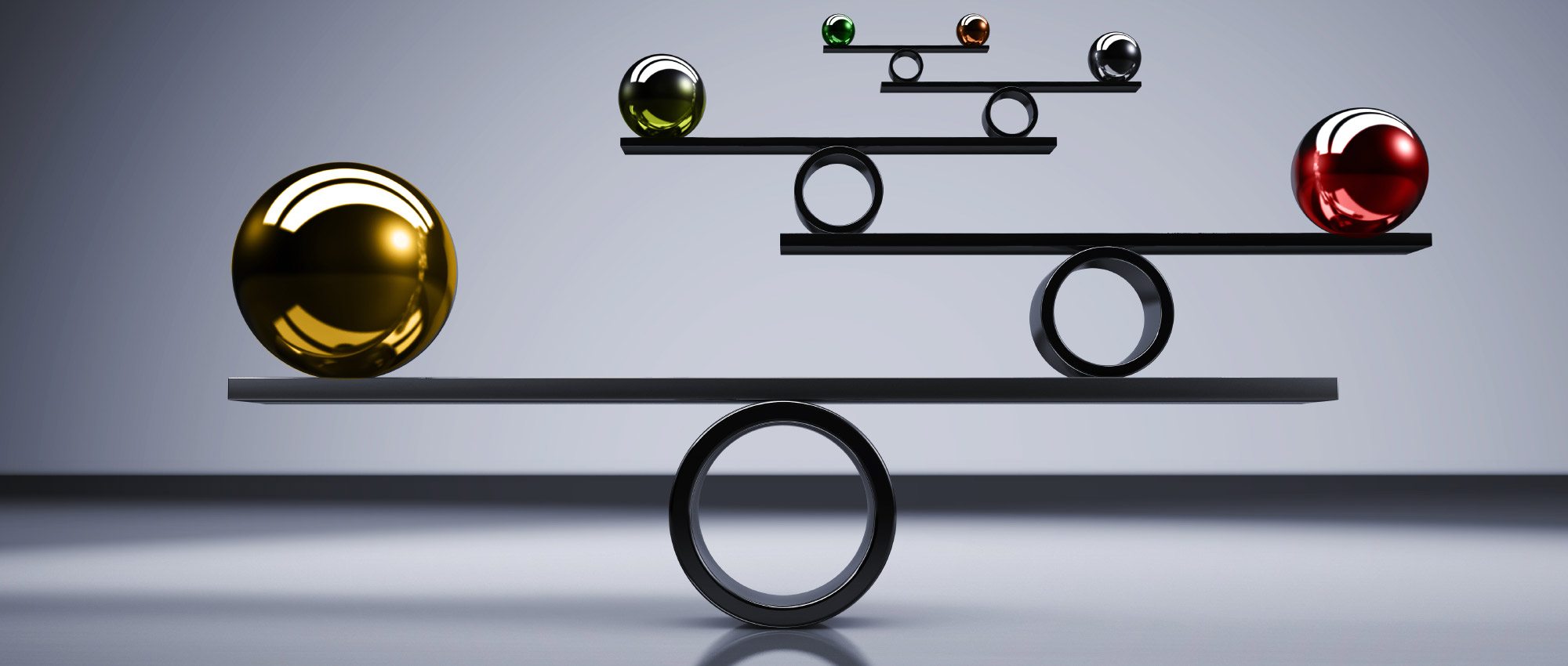
**Cloud Load Balancing**



By Jenny Kapul & Andrey Semetchkov

Guided by Shay Vargaftik

12/11/18

Contents

[Load Balancing – Introduction 4](#_Toc529448156)

[Algorithms 4](#_Toc529448157)

[Project Overview 6](#_Toc529448158)

[Phase 1 - Research and information gathering 7](#_Toc529448159)

[Figure 1 – Netflix Eurika vs Haroxy 7](#_Toc529448160)

[Phase 2 – Setup of the POC load-balanced system 9](#_Toc529448161)

[Client 9](#_Toc529448162)

[Server 10](#_Toc529448163)

[Phase 3 – Persistent idle implementation in HaProxy 12](#_Toc529448164)

[Why HaProxy? 12](#_Toc529448165)

[Figure 2 - HaProxy Popularity 12](#_Toc529448166)

[Persistent idle implementation 13](#_Toc529448167)

[Intro 13](#_Toc529448168)

[EBtree Data Structure 13](#_Toc529448169)

[Figure 3 – Binary Search Tree 14](#_Toc529448170)

[Modules 15](#_Toc529448171)

[HaProxy Load Balancing Algorithms API 16](#_Toc529448172)

[Important implementation details 19](#_Toc529448173)

[Config Parsing module 20](#_Toc529448174)

[How To Compile, Install and Run HaProxy From Source 23](#_Toc529448175)

[Service init script 24](#_Toc529448176)

[Sample Config file 29](#_Toc529448177)

[Project's Final Config File haproxy.cfg 30](#_Toc529448178)

[Run 31](#_Toc529448179)

[Debug 31](#_Toc529448180)

[Phase 4 – The Scaled-Up System & Networking 32](#_Toc529448181)

[Figure 4 – System Architecture 32](#_Toc529448182)

[Final System setup 33](#_Toc529448183)

[Load-Balancer 33](#_Toc529448184)

[Worker Servers 33](#_Toc529448185)

[Client 34](#_Toc529448186)

[Figure 5 – Results Data Table Format 34](#_Toc529448187)

[Phase 5 – Analysis of results and future suggestions 35](#_Toc529448188)

[Analysis 35](#_Toc529448189)

[Figure 6 – PI Best vs. Worst Run 35](#_Toc529448190)

[Figure 7 – LC Best vs. Worst run 36](#_Toc529448191)

[Figure 8 – RR Best vs. Worst Run 36](#_Toc529448192)

[Figure 9 – Worst Run 37](#_Toc529448193)

[Figure 10 – Best Run 38](#_Toc529448194)

[Suggestions For Future Analysis 39](#_Toc529448195)

# Load Balancing – Introduction

## Algorithms

In computing, load balancingimproves the distribution of workloads across multiple computing resources, such as computers, a computer cluster, network links, central processing units, or disk drives. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. Load balancing usually involves dedicated software or hardware, such as a multilayer switch or a Domain Name System (DNS) server process.

There are many different load-balancing algorithms of static and dynamic nature. Here are the commonly practiced:

* Round-Robin – one of the most basic static algorithms where the load-balancer delegates the tasks to the servers in round-robin fashion – without the consideration of the load of the tasks and the differences between a server’s performance. Such a choice is optimal when both servers are homogenous with deterministic performance and, the workload is homogenous and deterministic. Unfortunately, this is rarely the case in practice. Nevertheless, the main advantage of this simplistic approach is the fact that it does not require any computational or network resources and may perform well is homogenous systems under moderate load.
* Join the Shortest Queue (JSQ) – each server has a queue of tasks that have been delegated to it. This algorithm takes into consideration the state of the system before deciding which server has the shortest queue – hence the overhead may be considerable. In addition, the shortest queue does not necessarily mean the shortest execution time.
* JSQ (d) – same as JSQ but we poll only d different servers before each decision. This variation reduces the overhead on the expense of accuracy (choosing the actual shortest queue is not guaranteed).
* JSQ (d, m) – same as JSQ (d) but we also remember the m shortest queues from the previous check. We sample and decide among the previous m and the additional new d queues (d>=m>=1). This variation uses some additional memory and computation to save the state and make decisions but it also may increase accuracy, especially at high loads .
* Join the Idle Queue (JIQ) – we define an idle state of a server to be when the size of its queue is bellow a specific threshold (for example it could be an empty queue). We assume that the system’s initial state is when all servers are idle. Each time a new task is delegated, the load-balancer delegates it to one of the idle servers. If no idle server available – the task is delegated to a random server.
* **Persistent Idle** (PI) – PI exploits only idleness signals, similarly to JIQ. However, whereas in JIQ a random server is chosen when no idle server is available, in PI, the task is delegated to the server that the previous task was delegated to (i.e., the last-idle server).

The purpose of this project is to test the PI algorithm and compare its performance to other, known algorithms using a POC system.

Project Overview

In our awesome project we had to deal with 5 major challenges and a lot of system debugging. First, information was gathered regarding load balancing, service deploy from source, VM networking and python client server development. Next, a debug system was built with one client, one load-balancer and 2 servers. Also, networking capabilities were established and tested. Afterwards, HAPROXY version 1.8 was forked in Github using git and studied. Code analysis was performed and the load balancing flow was understood. Meaning we understood all the modules relevant too load balancing and that lead to them from main. In addition, the interactions between modules was understood. Then, we started integrating the PI algorithm into HAProxy by editing the source code. Specifically, the Least Connections algorithm was edited to become the PI module. When code debug was finished, we built the system where we run 5 web servers, client and proxy. Finally, the evaluation was performed as the final system worked after all the debugging and testing. The results were: 10 runs of all 3 algorithms different with 3 strong servers and 2 weaker servers. The results were graphed and analyzed.

To deal with each challenge at a time we decided to divide the project into 5 main phases:

**Phase 1 – research and information gathering**

**Phase 2 – Setup a debug network of the lightest load-balanced system possible – one client, one load-balancer, 2 servers**

**Phase 3 – Persistent idle implementation in HAProxy**

**Phase 4 – POC scaled system one client,**

**one load-balancer, 5 servers**

**Phase 5 – Analysis of simulation results and conclusions**

# Phase 1 - Research and information gathering

Since the purpose of the project wasn’t creating a load balancing software but rather extending it with a specific algorithm, we decided to find an open source load-balancing software which suits us the most. The prime criterion was our ability to easily add the PI algorithm as part of the existing software.

After online research, 2 final candidates were chosen for a more thorough comparison – HaProxy and Netflix Eurika.

## Figure 1 – Netflix Eurika vs Haroxy

Another issue we needed to dive into was the implementation of a lightweight web server. The network latencies might skew results and hurt heterogeneity of the network. To minimize the effects of the network we decided on a workload that primarily requires CPU resources (rather then I/O operations during which the CPU is idle). Each request to a server contains a long number >1000000000, and the server's task is to calculate the closest prime number such that . We intentionally did not optimize this calculation to make the task CPU bound.

The goal was to try to create a system that is not ideal and does not hold the assumptions of server or workload homogeneity. To mimic a heterogonous system, we could not use the same behavior for all servers. The problem was that all of them receive the same request statistically, and we cannot determine in advance which requests will be delegated to a specific server. The simplest solution we came up with was to determine the number of digits from the requested number the server takes into consideration. For example, assume we have 2 servers where server1 is faster than server2. Then, we will configure server1 to calculate for , while server2 will calculate for . In this example, server1 uses 6 digits from the request while server2 uses 9 digits. Statistically, the more digits we use the longer the calculation will take. The fact that prime numbers aren’t evenly distributed, helps us mimic the heterogeneity of the workload.



# Phase 2 – Setup of the POC load-balanced system

We decided to start working with virtual machines using Virtual Box. We have created the following setup – 2 virtual machines and the host:

1. Client – to test a load-balanced system, at least one client who sends the requests is required. We chose the host to be responsible for that part.
2. Load-Balancer – an Ubuntu based virtual machine. Described in detail in the next section.
3. Servers - an Ubuntu based virtual machine. We configured the VM with several cores, so each server would run on a separate core and left an additional core for the OS. In our case – total of 6 cores (5 servers).

## Client

A Python script was written from scratch. The script has 1 parameter so the command usage is as follows:

python client.py <output\_file\_name>

NUM\_OF\_THREADS – defines the number of parallel threads generating requests.

REQ\_PER\_THREAD - defines the number of sequential requests each thread generates.

The flow of the script is very simple – threads initialized and joined immediately after. The rest of the logic is converting the responses from all of the threads into a single csv file.

The most significant functions of the script:

"""

Creates NUM\_OF\_THREADS threads in the system when each thread is myThread

"""

**def** initializeWorkers**()**

"""

Sends REQ\_PER\_THREAD requests using httpGET() function. the parameter for each request is a randomly generated number

When the response received, it is stored in a list for further analysis

"""

**def** myThread**(**id**):**

"""

Sends HTTP get request to "http://192.168.56.254/isPrime/" URL with randNum parameter

"""

**def** httpGET**(**randNum**)**

## Server

We used web.py package for this module.

The script has 3 parameters so the command usage is as follows:

python server.py <#digits> <port\_number> <server\_name>

class Request – struct that holds all the data describing a single request.

class Result – struct that holds all the data describing a single result. Has a few methods to calculate different time gaps. This class is serialized into JSON format by overloading the \_\_str\_\_ method.

The main thread listens on a port specified by port\_number. Each received request, is serialized into Request class instance and enqueued into qin )asynchronous queue, from python Queue package).

The “worker” thread (which initialized once, when first request is received) pops requests from qin one at a time. For each request, closestPrime is called and Result class instance initialized with all the relevant data and enqueued into qout. The results from qout are returned in JSON format as HTTP response.

Each URL that the server supports must be defined in the urls tuple. Each url defined by two sequential tuple members – the first one the url string and the second the name of the class that implements GET and POST methods for that url (in our script classes isPrime and get\_post2).

The main function of the web application is run method of MyApplication class-

**class** **MyApplication(**web**.**application**):**

**def** run**(**self**,** port**=**int**(**port**),** middleware**):**

func **=** self**.**wsgifunc**(** middleware**)**

**return** web**.**httpserver**.**runsimple**(**func**,** **(**'0.0.0.0'**,** port**))**

we basically used it as it defined in the manuals of web.py package.

http://webpy.org/docs/0.3/tutorial

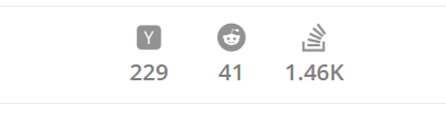


Phase 3 – Persistent idle implementation in HaProxy

## Why HaProxy?



* Well documented
* Very popular
* Reliable
* High Performance
* Variety of implemented LB algorithms



### Figure 2 - HaProxy Popularity

## Persistent idle implementation

### Intro

Git history available here  
<https://github.com/AndreySemechkov/HAproxy-PI>

In general in HAPROXY the load balancing algorithms are implemented in a common API with set functionality needed.  
Practically speaking our main goal is to implement the load Balancing API (see API section) used in HAPROXY implementing PI algorithm. Furthermore, to perform proper integration and call the persistent idle functions of load balancing API used in HAPROXY the config parser was edited. The module extracts the parameters needed for PI from the configuration file ( called haproxy.cfg).  
Lastly, we edited the structs needed to support the variables we need to use in our implementation between API calls.

### EBtree Data Structure

In computer science, an elastic binary tree (or EB tree or EBtree) is a binary search tree specially optimized to very frequently store, retrieve and delete discrete integer or binary data without having to deal with memory allocation. It is particularly well suited for operating system schedulers where fast time-ordering and priority-ordering are strong requirements. Insertion and lookups are performed in O(log n) while removal is done in O(1). The tree is not balanced but its height is bounded by the type of data it stores. Ordered duplicate entries are also natively supported.

In HAPROXY project EBtree is used with integer node keys which represent the weight of the server associated with the node. Some algorithms have a custom smart weights system, we did not do that. For the PI API implementation, we used the number of the servers **ACTIVE** HTTP connections only as the key. Thus, the connections task queue of each server was the same as the nodes key. Of course, there is a simple macro that converts each given node into the appropriate server struct that holds the whole servers state.



### Figure 3 – Binary Search Tree

### Modules

New Modules  
src/lb\_pi.c

include/types/lb\_pi.h

include/proto/lb\_pi.h

#### Edited modules:

src/backend.c

include/types/backend.h

include/types/server.h

src/cfgparse.c

src/haProxy.c

### HaProxy Load Balancing Algorithms API

Return next server from the pi tree in backend. If the tree is empty,  
return NULL. Saturated servers are skipped. A saturated server is defined in HAPROXY as a server that has above a certain (high) number of connections.

struct server   
 pi\_get\_next\_server(**struct** proxy \*p, **struct** server \*srvtoavoid)

Remove a server from a tree. It must have previously been dequeued. This function is meant to be called when a server is going down or weight changed.

static inline void   
pi\_remove\_from\_tree(**struct** server \*s)

Simply removes a server from a tree  
static inline void

pi\_dequeue\_srv(**struct server** \*s**)**

Queue a server in its associated tree.  
 Servers are sorted by the number of connections.  
 The server weight is not used  
 here.   
static inline void   
pi\_queue\_srv(**struct** server \*s)

Re-position the server in the pi tree after it has been assigned with a connection or after it has released one. Note that it is possible that  
 the server has been moved out of the tree due to failed health-checks.   
static void   
pi\_srv\_reposition(**struct** server \*s)

This function updates the server tree according to the server’s <srv>'s new state. It should be called when server <srv>'s status changes to down. It is not important whether the server was already down or not. It is not important either that the new state is completely down (the caller may not know all the variables of a server's state).  
This is used for when there are no idle servers and the current server getting the requests goes down. Thus the server that has the least current open http connections will be the next for new connections.   
static void pi\_set\_server\_status\_down(**struct** server \*srv)

This function updates the server trees according to server <srv>'s new  
 state. It should be called when server <srv>'s status changes to up.  
 It is not important whether the server was already down or not. It is not important either that the new state is completely UP (the caller may not know all the variables of a server's state). This function will not change the weight of a server which was already up.  
static void pi\_set\_server\_status\_up(**struct** server srv)

This function must be called after an update to server <srv>'s effective  
 weight. It may be called after a state change too.   
static void pi\_update\_server\_weight(**struct** server srv)

## Important implementation details

void pi\_init\_server\_tree(struct proxy p)

In order to perform a fast and clean integration we used src/lb\_fwlc.c module and expanded it when needed to get the exact Persistent idle implementation.

void

pi\_init\_server\_tree

We use http connections as the only value for node keys.

struct server

pi\_get\_next\_server(**struct** proxy p, **struct** server srvtoavoid)  
The main idea is to use the next server from lb\_fwllc.c and edit the parts of choosing the server from the elastic balancing tree and add the part where the server used last is taken. Node->key = num of active http connections the server represented by the node has. the conversion of node to server is easy by a macro (see details in code of this function).

### Config Parsing module

Below is a minimal subset of the bitwise coding in HaProxy. This subset is the group of numbers used for the parsing of the initialization files. This coding subset is the only encoding relevant to load balancing config and algorithms. Each config property is coded by the following method into integers that are compared with bitwise operations in if statements at runtime (for performance).

This coding is used to create the HAPROXY state as asked via config. Understand these to understand the parsing code related to PI.

Lower bits define the kind of load balancing method.This means the type of algorithm and which criterion it is based on. For this reason, those bits also include information about dependencies, so that the config parser can detect incompatibilities.LB parameters are on the lower 8 bits. This Depends on the LB kind.   
  
 BE\_LB\_RR\_ is used with BE\_LB\_KIND\_RR   
#define BE\_LB\_RR\_DYN 0x00000 dynamic round robin (default)   
#define BE\_LB\_RR\_STATIC 0x00001 static round robin   
  
 BE\_LB\_CB\_ is used with BE\_LB\_KIND\_CB   
#define BE\_LB\_CB\_LC 0x00000 least-connections   
#define BE\_LB\_CB\_PI 0x00007 persistent-idle   
#define BE\_LB\_PARM 0x000FF mask to get/clear the LB param

Algorithm   
#define BE\_LB\_KIND\_CB 0x02000 connection-based   
#define BE\_LB\_KIND\_HI 0x03000 hash of input (see hash inputs above)   
#define BE\_LB\_KIND 0x07000 mask to get/clear LB algorithm   
  
 All known variants of load balancing algorithms. These can be cleared using  
 the BE\_LB\_ALGO mask. For a check, using BE\_LB\_KIND is preferred.

The algorithms are classified as

CB= connection based or

RR =Round robin

the numbers are used in if statements to enforce the connection based system or RR system.

#define BE\_LB\_ALGO\_NONE (BE\_LB\_KIND\_NONE | BE\_LB\_NEED\_NONE) not defined   
#define BE\_LB\_ALGO\_RR (BE\_LB\_KIND\_RR | BE\_LB\_NEED\_NONE) round robin   
#define BE\_LB\_ALGO\_LC (BE\_LB\_KIND\_CB | BE\_LB\_NEED\_NONE | BE\_LB\_CB\_LC) least connections   
#define BE\_LB\_ALGO\_PI (BE\_LB\_KIND\_CB | BE\_LB\_NEED\_NONE | BE\_LB\_CB\_PI) persistent idle

Higher bits define how a given criterion is mapped to a server. In fact it  
 designates the LB function by itself. The dynamic algorithms will also have the DYN bit set. These flags are automatically set at the end of the parsing.  
   
#define BE\_LB\_LKUP\_NONE 0x00000 not defined   
#define BE\_LB\_LKUP\_MAP 0x10000 static map based lookup   
#define BE\_LB\_LKUP\_RRTREE 0x20000 FWRR tree lookup   
#define BE\_LB\_LKUP\_LCTREE 0x30000 FWLC tree lookup   
#define BE\_LB\_LKUP\_CHTREE 0x40000 consistent hash   
#define BE\_LB\_LKUP\_FSTREE 0x50000 FAS tree lookup   
#define BE\_LB\_LKUP 0x70000 mask to get just the LKUP value   
#define BE\_LB\_LKUP\_PITREE 0x60000 PI tree lookup   
  
 The scale factor between user weight and effective weight allows smooth weight modulation even with small weights (eg: 1). It should not be too high though because it limits the number of servers in FWRR mode in order to prevent any integer overflow. The max number of servers per backend is  
limited to about (2^32-1)/256^2/scale ~= 65535.9999/scale. A scale of 16 looks like a good value, as it allows 4095 servers per backend while leaving modulation steps of about 6% for servers with the lowest weight

#define BE\_WEIGHT\_SCALE 16

## How To Compile, Install and Run HaProxy From Source

We used git and this guide (it’s the best, we tried them all XD) that provide step by step instructions:

https://www.serverlab.ca/tutorials/linux/network-services/compiling-and-running-haproxy-from-source-on-ubuntu-14/

In case this link dies here is a summary:

1. Install the build-essential package.

sudo apt-get install build-essential

1. Install libssl-dev to allow SSL functionality

sudo apt-get install libssl-dev

1. Install PCRE development library for C++, used to improve regex performance.

sudo apt-get install libpcre++-dev

1. Create the user account that HAProxy will run as. This account should standard user privileges for security reasons. Running as Root can be done, but it is strongly discouraged!

sudo useradd haproxy

1. Use git to clone the source code repo
2. Git clone repo\_url
3. Compilation
4. make TARGET=linux2628 CPU=native USE\_STATIC\_PCRE=1 USE\_OPENSSL=1 USE\_ZLIB=1
5. Install the newly compiled program.

sudo make install

1. open and edit the file:  
   sudo touch /etc/init.d/haproxy
2. Make the file executable.

sudo chmod +x /etc/init.d/haproxy

1. Open the file in a text editor

sudo vi /etc/init.d/haproxy

1. paste this

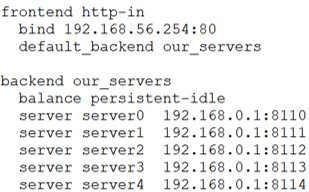
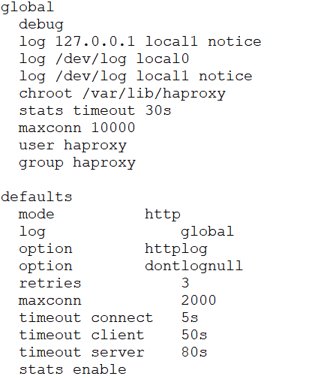
### Service init script

|  |
| --- |
| #!/bin/sh |
|  | ### BEGIN INIT INFO |
|  | # Provides: haproxy |
|  | # Required-Start: $local\_fs $network $remote\_fs |
|  | # Required-Stop: $local\_fs $remote\_fs |
|  | # Default-Start: 2 3 4 5 |
|  | # Default-Stop: 0 1 6 |
|  | # Short-Description: fast and reliable load balancing reverse proxy |
|  | # Description: This file should be used to start and stop haproxy. |
|  | ### END INIT INFO |
|  |  |
|  | # Author: Arnaud Cornet <acornet@debian.org> |
|  | # Modified by: Shane Rainville <shane.rainville@serverlab.ca> |
|  |  |
|  | PATH=/sbin:/usr/sbin:/bin:/usr/bin |
|  | PIDFILE=/var/run/haproxy.pid |
|  | CONFIG=/etc/haproxy/haproxy.cfg |
|  | HAPROXY=/usr/local/sbin/haproxy |
|  | CONFIGTEST\_LOG=/var/log/haproxy\_configtest.log |
|  | EXTRAOPTS= |
|  | ENABLED=0 |
|  |  |
|  | test -x $HAPROXY || exit 0 |
|  | test -f "$CONFIG" || exit 0 |
|  |  |
|  | if [ -e /etc/default/haproxy ]; then |
|  | . /etc/default/haproxy |
|  | fi |
|  |  |
|  | test "$ENABLED" != "0" || exit 0 |
|  |  |
|  | [ -f /etc/default/rcS ] && . /etc/default/rcS |
|  | . /lib/lsb/init-functions |
|  |  |
|  |  |
|  | haproxy\_start() |
|  | { |
|  | start-stop-daemon --start --pidfile "$PIDFILE" \ |
|  | --exec $HAPROXY -- -f "$CONFIG" -D -p "$PIDFILE" \ |
|  | $EXTRAOPTS || return 2 |
|  | return 0 |
|  | } |
|  |  |
|  | haproxy\_stop() |
|  | { |
|  | if [ ! -f $PIDFILE ] ; then |
|  | # This is a success according to LSB |
|  | return 0 |
|  | fi |
|  | for pid in $(cat $PIDFILE) ; do |
|  | /bin/kill $pid || return 4 |
|  | done |
|  | rm -f $PIDFILE |
|  | return 0 |
|  | } |
|  |  |
|  | haproxy\_reload() |
|  | { |
|  | $HAPROXY -f "$CONFIG" -p $PIDFILE -D $EXTRAOPTS -sf $(cat $PIDFILE) \ |
|  | || return 2 |
|  | return 0 |
|  | } |
|  |  |
|  | haproxy\_status() |
|  | { |
|  | if [ ! -f $PIDFILE ] ; then |
|  | # program not running |
|  | return 3 |
|  | fi |
|  |  |
|  | for pid in $(cat $PIDFILE) ; do |
|  | if ! ps --no-headers p "$pid" | grep haproxy > /dev/null ; then |
|  | # program running, bogus pidfile |
|  | return 1 |
|  | fi |
|  | done |
|  |  |
|  | return 0 |
|  | } |
|  |  |
|  | haproxy\_configtest() |
|  | { |
|  | $HAPROXY -f "$CONFIG" -c > "$CONFIGTEST\_LOG" 2>&1 |
|  | ret=$? |
|  | if [ $ret -eq 0 ]; then |
|  | # Valid config - remove $CONFIGTEST\_LOG |
|  | rm "$CONFIGTEST\_LOG" |
|  | fi |
|  |  |
|  | return $ret |
|  | } |
|  |  |
|  |  |
|  | case "$1" in |
|  | start) |
|  | log\_daemon\_msg "Starting haproxy" "haproxy" |
|  | haproxy\_start |
|  | ret=$? |
|  | case "$ret" in |
|  | 0) |
|  | log\_end\_msg 0 |
|  | ;; |
|  | 1) |
|  | log\_end\_msg 1 |
|  | echo "pid file '$PIDFILE' found, haproxy not started." |
|  | ;; |
|  | 2) |
|  | log\_end\_msg 1 |
|  | ;; |
|  | esac |
|  | exit $ret |
|  | ;; |
|  | stop) |
|  | log\_daemon\_msg "Stopping haproxy" "haproxy" |
|  | haproxy\_stop |
|  | ret=$? |
|  | case "$ret" in |
|  | 0|1) |
|  | log\_end\_msg 0 |
|  | ;; |
|  | 2) |
|  | log\_end\_msg 1 |
|  | ;; |
|  | esac |
|  | exit $ret |
|  | ;; |
|  | reload|force-reload) |
|  | log\_daemon\_msg "Reloading haproxy" "haproxy" |
|  | haproxy\_reload |
|  | case "$?" in |
|  | 0|1) |
|  | log\_end\_msg 0 |
|  | ;; |
|  | 2) |
|  | log\_end\_msg 1 |
|  | ;; |
|  | esac |
|  | ;; |
|  | restart) |
|  | log\_daemon\_msg "Checking haproxy configuration" "haproxy" |
|  | haproxy\_configtest |
|  | ret=$? |
|  | case "$ret" in |
|  | 0) |
|  | log\_end\_msg 0 |
|  | ;; |
|  | 1) |
|  | log\_end\_msg 1 |
|  | echo "Restart process aborted." |
|  | echo "Check $CONFIGTEST\_LOG for details." |
|  | # Abort restart |
|  | exit $ret |
|  | ;; |
|  | esac |
|  | log\_daemon\_msg "Restarting haproxy" "haproxy" |
|  | haproxy\_stop |
|  | haproxy\_start |
|  | case "$?" in |
|  | 0) |
|  | log\_end\_msg 0 |
|  | ;; |
|  | 1) |
|  | log\_end\_msg 1 |
|  | ;; |
|  | 2) |
|  | log\_end\_msg 1 |
|  | ;; |
|  | esac |
|  | ;; |
|  | status) |
|  | haproxy\_status |
|  | ret=$? |
|  | case "$ret" in |
|  | 0) |
|  | echo "haproxy is running." |
|  | ;; |
|  | 1) |
|  | echo "haproxy dead, but $PIDFILE exists." |
|  | ;; |
|  | \*) |
|  | echo "haproxy not running." |
|  | ;; |
|  | esac |
|  | exit $ret |
|  | ;; |
|  | configtest) |
|  | haproxy\_configtest |
|  | ret=$? |
|  | case "$ret" in |
|  | 0) |
|  | echo "haproxy configuration is valid." |
|  | ;; |
|  | 1) |
|  | echo "haproxy configuration is NOT valid. Check $CONFIGTEST\_LOG for details." |
|  | ;; |
|  | esac |
|  | exit $ret |
|  | ;; |
|  | \*) |
|  | echo "Usage: /etc/init.d/haproxy {start|stop|reload|restart|status|configtest}" |
|  | exit 2 |
|  | ;; |
|  | esac |

### Sample Config file

|  |
| --- |
| global |
|  | log 127.0.0.1 local0 notice |
|  | maxconn 2000 |
|  | user haproxy |
|  | group haproxy |
|  |  |
|  | Defaults |
|  | log global |
|  | mode http |
|  | option httplog |
|  | option dontlognull |
|  | retries 3 |
|  | option redispatch |
|  | timeout connect 5000 |
|  | timeout client 10000 |
|  | timeout server 10000 |
|  |  |
|  | listen myapp 0.0.0.0:80 |
|  | mode http |
|  | stats enable |
|  | stats uri /haproxy?stats |
|  | stats realm Strictly\ Private |
|  | stats auth A\_Username:YourPassword |
|  | stats auth Another\_User:passwd |
|  | balance roundrobin |
|  | option httpclose |
|  | option forwardfor |
|  | server web-node-01 192.168.1.101:80 check |
|  | server web-node-02 192.168.1.100:80 check |

### Project's Final Config File haproxy.cfg



### Run

sudo service haproxy start

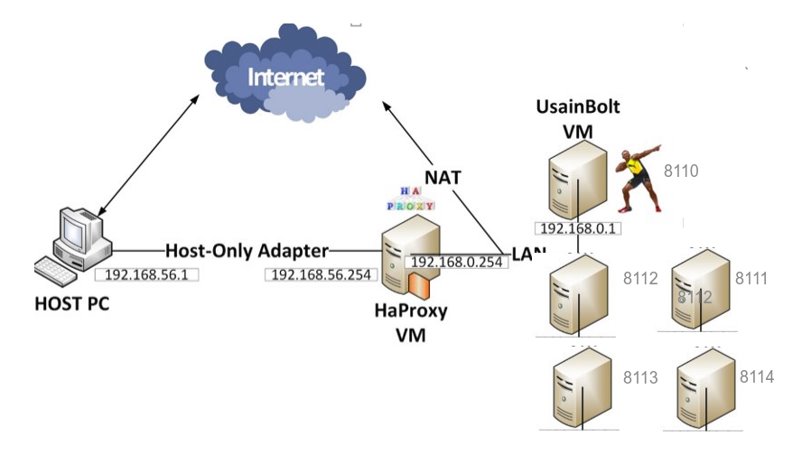
This triggers the bash script you pasted before that does all of the black magic needed to start the haproxy service you installed from source code.

### Debug

HAPROXY logs its output into a different process that can even run on a different server than haproxy itself, the process is called syslog.   
When you print out a variable into stdout from haproxy it goes to syslog. Syslog can be read using the command: journal.



# Phase 4 – The Scaled-Up System & Networking



## Figure 4 – System Architecture

One desktop lab pc running Ubuntu:

1) Host – running Virtual box for servers and the client script.

2) ProxyVM – running the HAProxy processes on 2 cores.

3) UsainBolt ServerVm – running 5 web servers on ubuntu.

**Networking using virtual box settings as described in the diagram. The numbers near the servers are the ports the servers listen at all on the IP above.**

To gather data for the analysis we have defined the following setup for the simulation.

## Final System setup

### Load-Balancer

VM running HaProxy with our code additions that support PI algorithm. We have allocated a single core and 1GB RAM memory for that VM.

### Worker Servers

single server VM which runs 5 web.py servers simultaneously (one on each core). We have allocated 6 cores (1 for each server process and another for the OS) and 8 GB memory. To make sure that each server runs on different core (making it independent of other servers) we have forced each process to run on a specific core (and that core only) using the following command:

taskset -c <core #> <process to run>

This way the server runs on one core and this core only thus the OS scheduling system does not interfere.

To try to mimic servers’ heterogeneity, we have configured 3 of them to calc with only 3 digits (faster servers) and the remaining 2 serves configured to calc with 4 digits (slower servers).

### Client

The host PC runs a script that generates requests and sends them to the load-balancer. The script creates 50 parallel threads, and each thread sends 1000 requests to the load-balancer serially (request is sent only after response from the previous request is received) – total of 50,000 requests per run. Each request is a random 5-digit number. When response is received all the related data saved to an CSV file.

The format of the CSV columns is as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **total Handling Time** | **Server Name** | **request Num** | **time Finished Calc** | **num To Calc** | **time Request Received** | **time Started Calc** |
| 0.000319958 | server0 | 265993 | 1537732236 | 52705 | 1537732236 | 1537732236 |
| 0.000201941 | server2 | 265573 | 1537732236 | 19190 | 1537732236 | 1537732236 |

### Figure 5 – Results Data Table Format

To be able to analyze results for each algorithm we have simulated 10 runs for each, so we could minimize the error caused by passing network delays that can happen in a single run.

# Phase 5 – Analysis of results and future suggestions

## Analysis

We have concluded that the best way to compare between the runs is by using a histogram where each bin is a range of total handling time per single request, and the value in each bin is the number of requests whose total handling time is within the range.

Since there were slight differences between the runs, we decided to compare between the best and the worst run of each algorithm.

The range of total handling time is quite wide so after fine tuning we have defined the range bins as follows:

Within values 0.00001-0.0001 – the “width” of range of each bin is 0.00001. Within the values 0.0001-0.03 - the “width” of range of each bin is 0.0005

### Figure 6 – PI Best vs. Worst Run

### Figure 7 – LC Best vs. Worst run

### Figure 8 – RR Best vs. Worst Run

We can conclude from these graphs that PI algorithm is the most stable, since there is almost no difference between two edge runs.

Of course, the main goal war to compare between the different algorithms. The next 2 graphs compare between those 3 algorithms – for both their best and worst runs.

### Figure 9 – Worst Run

### Figure 10 – Best Run

We can conclude from these graphs that when using PI algorithm, less requests handled quickly and that there are more requests with a significantly longer handling time.

## Suggestions For Future Analysis

1. There are many parameters that could skew these results, thus we recommend running additional simulations with different setups to be able to fully compare between those algorithms.
2. Alter the number of servers in the backend.
3. Passing different size numbers to the server for prime calculation (see phase 2 server) did not provide the expected heterogeneity, better use hardware or thread based heterogeneity.
4. Alter the number of threads and the total amount of requests generated per run.
5. Alter the heterogeneity of the servers and the requests by hardware using a cloud provider such as AWS.
6. To better imitate large scale big data consuming applications let the servers handle many HTTP connections simultaneously using multithreading and multi processes server backend.
7. Implementation of the PI algorithm within HAProxy from scratch – without relying on LC implementation.
8. Alter the threshold that defines idle state (we used the trivial threshold of 0).