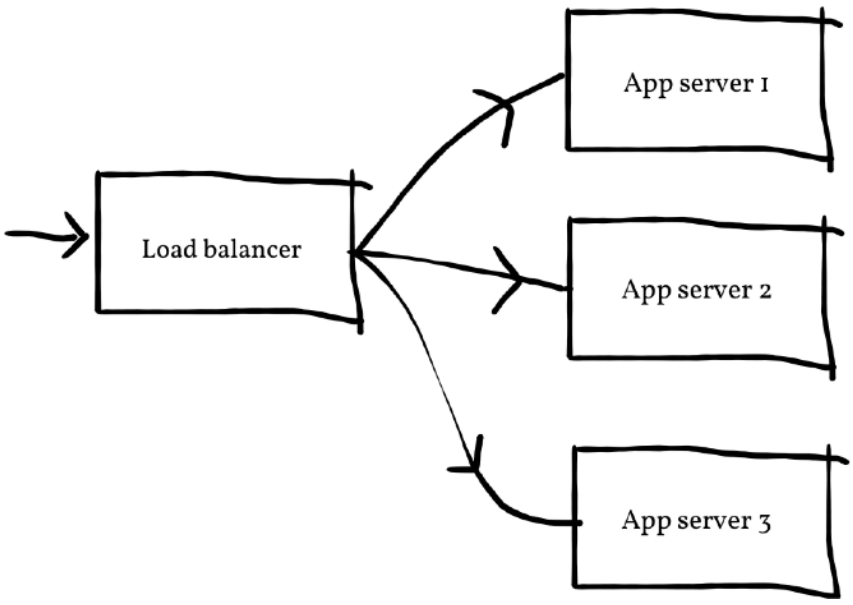
High availability with Keepalived

Keepalived is a Linux implementation of the VRRP protocol for Linux servers. VRRP uses elections to define a master node and a backup node amongst the participating routers, which communicate using the multicast address `224.0.0.18`. The master router is 'assigned' the responsibility of the virtual router, and the virtual router's address. If the backup node fails to receive VRRP messages from the master node after a given period of time, the backup node takes on the responsibility of the virtual address, replacing the master.

In this practical, Keepalived will run alongside the HAProxy nodes performing health checks on HAProxy and on each other using VRRP. The virtual ip in this case will be the address used to access the web application.

4. Load Balancing

The first part of the setup will deal with the configuration of the web servers and of HAProxy, according to the following diagram:



As the diagram suggests, the setup consists of one haproxy instance which will be configured to point to each of the application servers. When an incoming client requests the application's address, the request is sent to the load balancer, which decides which server to pass the request on to.

4.1 Setting up the application servers

The application servers consist of three simple nginx instances serving a simple static web page. Each server is identical to the other with the exception of the main title, in order to distinguish between them when the load balancer is doing its work.

4.2 Configuration of HAProxy

Although HAProxy is relatively simple for small topologies like this one, configuring it properly can be extremely finicky and time consuming. For this reason I decided it would be more efficient to do all testing on localhost before deploying the configuration to the pi cluster. The final outcome can be seen below:

```
global
  log 127.0.0.1 local0
  log 127.0.0.1 local1 notice

defaults
  log global
  option httplog
  option dontlognull
  mode http
  timeout connect 5000
  timeout client 10000
  timeout server 10000

frontend http_in
  bind 0.0.0.0:80
  mode http
  default_backend servers

backend servers
  mode http
  option forwardfor
  balance roundrobin
  server server1 server1:80 check
  server server2 server2:80 check
  server server3 server3:80 check
```

```
  option httpchk GET /
  http-check expect status 200
```

The haproxy configuration file is divided into four main sections.

The first few lines enable global logging and logging for local host, used for troubleshooting later.

In the `defaults` section default values are provided for wait timers and the protocol being used (HTTP) if these are not defined later. The `frontend` and `backend` sections are the most important sections of the configuration file.

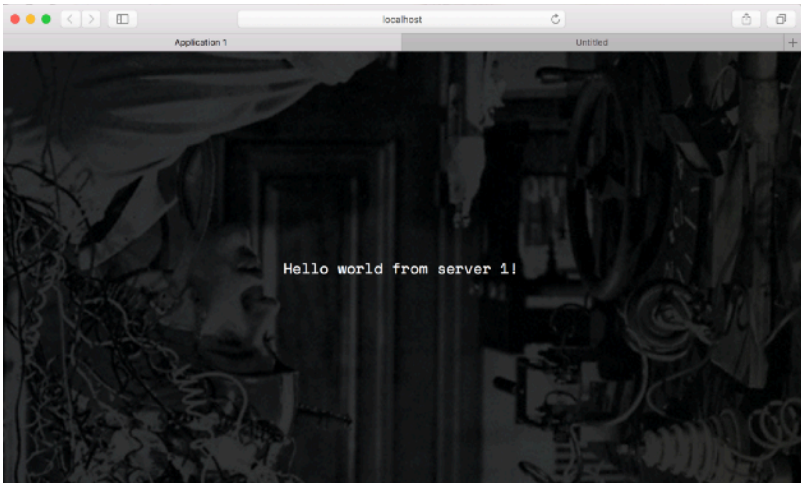
The frontend, which I've called `http_in`, defines the address/port combinations that haproxy will listen on for incoming traffic. Accepted traffic will be forwarded to the backend servers according to whatever load balancing algorithm is chosen. I decided to use the most simple `roundrobin` algorithm, which sequentially distributes incoming requests to the servers specified in the `backend` section.

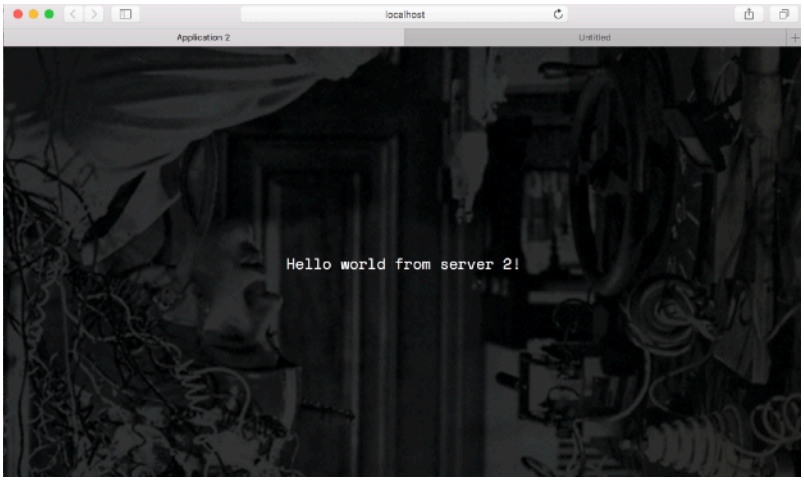
An important function of the backend is to guarantee high availability by performing periodic health checks on the servers, ensuring that incoming traffic is only forwarded on to valid, healthy servers: a user should be oblivious to any state changes in the backend, and ideally their experience should not be disrupted by these whatsoever.

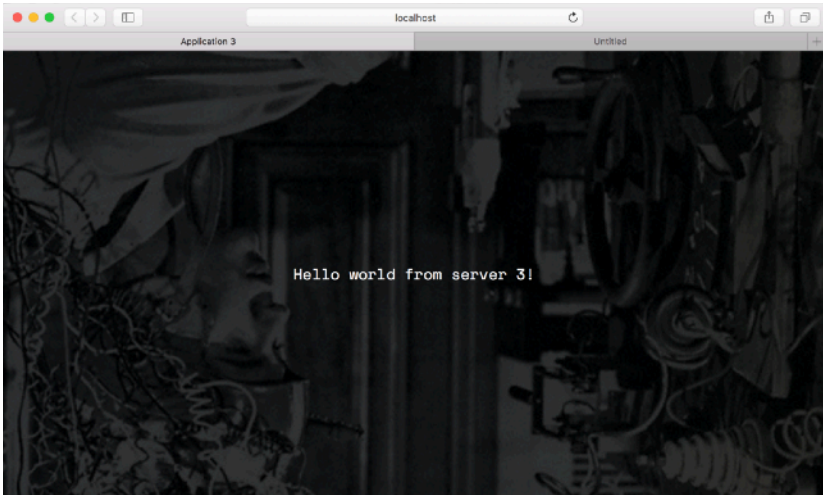
Because HAProxy acts as a reverse proxy, the traffic the servers receive is all shown to be sent by the load balancer. HAProxy provides a few commands to avoid anonymity and retain the identity of the requesting client by updating the IP packets with the client's real address. This is what the '`option forwardfor`' statement is doing.

—

At this point, after starting the HAProxy service and navigating to localhost, everything should be working wonderfully and the load balancer should be distributing HTTP requests across each server in turn on every refresh.







4.3 Building the docker images

For simple deployment I thought it would be nice to move each of these services to their own docker images. This will save on hefty installation time and will allow for easily reproducible copies of the load balancer and servers, which would be helpful if I ever wanted to introduce a second cluster or reuse them in a different context.

To construct the Dockerfiles on my OS I used the `nginx:1.11-alpine` base image for the servers and a `CentOS 7` base image for the load balancers. For testing purposes it's much quicker to start with an haproxy base image but because there are none for the arm64 architecture I chose to build it from source.

The final `docker-compose` file for the first part of the setup looks like this:

```
haproxy:
  build: ./haproxy
```

```
  links:
   - server1
   - server2
   - server3
  ports:
   - “80:80"

server1:
  build: ./nginx
  volumes:
    - ./nginx/server1/:/usr/share/nginx/html/

server2:
  build: ./nginx
  volumes:
    - ./nginx/server2/:/usr/share/nginx/html/

server3:
  build: ./nginx
  volumes:
    - ./nginx/server3/:/usr/share/nginx/html/
```

In the first section, the docker image is built from the `haproxy` folder, which contains a Dockerfile with the same HAProxy configuration as shown before. It exposes port 80 internally (HAProxy is listening on port 80, as defined in the `bind 0.0.0.0:80` statement from above) and publishes this port to the outside world.
The servers are all identical, with an even more simple configuration, and are built from the Dockerfile in the `nginx` folder. The `volumes` statement allows to distinguish between each server by using unique index.html pages for each of the servers.

With a `docker-compose up` the configuration is working exactly like it was before but is now using more easily reproducible docker containers instead.

4.4 Setting up the pi cluster

Because the ip configuration of the raspberry pis was changing quite often, to speed up the process it made sense to set up an ad-hoc ethernet network which also had internet connectivity via the wireless card on my laptop. With this in place all of the devices in the cluster could be accessed via SSH and statically assigned addresses within the local network.

4.5 Ansible

I looked into Ansible as a solution to the tedium of executing the same tasks five times over for each node in the cluster.
The main tasks I had to accomplish were:

- Updating & upgrading Ubuntu
- Installing software dependencies like Docker
- Starting docker
- Building and launching the docker images
- Installing and configuring Keepalived (for later)

Installing & configuring Ansible

Ansible uses SSH to run these tasks from each device simultaneously. To avoid having to type in passwords every time, I copied the contents of my public SSH key `~/.ssh/id_rsa.pub` to a file in each device called `~/.ssh/authorized_keys`. SSH uses a private and public key pair for authentication. By copying my public key to the `authorized_keys` file of each device, as long as I can prove ownership of the corresponding private key and the key pair can be verified, passwordless access to the devices is allowed.

Ansible is dependent on Python which must first be installed on each device before it can be used:
`sudo apt-get install -y python`

Lastly, Ansible will need sudo permissions to properly execute most of the commands. This could be done by modifying the `visudo` file and adding an extra line for the user, `` `ubuntu` `` in my case. Depending on the placement of this new line in the file, it could easily be overwritten by other lines in the file. It would be best practice to make a new file, called 'overrides' or 'myOverrides' in the `sudoers.d` directory:
`sudo visudo -f /etc/sudoers.d/myOverrides`

and adding the line:
`ubuntu ALL=(ALL:ALL) ALL`

To verify connectivity, I used the ansible `` `ping` `` module. This module issues a `` `ping` `` which attempts to connect to each device ensuring that python is enabled, and returns a 'pong' if successful. To test for all devices:

```
# Test all devices can be logged on to and
execute python with json lib.

ansible -m ping all
```

```
10.42.0.175 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
10.42.0.38 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
10.42.0.152 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
10.42.0.73 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
10.42.0.135 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Ansible Playbooks

Ansible's docs describe playbooks as the "instruction manuals" to manage configuration and deployment to remote machines. Playbooks simply create a mapping between certain "roles", or lists of tasks, to certain devices in the network.

For the tasks described above I ended up with a total of five playbooks:

Install (install.yml)

Runs the `install` role on all devices, which updates and upgrades Ubuntu, installs docker and starts the docker daemon.

Build images (buildimages.yml)

Runs the `docker_server` and `docker_balancer` roles. The `docker_balancer` role builds the haproxy image on each load balancer and then runs the container on port 80. The `docker_server` role builds the image for the nginx servers, passing in a parameter for the mount paths (as seen before in the `docker-compose` file) and runs the containers on the server nodes.

Reboot servers (reboot.yml)

Runs the `reboot` role, which simply reboots all the devices in the cluster.

Shutdown Servers (servershutdown.yml)

Runs the `shutdown` role, which shuts down all the devices in the cluster.

Keepalived (keepalived.yml)

This playbook is used later to install Keepalived on the load balancers, and copy over the correct configuration files needed for Keepalived. It calls the `keepalived` role.

Running the playbook with the command `ansible-playbook -i inventories/ha.ini playbooks/install.yml` the output looks like this:

```
PLAY [Install dependencies] ****************************************************

TASK [setup] *******************************************************************
ok: [10.42.0.175]
ok: [10.42.0.135]
ok: [10.42.0.73]
ok: [10.42.0.38]
ok: [10.42.0.152]

TASK [install : Update and upgrade Ubuntu] *************************************
changed: [10.42.0.38]
 [WARNING]: Consider using apt module rather than running apt-get

changed: [10.42.0.152]
changed: [10.42.0.73]
changed: [10.42.0.135]
changed: [10.42.0.175]

TASK [install : Install docker] ************************************************
changed: [10.42.0.175]
changed: [10.42.0.73]
changed: [10.42.0.152]
changed: [10.42.0.38]
changed: [10.42.0.135]

TASK [install : Start docker daemon] *******************************************
```

And for the docker images playbook, `ansible-playbook -i inventories/ha.ini playbooks/buildimages.yml`:

```
TASK [docker_server : Run nginx container] ************************************
changed: [10.42.0.175]

PLAY [Build and run nginx docker image] ***************************************

TASK [setup] ******************************************************************
ok: [10.42.0.135]

TASK [docker_server : Make nginx dir] *****************************************
changed: [10.42.0.135]

TASK [docker_server : Copy docker files] **************************************
changed: [10.42.0.135]

TASK [docker_server : Build nginx image] **************************************
changed: [10.42.0.135]

TASK [docker_server : debug] **************************************************
ok: [10.42.0.135] => {
    "mount_path": "/home/ubuntu/nginx/server3"
}

TASK [docker_server : Run nginx container] ************************************
changed: [10.42.0.135]

PLAY RECAP ********************************************************************
10.42.0.135                : ok=6    changed=4    unreachable=0    failed=0
10.42.0.152                : ok=6    changed=4    unreachable=0    failed=0
10.42.0.175                : ok=6    changed=4    unreachable=0    failed=0
```

Test configuration on the load balancer by doing a curl to localhost:

```
</html>ubuntu@pine64:~$ curl http://localhost
<!DOCTYPE html>
<html>
<head>
        <title>Application 1</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 1!
        </div>
</body>

</html>ubuntu@pine64:~$ curl http://localhost
<!DOCTYPE html>
<html>
<head>
        <title>Application 2</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 2!
        </div>
</body>

</html>ubuntu@pine64:~$ curl http://localhost
<!DOCTYPE html>
<html>
<head>
        <title>Application 3</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 3!
        </div>
</body>

</html>ubuntu@pine64:~$
```

5. High Availability

Although the current setup contains redundancy in the form of multiple application servers, the load balancer still acts as a single point of failure: if it had to go down, regardless of the number of application servers, the application would still not be accessible to the outside world.

The second part of the setup introduces a second load balancer to avoid that scenario. The load balancers will communicate with each other and perform health checks using Keepalived, which uses the VRRP protocol to allow the nodes to talk to each other by sending and receiving VRRP multicast packets.

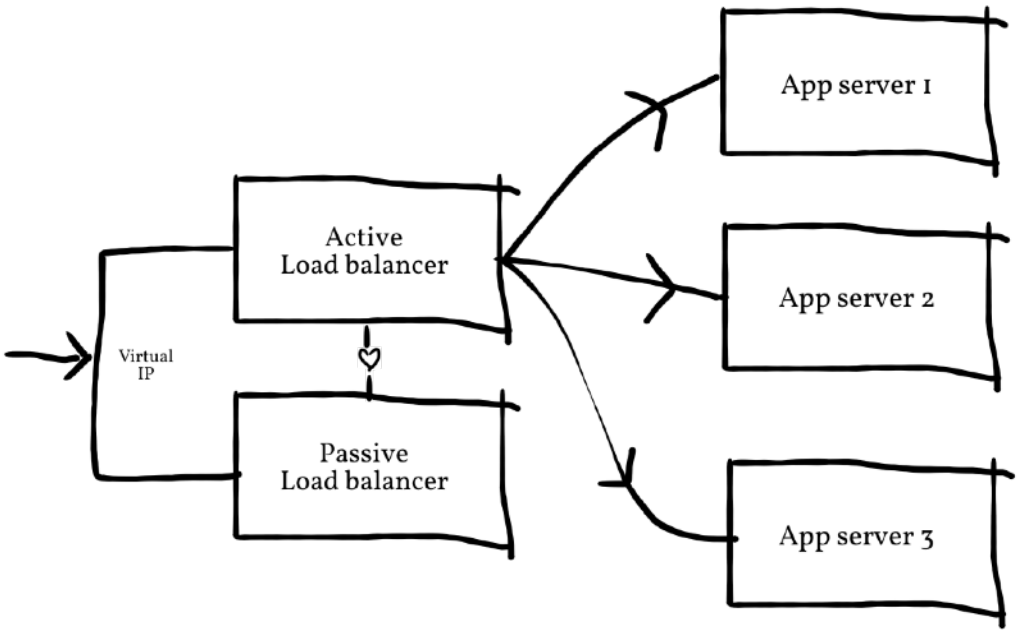
Keepalived will run on the load balancing servers, outside of the containers, and perform continuous health checks on the docker/haproxy process. If the master instance were to go down, the backup instance would pick up on this and immediately take its place as master.

Because docker processes are run in isolation on the host machine, the haproxy instance running inside the container can be accessed from the outside. This means that Keepalived can perform health checks on HAProxy itself (not docker) without sitting inside the container. If Keepalived had to be placed inside the container, it would have to be assigned a virtual IP within the docker subnet, making the virtual IP non-accessible to the outside.

```
root     25597     2  0 10:51 ?        00:00:00 [kworker/u8:2]
root     26671     2  0 Apr09 ?        00:00:00 [kworker/3:1]
root     26832     2  0 10:56 ?        00:00:00 [kworker/u8:1]
root     26992   841  0 Apr09 ?        00:00:00 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 80 -container-ip
root     26998   850  0 Apr09 ?        00:00:00 containerd-shim 67aad0561cdb7e1e8953d6d75101293b260e158230fbc074ae68034333617
root     27014 26998  0 Apr09 ?        00:09:02 ./haproxy -f /usr/local/haproxy/haproxy.cfg
root     27772   838  0 11:00 ?        00:00:00 sshd: ubuntu [priv]
ubuntu   27781 27772  0 11:00 ?        00:00:00 sshd: ubuntu@pts/0
ubuntu   27782 27781  0 11:00 pts/0    00:00:00 -bash
root     27808   512  0 11:00 ?        00:00:00 sleep 5
ubuntu   27812 27782  0 11:00 pts/0    00:00:00 ps -ef
root     28906     1  0 Apr09 ?        00:00:26 keepalived --vrrp
root     28907 28906  0 Apr09 ?        00:06:13 keepalived --vrrp
```

Fig. Running the `ps -ef` command

The diagram below illustrates the passive/active or master/backup roles of the load balancers. This time, the address the client uses to access the web application corresponds to the virtual ip address Keepalived has configured. When the client requests the application, the active load balancer will process the request and pass it on to whichever application server it chooses. Again, as mentioned above, if the active load balancer were to go down at any stage the passive load balancer would immediately switch over to replace it.



5.1 Configuring Keepalived

The Ansible playbook from earlier is used to install Keepalived with the `sudo apt-get install -y keepalived` command and to copy over the configuration files, seen below:

```
# Active load balancer configuration
vrrp_script check_haproxy {
        script "killall -0 haproxy"
        interval 2
        weight 2
}

vrrp_instance haproxy {
        interface eth0
        state MASTER
        virtual_router_id 51
        priority 101
        virtual_ipaddress {
            10.42.0.240
        }
        track_script {
            check_haproxy
        }
}

# Passive load balancer configuration
vrrp_script check_haproxy {
```

```
        script "killall -0 haproxy"
        interval 2
        weight 2
}

vrrp_instance haproxy {
        interface eth0
        state BACKUP
        virtual_router_id 51
        priority 100
        virtual_ipaddress {
            10.42.0.240
        }
        track_script {
            check_haproxy
        }
}
```

Ensure that `killall` is available by installing the psmisc toolkit:
`sudo apt-get install psmisc`

Both files start with a vrrp_script which performs the health checks. The script used above contains just one line, `"killall -0 haproxy"`. The `killall -0` sends a signal to the `haproxy` process to check whether it is running or not. If the script returns an exit status other than 0, the process is dead.
The weight parameter reduces the priority by 2 on “fall”. This means that when the script returns a non-zero code, the node’s priority (if it is the master) will drop to 99 and the backup node, having a priority of 100, will become the master. The device’s state will also be changed to FAULT and the virtual IP removed from its interface. It will then finally stop sending multicast packets.

Both configuration files are pretty much identical, with the exception of the `state` and `priority` parameters. Naturally, the master’s state will be MASTER and the backup’s state will be BACKUP. The master will always have a higher priority than the backup node, and because the weight specified is 2, as per the process described before the difference must always be less than 2 for the failover mechanism to work as expected.

5.2 Troubleshooting & Failover

For troubleshooting, I used the following commands:
- `tail -f /var/log/syslog`
- `sudo tcpdump -vvv -n -i eth0 host 224.0.0.18`

The first simply prints out the system log. The `tcpdump` command sniffs packets to and from the VRRP multicast address on the given interface (http://www.tcpdump.org/tcpdump_man.html is a useful link to see all the possible flags and parameters that can be used with the tcpdump command).
The combination of these two commands give a good idea about the state of the nodes and whether they are communicating or not.

An example of the tcpdump is shown below:

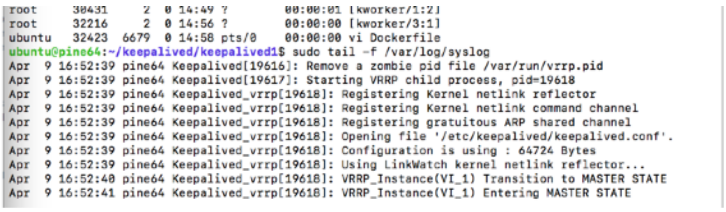
```
ubuntu@pine64:~/keepalived/keepalived2$ sudo tcpdump -vvv -n -i eth0 host 224.0.0.18
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
16:46:04.767990 IP (tos 0xc0, ttl 255, id 585, offset 0, flags [none], proto VRRP (112), length 40)
    10.42.0.73 > 224.0.0.18: vrrp 10.42.0.73 > 224.0.0.18: VRRPv2, Advertisement, vrid 51, prio 101, authtype none, intvl 1s, length 20, addrs: 10.42.0.240
16:46:05.768223 IP (tos 0xc0, ttl 255, id 586, offset 0, flags [none], proto VRRP (112), length 40)
    10.42.0.73 > 224.0.0.18: vrrp 10.42.0.73 > 224.0.0.18: VRRPv2, Advertisement, vrid 51, prio 101, authtype none, intvl 1s, length 20, addrs: 10.42.0.240
16:46:06.768493 IP (tos 0xc0, ttl 255, id 587, offset 0, flags [none], proto VRRP (112), length 40)
    10.42.0.73 > 224.0.0.18: vrrp 10.42.0.73 > 224.0.0.18: VRRPv2, Advertisement, vrid 51, prio 101, authtype none, intvl 1s, length 20, addrs: 10.42.0.240
```

This particular screenshot shows that only one of the load balancers is sending multicast packets. This seemed to be a quite a common problem and usually one related to firewalls.
In my case executing the commands below from the load balancers solved the problem:

- `iptables -A INPUT -i eth0 -d 224.0.0.0/8 -j ACCEPT`
- `iptables -A INPUT -p vrrp -i eth0 -j ACCEPT`

The first command allows all incoming multicast traffic on the eth0 interface. The second allows all incoming VRRP traffic (`-p` specifies the protocol).
IP forwarding must also be enabled, if it is not already. In the `/etc/sysctl.conf` file:
`net.ipv4.ip_forward = 1`

When the devices are properly configured, the `syslog` gives a great idea to what is happening in the background:



The keepalived process is started and the master node enters MASTER state:

```
@pine64:~/keepalived/keepalived1$ sudo tail -f /var/log/syslog
 16:52:39 pine64 Keepalived[19616]: Remove a zombie pid file /var/run/vrrp.pid
 16:52:39 pine64 Keepalived[19617]: Starting VRRP child process, pid=19618
 16:52:39 pine64 Keepalived_vrrp[19618]: Registering Kernel netlink reflector
 16:52:39 pine64 Keepalived_vrrp[19618]: Registering Kernel netlink command channel
 16:52:39 pine64 Keepalived_vrrp[19618]: Registering gratuitous ARP shared channel
 16:52:39 pine64 Keepalived_vrrp[19618]: Opening file '/etc/keepalived/keepalived.conf'.
 16:52:39 pine64 Keepalived_vrrp[19618]: Configuration is using : 64724 Bytes
 16:52:39 pine64 Keepalived_vrrp[19618]: Using LinkWatch kernel netlink reflector...
 16:52:40 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Transition to MASTER STATE
 16:52:41 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Entering MASTER STATE
```

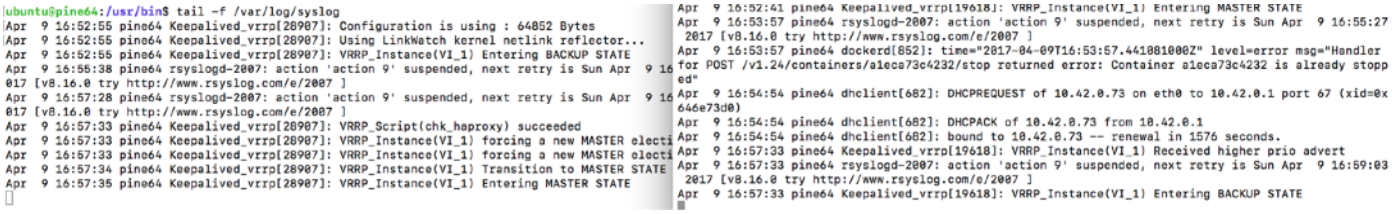
The backup node enters BACKUP state:

```
pine64:~/keepalived/keepalived2$ tail -f /var/log/syslog
16:52:55 pine64 Keepalived[28905]: Remove a zombie pid file /var/run/keepalived.pid
16:52:55 pine64 Keepalived[28905]: Remove a zombie pid file /var/run/vrrp.pid
16:52:55 pine64 Keepalived[28906]: Starting VRRP child process, pid=28907
16:52:55 pine64 Keepalived_vrrp[28907]: Registering Kernel netlink reflector
16:52:55 pine64 Keepalived_vrrp[28907]: Registering Kernel netlink command channel
16:52:55 pine64 Keepalived_vrrp[28907]: Registering gratuitous ARP shared channel
16:52:55 pine64 Keepalived_vrrp[28907]: Opening file '/etc/keepalived/keepalived.conf'

16:52:55 pine64 Keepalived_vrrp[28907]: Configuration is using : 64852 Bytes
16:52:55 pine64 Keepalived_vrrp[28907]: Using LinkWatch kernel netlink reflector...
16:52:55 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) Entering BACKUP STATE
```

The two nodes are communicating and Keepalived is working as expected. Curling the virtual ip shows the same output as before:

```
ubuntu@pine64:~$ curl http://10.42.0.240
<!DOCTYPE html>
<html>
<head>
        <title>Application 3</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 3!
        </div>
</body>

</html>ubuntu@pine64:~$
ubuntu@pine64:~$ curl http://10.42.0.240
<!DOCTYPE html>
<html>
<head>
        <title>Application 2</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 2!
        </div>
</body>

</html>ubuntu@pine64:~$ curl http://10.42.0.240
<!DOCTYPE html>
<html>
<head>
        <title>Application 1</title>
        <link href="https://fonts.googleapis.com/css?family=Space+Mono" rel="stylesheet">
        <link href="style.css" rel="stylesheet">
</head>

<body>
        <div class="page">
                Hello world from server 1!
        </div>
</body>

</html>ubuntu@pine64:~$
```

To test if the failover mechanism is working, I could either shut down the server, stop docker or the kill the HAProxy instance.
The example below shows what happens when the HAProxy instance is killed:



The master node's health check to haproxy fails. A new election is forced and the backup node enters master state. The old master enters backup state.

```
pine64:/usr/bin$ tail -f /var/log/syslog
16:52:55 pine64 Keepalived_vrrp[28907]: Configuration is using : 64852 Bytes
16:52:55 pine64 Keepalived_vrrp[28907]: Using LinkWatch kernel netlink reflector...
16:52:55 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) Entering BACKUP STATE
16:55:38 pine64 rsyslogd-2007: action 'action 9' suspended, next retry is Sun Apr  9 16:57:08 2
.16.0 try http://www.rsyslog.com/e/2007 ]
16:57:28 pine64 rsyslogd-2007: action 'action 9' suspended, next retry is Sun Apr  9 16:58:58 2
.16.0 try http://www.rsyslog.com/e/2007 ]
16:57:33 pine64 Keepalived_vrrp[28907]: VRRP_Script(chk_haproxy) succeeded
16:57:33 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) forcing a new MASTER election
16:57:33 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) forcing a new MASTER election
16:57:34 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) Transition to MASTER STATE
16:57:35 pine64 Keepalived_vrrp[28907]: VRRP_Instance(VI_1) Entering MASTER STATE
```

```
ubuntu@pine64:~/keepalived/keepalived1$ sudo tail -f /var/log/syslog
Apr  9 16:52:40 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Transition to MASTER STATE
Apr  9 16:52:41 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Entering MASTER STATE
Apr  9 16:53:57 pine64 rsyslogd-2007: action 'action 9' suspended, next retry is Sun Apr  9 16:55:27
 2017 [v8.16.0 try http://www.rsyslog.com/e/2007 ]
Apr  9 16:53:57 pine64 dockerd[852]: time="2017-04-09T16:53:57.441081000Z" level=error msg="Handler
for POST /v1.24/containers/a1eca73c4232/stop returned error: Container a1eca73c4232 is already stopp
ed"
Apr  9 16:54:54 pine64 dhclient[682]: DHCPREQUEST of 10.42.0.73 on eth0 to 10.42.0.1 port 67 (xid=0x
646e73d0)
Apr  9 16:54:54 pine64 dhclient[682]: DHCPACK of 10.42.0.73 from 10.42.0.1
Apr  9 16:54:54 pine64 dhclient[682]: bound to 10.42.0.73 -- renewal in 1576 seconds.
Apr  9 16:57:33 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Received higher prio advert
Apr  9 16:57:33 pine64 rsyslogd-2007: action 'action 9' suspended, next retry is Sun Apr  9 16:59:03
 2017 [v8.16.0 try http://www.rsyslog.com/e/2007 ]
Apr  9 16:57:33 pine64 Keepalived_vrrp[19618]: VRRP_Instance(VI_1) Entering BACKUP STATE
```

Conclusion

This project has shown how it is possible to implement, with proven open source software:

- Reusability, thanks to Docker and Ansible
- Redundancy, by introducing redundant servers and a passive/active load balancer configuration into the initial setup
- Reliability and availability, given by the combination of Keepalived & HAProxy and the healthchecks and monitoring these services perform.

All of these concepts and tools provide the first step towards provisioning a production ready, high availability environment, providing the end user with a much more stable and consistent application.

Resources

The docker files, ansible playbooks, html pages and all the configuration files used in this project can be found on my github at https://github.com/dimitraz/high-availability-poc/.

The base images used are available on docker hub from:

- `bobsense/nginx-arm64`
- `project31/aarch64-centos:7`

References

[1] R. Strickland, in Cassandra High Availability, Birmingham: Packt Publishing, 2014.

[2] Oracle (n.d.). High Availability [Online]. Available: https://docs.oracle.com/cd/B19306_01/server.102/b14210/overview.htm. [Accessed: 4- Mar- 2017]

Bibliography

[1] Mitchell Anicas (2017). An Introduction to HAProxy and Load Balancing Concepts [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts [Accessed: 10- Feb- 2017]

[2] Erika Heidi (2016). What is High Availability? [Online]. Available: https://www.digitalocean.com/community/tutorials/what-is-high-availability [Accessed: 10- Feb- 2017]

[3] Servers for hackers (2014). Load Balancing with HAProxy [Online]. Available: https://serversforhackers.com/load-balancing-with-haproxy [Accessed: 19- Feb- 2017]

[4] HAProxy Documentation (2017). Starter Guide [Online]. Available: http://cbonte.github.io/haproxy-dconv/1.7/intro.html [Accessed: 4- Mar- 2017]

[5] HAProxy Documentation (2017). Configuration Manual [Online]. Available: http://cbonte.github.io/haproxy-dconv/1.7/configuration.html [Accessed: 4- Mar- 2017]

[6] Docker Documentation (2017). Docker Compose [Online]. Available: https://docs.docker.com/compose/ [Accessed: 10- Mar- 2017]

[7] Ansible Docs (2017). Homepage [Online]. Available: http://docs.ansible.com/ansible/index.html [Accessed: 20- Mar- 2017]

[8] Cisco (n. d.). What is VRRP? [Online]. Available: http://www.cisco.com/c/en/us/support/docs/security/vpn-3000-series-concentrators/7210-vrrp.html [Accessed: 24- Mar- 2017]

[9] Oracle (2017). Installing and Configuring Keepalived [Online]. Available: https://docs.oracle.com/cd/E37670_01/E41138/html/section_ksr_psb_nr.html [Accessed: 24- Mar- 2017]

[10] Keepalived (n. d.). Homepage [Online]. Available: http://www.keepalived.org/documentation.html [Accessed: 01- Apr- 2017]

[11] Tobias Brunner (2013). Keepalived Check and Notify Scripts [Online]. Available: https://tobrunet.ch/2013/07/keepalived-check-and-notify-scripts/ [Accessed: 01- Apr- 2017]

[12] Ubuntu Documentation (2017). Iptables how to [Online]. Available: https://help.ubuntu.com/community/IptablesHowTo?action=show&redirect=Iptables [Accessed: 07- Apr- 2017]