Non-stop Provision of Internet Services via a Reflectively Load-Sharing Architecture

Kostas Zorbadelos[1], Christos KK Loverdos[1], and Alex Delis[2]

[1] OTENET S.A.
15124, Maroussi, Greece
{kzorba,loverdos}@otenet.gr

[2] University of Athens
15771, Athens, Greece
ad@di.uoa.gr

Abstract. There exists a growing need to offer continuously available and reliable Internet services to a very large number of companies, organizations and individuals. Such services include, but are not limited to, http, ldap, smtp, radius, and dns. In this paper, we propose a reflective, load-balancing, and highly available architecture to address this problem and discuss key aspects of our prototype implementation. Our solution is based on farms or pools of computing nodes that are essentially responsible for the delivery of services. All these nodes operate behind a network-device that acts as the front-end to the services and coordinates the work of the nodes in a reflective and load-conscious manner. A comprehensive and highly configurable administration shell allows for the on-line manipulation and handling of both network-device and computing nodes in either interactive or batch-node. Our main objective is to create an open-source-based system that not only provides 7x24-availability to Internet services offered, but can also grow in terms of resources gracefully while avoiding down-times and long reconfiguration delays. To this effect, our solution aspires to successfully replace proprietary embedded machinery geared into offering non-stop provision of Internet services.

1 Introduction

The ongoing surge on Internet traffic and the continual deployment of new Web and information services do pose ever-increasing demands on the performance of ISP servers. Clearly, one could address high-rates of Internet traffic and requests for services by simply over-provisioning through upgrades of few critical servers and ultimate purchase of more powerful hardware. Yet, this solution is often deemed limited, in the sense that soon new upgrades will be required. Alternatively, we can expand the networks of ISP-servers with new, often off-the-self, and inexpensive hardware to form networks of computing nodes that will collectively facilitate the ISP incoming/outgoing traffic. The presence of multiple such machines calls for not only an organizational structure but perhaps more critical is the requirement to have all elements in the network of nodes to be equally utilized to avoid resource overload.

In light of a growing number of servers, there is an ever more frequent problem of nodes failing. Our architecture should feature an "intelligent" way to cease forwarding traffic to failed servers and start using them again once their corresponding services recover. Ideally, the system should self-adapt its configuration so that any connections to failed servers are migrated to operational ones without clients noticing the difference [5]. In this manner, we should be able to provide transparent fail-over and high-availability both crucial in ensuring the quality of rendered Web and information services.

To address the above requirements when it comes to ISP services, we use an approach that combines both load-sharing and high-availability. In our proposed scheme, any Internet service is given a virtual *IP* on a *network-device* for which we guarantee 7x24-availability; the actual client requests are load-balanced in a cluster of machines physically located *behind* the network-device in question. Our approach has a few notable advantages:

- *extensibility*: more servers can be readily added to the cluster as soon as traffic increases.
- *cost-effectiveness*: through the use of commodity hardware, we intend to maximize performance/cost ratio.
- *better perceived QoS for the end user*: this is attained by minimizing down-times for Web and information services. To this effect, any organization that adopts such an architecture is anticipated to enhance its credibility among the client base.
- *single point of entry*: from a client's point of view, the cluster appears to be a single server that responds to its requests. By utilizing network address manipulation techniques such as PF's `nat` and `rdr`, the back-end network may consist of machines running any operating system with a TCP/IP stack. The cluster nodes may use private Internet addresses; however, only one public IP-address is visible to the outside word as a point of entry.
- *easy security policy enforcement*: the isolation of the cluster nodes from the outside world requires each packet pass through the *network-device* offering a single point for enforcing low-level security policies.
- *administration convenience*: software upgrades can be scheduled with no loss of service at any moment; besides, more nodes can be added/taken-off from the cluster and nodes can be easily reassigned to other services on the fly.

Our proposed solution involves the following well-established open-source technologies:

– FreeBSD [10], the contemporary operating system derived from the BSD version of UNIX®. FreeBSD enjoys the largest installation base among BSD-derived systems and has an established reputation for system stability, performance, security and advanced networking features.
– The Packet Filter (PF) [16, 14], used for filtering TCP/IP traffic and carrying out necessary network address manipulation. It is also capable of normalizing and conditioning TCP/IP traffic, providing bandwidth control and packet prioritization. PF can be also used to create powerful and flexible firewalls [7]. It is derived from OpenBSD [15], a project widely known for its efforts to offer security and integrated cryptography at the system design level. PF was ported to FreeBSD and became an integral part of its base system since `5.3-RELEASE`.
– Common Address Redundancy Protocol (CARP) [14] and pfsync which are collectively used to ensure the high availability of our solution [8]. CARP, is a secure free alternative to other redundancy protocols [6, 4] while pfsync is a UNIX device that transfers

connection states between different machines. They both originate from the OpenBSD project.

– Python [17], the scripting language with rapid prototyping capabilities is predominantly used in the implementation of our administration shell.

The rest of the paper is organized as follows: Section 2 outlines our proposed system architecture and Section 3 analyzes our design decisions and discusses our prototype implementation. Section 4 presents the evaluation of our a few typical scenarios. Finally, Section 5 presents related work and Section 6 concludes with remarks for planned future extensions.

2 System Architecture

We build a FreeBSD-based network-device (Fig. 1), placed between two networks, a publicly accessible and an internal one. Each Internet service rendered is assigned an IP is the publicly accessible network and a corresponding server farm (or pool) in the internal network. This group of nodes that realizes a specific service constitutes a *virtual server*. All pertinent traffic to the service regardless of its origination has to pass through the network-device that features the specific IP and is distributed to the corresponding server pool.

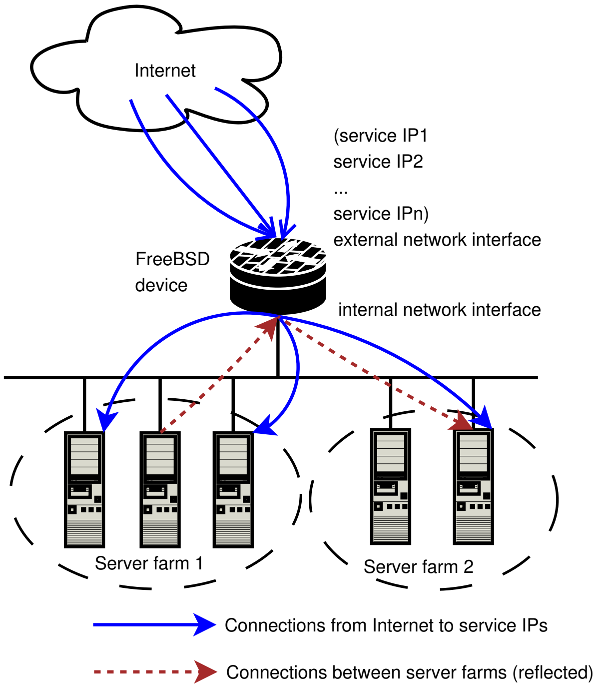


Fig. 1. Architecture overview.

In our solution, PF provides the load-sharing functionality and CARP with pfsync are responsible for giving high availability to the gateway device. Moreover, we provide a specialized shell, whose key objectives are to coordinate the function of the underlying system and help configure it in a straightforward manner. The shell is capable of both (re)configuring the load-balancing device and monitoring the running machines and services. The shell co-operates with a *poller* component to achieve the monitoring functionality; based on the polling outcome our system may eliminate defective machines from the server farms to ensure high service availability.

2.1 Reflective Load-Sharing

As farms of servers are the main elements for service provision in the internal network, every farm is designated to handle a specific type of requests. Types of requests include http, ssh, ldap, smtp, and radius. It is worth pointing out that a server may be shared among multiple pools.

Our architecture employs a reflective-approach in handling inter-related services. For instance, an smtp session may necessitate multiple ldap requests. The latter have to be serviced by a different farm and individual ldap calls may finally be answered by different servers in the corresponding farm. The decision of which server is used at any specific moment in time is undertaken by the network-device and is the major reflection aspect of our architecture; the reflection is depicted in Fig. 1 by the dashed lines. Overall, the network-device carries out not only balancing of outside-world requests, but also load-shares requests among the farms themselves. The reflective aspects of our proposed architecture are discussed in the context of two specific examples below:

Mail delivery with LDAP queries : In this scenario, we deal with smtp and ldap services that need to be collectively load-shared as Fig. 2 indicates. Our publicly accessible service IPs are located in the range `62.103.127.144/28` and the actual servers are placed in the LAN `192.168.1.0/24`. For each service we configure a separate IP address as shown below in the DNS entries:

```
[zone ourdomain.tld]
...
@    IN  MX  10  mx.ourdomain.tld.
mx   IN  A   62.103.127.154
ldap IN  A   62.103.127.157
```

The remote Mail Transfer Agent (MTA)(e.g sendmail or postfix) attempts to deliver a message destined to a recipient at `ourdomain.tld`. For this purpose, it opens a connection to `mx.ourdomain.tld` (the declared `MX` in DNS for `ourdomain.tld` zone) port 25. The connection is finally routed to the external interface of our device, since the `MX`-IP is one of the service IPs configured. According to the PF rules active on the network-device shown in Fig. 3, the connection is redirected to one of the machines in the pool for the mail delivery service (line 17). This is accomplished in a round-robin fashion.

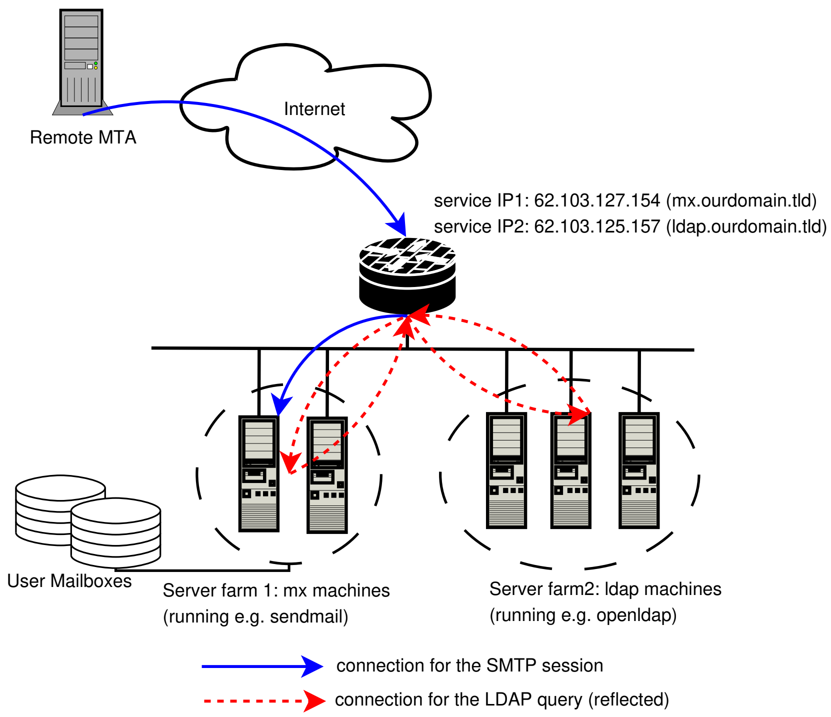


Fig. 2. Email delivery with LDAP queries.

The target machine issues an ldap query to determine whether the user exists and to find all the necessary information to perform the delivery (e.g., where the location of her mailbox is). An ldap query can also be performed for other reasons, like the resolving of mail aliases stored in the catalogue. The ldap query causes a new connection to the service IP configured for the ldap service (`ldap.ourdomain.tld`). This time the connection reaches our device through its internal interface. Therefore, the redirection rule on the external interface is not applicable. The reason is that the TCP/IP stack on the device compares the destination address of incoming packets with its own addresses and aliases and detects connections to itself as soon as they have passed the internal interface. It is for these reflected connections (connections to public service IPs from clients inside the machine pools network) that `rdr` and `nat` rules are necessary on the internal interface (lines 18, 21, 24, 27, 28, of Fig. 3).

By using PF tables in our rules, it is easy to manipulate them on the fly using `pfctl` [14]. This way, we can detect and eliminate any failed nodes in our setup by polling the cluster machines. However, at this time, no other algorithms other than `round-robin` can be used with PF tables. Other supported algorithms in PF include `bitmask`, `random` and `source-hash` but require that the pool be expressed in a CIDR (Classless Inter-Domain Routing) block. All these algorithms are described in the pf.conf man page. The achieved result is that both connections originating from the outside world and the reflected ones form the inside network get load balanced to the service machine pools. Failing nodes are transparently eliminated.

```
ext_if="xl0"    # replace with actual external interface name i.e., dc0
int_if="xl1"    # replace with actual internal interface name i.e., dc1
internal_net="192.168.1.0/24"
internal_gw="192.168.1.1"
external_addr="62.103.127.153"
# The service IP for mx.ourdomain.tld
mx_service_ip="62.103.127.154"
# The service IP for ldap.ourdomain.tld
ldap_service_ip="62.103.127.157"
# The pool of mx mail servers
table <mx_pool> {192.168.1.2,192.168.1.3}
# The pool of ldap servers
table <ldap_pool> {192.168.1.2,192.168.1.3,192.168.1.4}
# This is required for the internal machines to see the outside world
nat on $ext_if from $internal_net to any -> $external_addr
# The rdr rules for the mx mail service
rdr on $ext_if proto tcp from any to $mx_service_ip port 25 -> <mx_pool>
rdr on $int_if proto tcp from $internal_net to $mx_service_ip port 25 -> <mx_pool>
# This is necessary for the reflection of connections from internal net
# to the mx mail service
nat on $int_if proto tcp from $internal_net to <mx_pool> port 25 -> $internal_gw
# The rdr rules for the ldap service
rdr on $ext_if proto tcp from any to $ldap_service_ip port 389 -> <ldap_pool>
rdr on $int_if proto tcp from $internal_net to $ldap_service_ip port 389 -> <ldap_pool>
# This is necessary for the reflection of connections from internal net
# to the ldap service
nat on $int_if proto tcp from $internal_net to <ldap_pool> port 389 -> $internal_gw
no nat on $int_if proto tcp from $int_if to $internal_net
```

Fig. 3. The PF rules for mail delivery with ldap lookups.

Web sites with user sessions and LDAP authentication There are two services in this example, http and ldap. The catalog, maintains the user database for the registration in the sites. The following DNS entries demonstrate the service IPs we use in our setup.

```
[zone ourdomain.tld]
...
www   IN  A   62.103.127.154
ldap  IN  A   62.103.127.157
```

The main difference with the scenario presented before, is the need to direct each client to the same web server. There are many web applications that require some form of authentication/registration of the user and create sessions that persist for a period of time. The http session objects can contain a wide variety of data including like user personal preferences and shopping cart information. It is therefore necessary to always direct the connections of a specific client to the web server that holds its session.

There are two ways to achieve the above functionality with PF. The first is the `source-hash` load balancing method that can be declared in a `rdr` or `nat` rule that deals with address pools. In this method, each client's IP is hashed and always mapped on the same server, which solves our problem. However, in the current implementation of PF, all load balancing methods except for the `round-robin`, require that the address pool be expressed as a CIDR network block and the use of PF tables to express the pool is prohibited. There is a second way that provides the required functionality, without all the previously mentioned restrictions. We can use the `sticky-address` option in `random` and `round-robin` pool types to ensure that a particular source address is always mapped to the same redirection address. This, by default ensures the stickiness of a client as long as there is at least a connection state entry in PF for that client. Nonetheless, `sticky-address` works also without states, provided that one sets a reasonable value for the `src.track` time-related parameter as described in pf.conf man page.

```
1  ext_if="xl0"    # replace with actual external interface name i.e., dc0
2  int_if="xl1"    # replace with actual internal interface name i.e., dc1
3  internal_net="192.168.1.0/24"
4  internal_gw="192.168.1.1"
5  external_addr="62.103.127.153"
6  # The service IP for ldap.ourdomain.tld
7  ldap_service_ip="62.103.127.157"
8  # The service IP for www.ourdomain.tld
9  www_service_ip="62.103.127.154"
10 # The pool of ldap servers
11 table <ldap_pool> {192.168.1.2,192.168.1.3,192.168.1.4}
12 # The pool of web servers
13 table <www_pool> {192.168.1.2,192.168.1.3,192.168.1.4}
14 # This is required for the internal machines to see the outside world
15 nat on $ext_if from $internal_net to any -> $external_addr
16 # The rdr rules for the ldap service
17 rdr on $ext_if proto tcp from any to $ldap_service_ip port 389 -> <ldap_pool>
18 rdr on $int_if proto tcp from $internal_net to $ldap_service_ip port 389 -> <ldap_pool>
19 # This is necessary for the reflection of connections from internal net
20 # to the ldap service
21 nat on $int_if proto tcp from $internal_net to <ldap_pool> port 389 -> $internal_gw
22 # The rdr rules for the www service
23 rdr on $ext_if proto tcp from any to $www_service_ip port 80 -> <www_pool> sticky-address
24 no nat on $int_if proto tcp from $int_if to $internal_net
```

Fig. 4. The PF rules for web sites with ldap authentication.

The `sticky-address` method is used in our rules as depicted in line 23 of Fig. 4. Note that there is no need for connection reflection in the http service since no connections to web servers are generated from machines in the internal network.

2.2 High Availability

From all previous sections it becomes apparent that our network-device is critical for the operation of both the services and the network connectivity of the constituent servers to the outside world. In our experimental setup, the device also provides DNS service for our domain (`ourdomain.tld`). It is therefore important to take measures to ensure that the device is not a single point of failure in the network.

CARP and pfsync are the components we use to address this problem. They both come from OpenBSD (since its `3.5 Release`) just like PF. With the use of these tools we can achieve a setup similar to the one shown in Fig. 5, where we have two devices in parallel.

All traffic passes through the primary device and in case of failure the backup device assumes the primary's role and continues where the first left off.

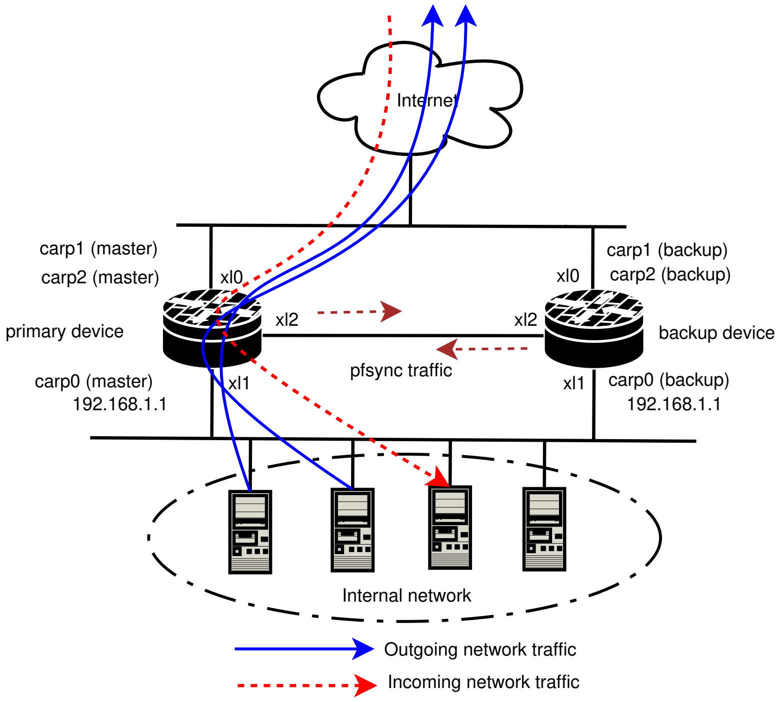


Fig. 5. Highly available device setup.

The Common Address Redundancy Protocol manages fail-over at the Link and IP layers (layers 2 and 3 in the OSI Model respectively). It provides the capability to define groups with each CARP group having a virtual MAC (link layer) address, and one or more virtual host IP addresses (the common address). The master of an address sends out CARP advertisement messages via multicast using the CARP protocol on a regular basis, and the backup hosts listen for this advertisement. If the advertisements stop from the master, the backup hosts begin advertising (becoming masters themselves). The advertisement frequency is configurable, and the host which advertises more frequently is the one most likely to become master in the event of a failure. CARP has a few technical differences from similar protocols [4, 6] and it is not patent encumbered. CARP advertisements are cryptographically protected.

Pfsync transfers `nat` and `rdr` state information between the devices. Each device sends these messages out via multicast on a specified interface, using the PFSYNC protocol. It also listens on that interface for similar messages from other devices, and imports them into the local state table. This way the connection states propagate on both devices. In the event of a device failure the second device can resume operations with no connections affected. Since there is no built-in authentication or any other sort of protection for the

PFSYNC messages, it is strongly recommended that a dedicated, trusted network be used for pfsync (like a simple crossover cable between interfaces on the two devices).

In our setup, we create a carp interface to hold the gateway IP for the internal servers (`192.168.1.1`) as well as a carp interface for each public service IP. We have also enabled the `sysctl` carp preempt option (net.inet.carp.preempt) on both devices, since it is desirable to fail-over all of the carp interfaces together, when one of the physical interfaces goes down. The exact configuration in the interfaces of our devices is depicted in Fig. 6. Notice the difference in the `advskew` parameter that causes the first device to become master whenever it is available, since the net.inet.carp.preempt `sysctl` state is in effect.

```
[device-1 (master)]
ifconfig_carp0="vhid 1 pass carp0pass 192.168.1.1/24"
ifconfig_carp1="vhid 2 pass carp1pass 62.103.127.154/28"
ifconfig_carp2="vhid 3 pass carp2pass 62.103.127.157/28"

[device-2 (backup)]
ifconfig_carp0="vhid 1 pass carp0pass advskew 100 192.168.1.1/24"
ifconfig_carp1="vhid 2 pass carp1pass advskew 100 62.103.127.154/28"
ifconfig_carp2="vhid 3 pass carp2pass advskew 100 62.103.127.157/28"
```

Fig. 6. The /etc/rc.conf definitions for the carp interfaces.

3 Software Components for the Handling of the Cluster Elements

By using the separation of concerns principle [9] during our design phase, we have identified that the following key elements are required for the implementation of the proposed architecture:

- An administration shell.
- A health checking component.
- A data model.

We introduce the *administration shell* for the configuration of our load balancing network-device and for service status reporting. The *health checking* component is responsible for monitoring service liveness and acting appropriately on service failure or recovery. Last but not least, a *data model* is needed to *a)* represent the basic concepts of our architecture and *b)* provide a communication medium between the administration shell and the health checking component. Below, we further present the specific functionalities and design choices for the three elements.

3.1 lbsh: A Domain Specific Shell for Administration

Although PF rules are straightforward to write, we believe that we need a specific command repertoire to directly support the proposed architecture and its concepts such as *services* and *farms/pools of servers*. lbsh, which we have developed for this purpose, interprets a Domain Specific Language (DSL) designed for our environment. lbsh is actually an application of a "meta-shell" that we have implemented and named oyster. oyster offers a generic API for the creation of application specific shells.

The code in Fig. 7, written in our DSL, configures the device for serving smtp and ldap requests by dedicated pools of servers and defines both external and internal interfaces, the internal network range used by the server farms and the gateway for the internal network. Comments are introduced using the '#' character.

```
interface external xl0
interface internal xl1

internal net 192.168.1.0/24
internal gw  192.168.1.1

service SMTP is 62.103.127.154 tcp 25
service LDAP is 62.103.127.157 tcp 389 reflect

# The pool of SMTP servers
pool mx_pool for SMTP
add 192.168.1.2 to mx_pool
add 192.168.1.3 to mx_pool

# The pool of LDAP servers
pool ldap_pool for LDAP
add 192.168.1.2 to ldap_pool
add 192.168.1.3 to ldap_pool
add 192.168.1.4 to ldap_pool
```

Fig. 7. Sample device configuration using lbsh.

Our Domain Specific Language statements are directly mapped to underlying PF rules. Moreover, the shell provides commands for checking the status of services which is particularly useful in interactive sessions. An example of such an interactive session is given in Fig. 8. lbsh is capable of providing hints for command completion and usage when executing interactively, as seen in lines 25 and 26 of Fig. 8.

A brief description of the meta-shell: oyster is an object-oriented framework that assists in creating extensible shells [13]. The main idea is to provide a set of APIs for:

- The specification and interpretation of Domain Specific Languages.
- The generation of shells that can be executed either interactively or in batch mode.

oyster provides:

- A uniform object-oriented API that enables the creation of new commands.
- Programmable completion used in interactive shell sessions.

```
>> status 192.168.1.2
192.168.1.2 is configured for SMTP, LDAP
SMTP on 192.168.1.2 is UP   [1 days, 00:12:34]
LDAP on 192.168.1.2 is DOWN [0 days, 00:00:02]

>> status ldap_pool
ldap_pool       LDAP
--------------- ----
192.168.1.2     DOWN
192.168.1.3     UP
192.168.1.4     UP

>> pool SOME_POOL for FOO
ERROR: undefined service FOO

>> service FOO as 62.103.127.150 tcp 39
Added new service FOO with dedicated IP 162.103.127.150 and
TCP Port 39

>> pool SOME_POOL for FOO
>> help add
Syntax: ``add'' IP ``to'' POOL
Example: add 192.168.1.100 to some_pool_name

>> add 192.168.1.100 to <Tab is pressed>
ldap_pool mx_pool SOME_POOL <completions suggested by the shell>
>> add 192.168.1.100 to SOME_POOL
Added 192.168.1.100 to pool SOME_POOL
```

Fig. 8. lbsh interactive session.

- Namespaces so that commands with logically related functionality can be grouped together.
- An abstraction layer for input sources so that we can uniformly treat both a readline-capable terminal session and a file.
- A reference shell implementation with a predefined set of commands.

oyster does not enforce the specification of a DSL by means of a grammar, but this functionality can be easily built by using the facilities oyster provides. A prototype implementation exists in Python, leveraging the full power of this scripting language [13]. Describing the full details of oyster's design and prototype implementation is beyond the scope of this paper; however, we present in Fig. 9 its core structure in UML. We intend to release oyster soon as an open-source project.

lbsh commands: We leverage the API provided by oyster in order to create our administration shell. The domain specific commands are created under namespace `lb`. In the following paragraphs, we give the command usage and explain the interpretation of three such statements, namely, `service`, `pool` and `add`. We represent literal strings using single quotes and optional parameters using brackets.

◇ `'interface' 'internal'|'external' NAME [ifconfig-params]`
This is used to declare the internal and external interfaces of the device. If the optional

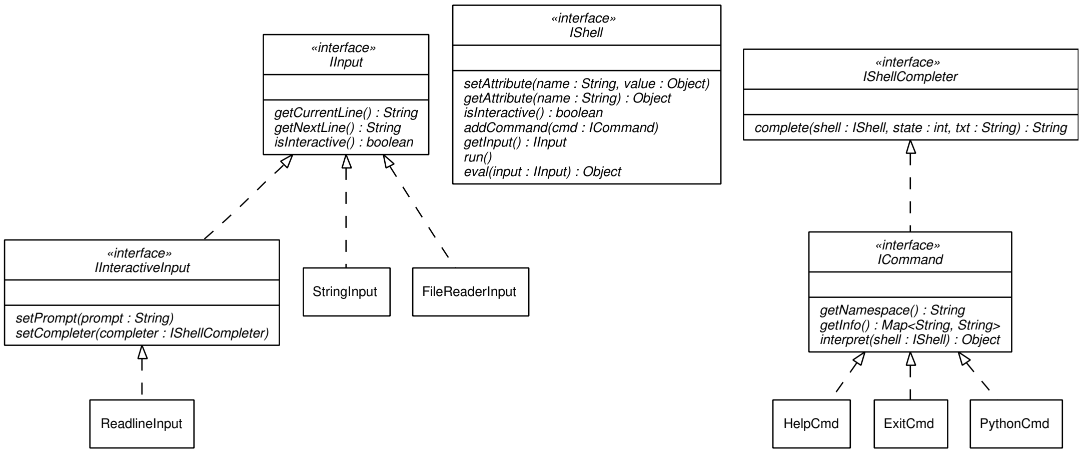


Fig. 9. A snapshot of oyster's Object-Oriented Programming Interface.

`ifconfig-params` are given, then lbsh assumes that the user wants to configure those interfaces and so `ifconfig` is executed. When the command is interpreted by the shell, appropriate entries are created in the data model for the names of the interfaces.

◇ `'internal net' CIDR`
The command is used to declare the network block for the server farms.

◇ `'internal gw' IP`
This declares the gateway IP to be used by the machines in the server pools.

◇ `'service' NAME 'is' IP PROTO PORT ['sticky'] ['reflect']`
This statement is used to define our services. We give a compact specification, containing all the necessary parameters for the generation of appropriate PF rules. If the optional parameters `sticky` and `reflect` are present, they are used to enable the `sticky-address` feature and connection reflection respectively. At the lower level, this command just updates the data model with a new definition.

◇ `'pool' NAME 'for' SERVICE`
A named pool is created for the given service. Currently, there is one-to-one correspondence between a pool and a service. By default, the command creates both a PF table and an `rdr` rule for the service. If `reflect` has been specified in the service's corresponding definition, then additional `rdr` and `nat` rules are generated. For example, these commands:

```
service WWW is 62.103.127.154 tcp 80 sticky
pool www_pool for WWW
```

generate the following PF rules:

```
table <www_pool> {}
rdr on xl0 proto tcp from any to 62.103.127.154 port 80 -> <www_pool> sticky-address
```

where `xl0` is assumed to be the name given by a preceding '`interface external xl0`' command.

On the other hand, if we would like to have service requests from the internal machines reflected, as is the case of LDAP in Section 2.1, then a `reflect` parameter is mandatory:

```
service LDAP is 62.103.127.157 tcp 389 reflect
pool ldap_pool for LDAP
```

The configuration above produces the PF rules:

```
table <ldap_pool> {}
rdr on xl0 proto tcp from any to 62.103.127.157 port 389 -> <ldap_pool>
rdr on xl1 proto tcp from 192.168.1.0/24 to 62.103.127.157 389 -> <ldap_pool>
nat on xl1 proto tcp from 192.168.1.0/24 to <ldap_pool> port 389 -> 192.168.1.1
```

where `192.168.1.0/24` is assumed to be the internal network defined by a preceding '`internal net 192.168.1.0/24`' command and `192.168.1.1` is the internal gateway defined with a prior issued '`internal gw 192.168.1.1`' command.
We should note that the generated PF rules are incorporated into the load balancing device using pfctl.

- ◇ `'add' IP 'to' POOL`
  We use this command to declare that the machine holding the specified internal IP, is able to handle the service corresponding to the designated pool. When the add command is interpreted, it updates the data model with the new IP/pool association. At the PF rules level, assuming that `service` and `pool` commands have been issued previously, we handle add by calling pfctl to insert the corresponding IP into the defined PF table.
- ◇ `'remove' IP 'from' POOL` or `'remove' SERVICE`
  `remove` does exactly the opposite of add.
- ◇ `'status' SERVICE|POOL|IP`
  This command provides reports for each specified entity.
- ◇ `'save'`
  This command is used to produce a valid, low-level packet filter configuration file; for PF under FreeBSD this file is `/etc/pf.conf`.

Help on each command can be given at any time from an interactive lbsh session:

```
>>> help interface
name: lb.interface
class: cmdInterface
usage: 'interface' 'internal'|'external' NAME [ifconfig params]
where: NAME: the network device name as understood by the os (e.g xl0 for FreeBSD)
```

3.2 Health Checking Component

Monitoring for the liveness of services, calls for a separate health checking component. In our implementation, this component, which we term *poller*, checks for the availability of specific services in the pools over regular time intervals. Whenever a service is detected to be either UP or DOWN, corresponding packet filter rules are executed in the network-device's kernel, so as to modify the routing of subsequent requests.

```
while True:
  choose the next IP and SERVICE to check, from all the pools
  call checktaskFactory to obtain a checkTask for the given IP and SERVICE
  call checkTask and obtain the service status
  call actionHandler with the produced status
```

Fig. 10. The scheduler generic algorithm.

The design of the *poller* is completely modular. In essence, all that is needed is the specification of the following objects:

pfDriver: Although we base our solution on BSD's PF, there is essentially no good reason to hard code this dependency in our implementation. What is actually needed is to abstract all the actions necessary to:
  - initialize the state of the underlying packet filter.
  - make the incremental (on demand) addition and removal of pool machines.

  We have achieved this by introducing the notion of a *packet filter driver* and by providing a first implementation based, of course, on PF. If someone wants to employ the architecture of our solution to a Linux-based setting, then one should, for instance, implement an iptables-based packet filter driver.

checktaskFactory: This object is responsible for using an appropriate protocol or method of some sort to check the availability of a specific service. For example, if we have a radius server, we can send a dummy authentication message, Access Request, to just check if the server is up and running. In effect, *checktaskFactory* is a factory of pluggable algorithms for checking services.

model: The model is the communication means between the shell and the poller and is described in more detail in Section 3.3.

actionHandler: The action handler performs specific actions on behalf of the poller, as soon a change in the status of a service is detected. In the current implementation, the default action handler updates the model with the new status and calls the packet filter driver, to update the underlying PF rules.

scheduler: The poller itself does not provide a specific polling algorithm. This is the responsibility of the scheduler object. The scheduler specifies an iteration algorithm over the servers in the various service pools; its job is described in the code snippet of Fig. 10.

3.3 Data Model

It is implied from the shell interactive session presented in Section 3.1 that the shell should be aware of the status of services in the farms involved. In this regard, a `status` command produces the requisite report. On the other hand, the service status is modified by the *poller*. This led us to introduce a separate entity which we name *data model*. In generic architecture terminology, this resembles a communication bus among several interacting components.

Our data model is hierarchical, represented as a tree of nodes. Each node is either a container and thus has children, or a leaf node. The tree root is termed `ROOT`. On each node we can store a set of attributes. For the purposes of our prototype implementation, we map this tree hierarchy directly on the filesystem, thus making the model persistent. Moreover, we implement atomic modification operations for attributes.

Table 1 shows the exact hierarchy we employ. The entries in the leftmost column are indented properly, to point-out parent-child relationship.

Node	Container?	Parent	attributes
`ROOT`	Yes		service_SMTP: {ip: ..., port: ..., sticky: ...} service_LDAP: {ip: ..., port: ..., sticky: ...} ... interface_internal: xl1 interface_external: xl0
`pools`	Yes	`ROOT`	
`mx_pool`	Yes	`pools`	service: SMTP
`192.168.1.2`	No	`mx_pool`	status: UP
`192.168.1.3`	No	`mx_pool`	status: UP
`ldap_pool`	Yes	`pools`	service: LDAP
`192.168.1.2`	No	`ldap_pool`	status: UP
`192.168.1.3`	No	`ldap_pool`	status: UP
`192.168.1.4`	No	`ldap_pool`	status: UP

Table 1. Hierarchical Data Model.

We would like to note the following for the adopted data model:

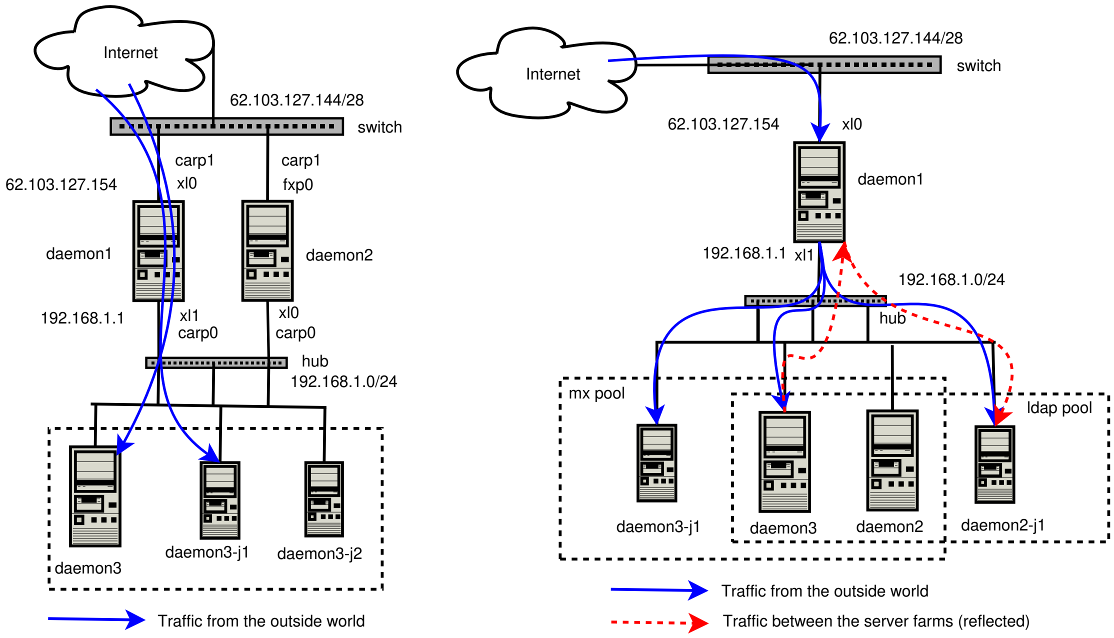
- We use the `ROOT` node to store information regarding the specification of services, as defined by the lbsh command `service`. We also store both internal and external interface definitions, as designated in the lbsh command `interface`.
- Each pool is represented by its exact name under the `pools` container node, which in turn resides under `ROOT`. There is exactly one attribute per pool tree-node which articulates the name of the service this pool was created for (using the `'pool POOL for SERVICE'` command).
- All the IPs that participate in a pool, are represented by leaf nodes under their parent pool tree-node. These leaf-level nodes have an attribute corresponding to their status.

4 System Evaluation

4.1 Experimental Test-bed Description

To thoroughly evaluate the pros and cons of our proposed approach, we created an experimental test-bed consisting of two networks; the public network is accessible in the IP range `62.103.127.144/28`, while the internal network uses IPs in the range `192.168.1.0/24`. We used three workstations with rather dated-hardware running FreeBSD 6.0 and we utilized the FreeBSD jail facility [14] that enabled us to simulate an environment with more than three nodes. The first machine, with hostname daemon1, is a Pentium III@850MHz with 256 MB memory, two 3Com 3c905C-TX network interfaces and has a single 40GB hard drive. The second workstation named daemon2, is a Pentium III@730MHz, with 256 MB memory, a 10GB hard drive and two network interfaces: an Intel 82559 Pro/100 and a 3Com 3c905C-TX. We also configured a jail environment on this machine, daemon2-j1. Finally, the third machine named daemon3, is a Pentium III@900MHz with 512 MB memory, 80GB disk space and a single Intel 82550 Pro/100 network interface. We created two jails on this machine namely, daemon3-j1 and daemon3-j2. Each jail appears as a separate host and is configured to provide smtp, ldap, http services running sendmail, openldap and apache respectively.

We have investigated two arrangements for our workstations: the first arrangement tests how well the failover behaves and the second stress-examines the provided services. Fig. 11(a) shows the first arrangement in which the daemon2-j1 jail was de-activated and daemon2 was placed in "in parallel" with daemon1. Host daemon1 plays the role of the primary gateway load-sharing *network-device*, while daemon2 was the hot-standby. Fig. 11(b) depicts our second experimental arrangement; we activated all jails and placed



(a) Setup 1 - Parallel gateway devices

(b) Setup 2 - A gateway device with server farms

Fig. 11. Different workstation arrangements in our experimental test-bed.

daemon2 in the internal network while having daemon3. This setup, provided a total of five nodes to be used in server farms. In both arrangements, the network-device undertook the role of authoritative nameserver for the entire ourdomain.tld domain to which all of our machinery belongs. In this context, our gateway network-device has to manage additional load caused by the name resolution process.

4.2 Test Scenarios

For the execution of our tests, we developed an smtp workload generation script. The script can simulate a number of concurrent *clients*, each of which sends a configurable *count* of mail messages with a specified *delay* between consecutive sends. The mail items dispatched, are rather small in terms of size –approximately 1kb in length– as we want to avoid over-stressing our dated hardware. Each mail is destined to an alias address (`authors@ourdomain.tld`) that `sendmail` has to resolve in ldap; so, *reflected connections* are generated from the mx pool of machines towards the ldap farm.

Simulation of failed service nodes: We arrange the workstations according to Fig. 11(b) with `daemon1` assuming the role of the network-device. There are two pools of servers: the first consists of `daemon2`, `daemon3` and `daemon3-j1` and provides the smtp service; the second contains `daemon3`, `daemon2` and `daemon2-j1` and offers ldap. We generated smtp workloads with a variable number of concurrent clients (4, 8, 16, 32, 64, 128). Each client sent 50 mails with 1 second delay between consecutive dispatches. We caused a failure in `daemon3-j1`'s sendmail. The *poller* notices the failure after a configurable time *interval* which, for the purposes of this test, is set to 3 seconds. We measured each machine's maximum average load and the number of failed smtp sessions against the number of concurrent clients. We present the results in Fig. 12.

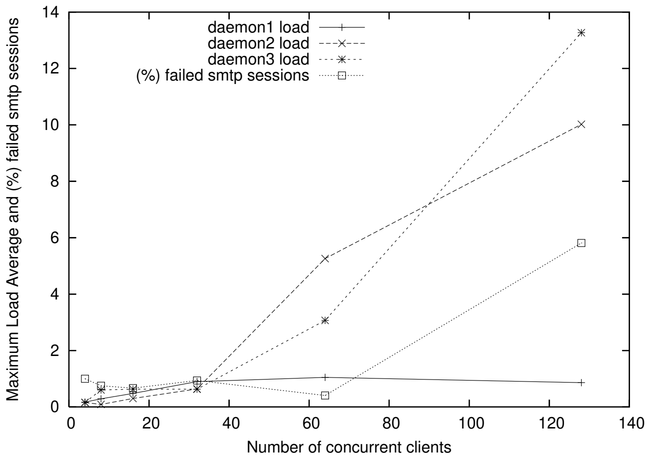


Fig. 12. Maximum load and (%) failed smtp sessions when simulating node failures.

We observe that the network-device handled the internal machine's failure gracefully, eliminating it transparently for the clients. The lost smtp sessions were caused mainly because of the load increase in the backend servers and not due to the network-device. The load after the internal machine's failure was balanced evenly among the remaining nodes.

Stress-Testing: Using our second arrangement (Fig. 11(b)), `daemon1` is the gateway load-sharing network-device, `daemon2`, `daemon3` and `daemon3-j1` are in the pool of mx mail-servers running `sendmail` and finally, `daemon3`, `daemon2` and `daemon2-j1` provide the ldap service. We generated an smtp-based workload with a variable number of concurrent clients (4, 8, 16, 32, 64, 128). Each client sends 100 mails with 1 second delay between consecutive dispatches. Our measurements include the maximum average load on the various machines (as shown by the UNIX command `uptime`), the number of created `rdr/nat` state entries in PF, the number of failed `smtp` sessions and finally the mean time for the completion of a single smtp session. Figures 13, 14(a) and 14(b) show our results.

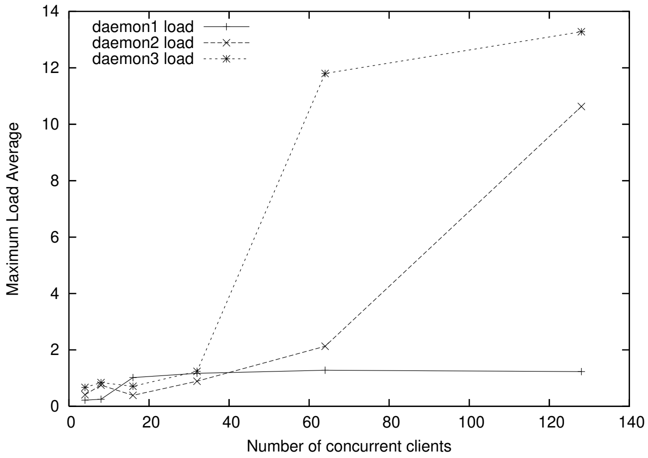
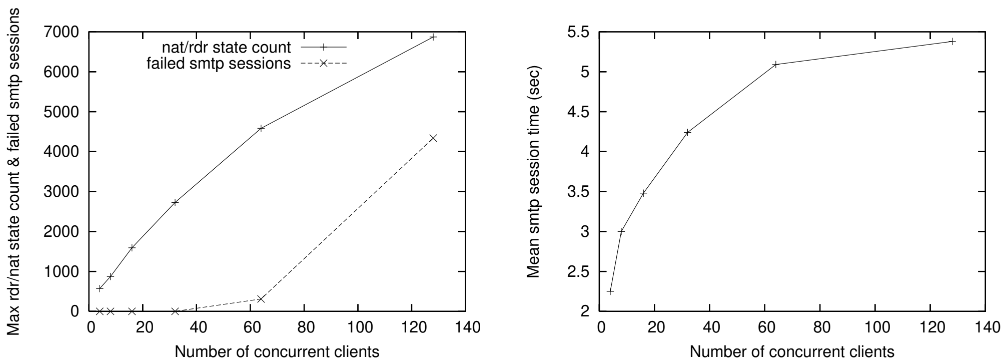


Fig. 13. Device (`daemon1`) and servers (`daemon2`, `daemon3`) load graph.



(a) Lost sessions and rdr/nat state count graph.

(b) Mean `smtp` session time.

Fig. 14. Graphs for mean smtp session time, lost sessions and rdr/nat state count.

In all cases, the maximum load of `daemon1` did not exceed 1.28, while PF state entries reached the number of 6872 with 128 concurrent clients. The default limit in PF is 10,000 entries. With 64 and 128 concurrent clients, the load of the servers in the two pools exceeded sendmail's threshold of service (10 by default in FreeBSD) and there were failed smtp sessions.

5 Related Work

Since the problems we address are common to a growing number of companies and organizations [5, 2], there exist competing solutions that are either open source or proprietary. In the open source arena, a prominent project is the Linux Virtual Server (LVS) [12] whose goals are *"to build a high-performance and highly available server for Linux using clustering technology, which provides good scalability, reliability and serviceability"*. There is also a port of LVS to FreeBSD. The aforementioned project's site contains a great deal of useful information about load balancing and different approaches in dealing with the issues involved. The Linux High Availability Project [11] provides heartbeat as a core component that can be used with technologies from LVS to provide high availability solutions in machine clusters. Although the project has the term Linux as part of its name, it is portable and FreeBSD is a supported platform. Apart from open source solutions, there are commercial products [1, 3] from a number of vendors that partially attack the problem we addressed in this paper. Little is known other than their published interface, for the architectural organization of these products as they remain proprietary.

6 Summary and Future Work

In this paper, we propose a reflective, load-balancing, and highly available architecture based on commodity hardware and FreeBSD and whose main objective is to deliver very reliable and 7x24-available Internet services. Our approach is based on farms of computing nodes that implement services such as http, ldap, smtp, radius, and dns; these nodes are coordinated by a network-device that acts not only as the nameserver but also as the coordinator and load-balancing element of the networked resources. We outline our overall architecture organization and discuss our main design choices. We offer administrative and coordination support via lbsh, a flexible and effective shell. We finally provide an experimental evaluation of our prototype system.

By utilizing lbsh's flexible software architecture, there are a few features we would like to add to our work as future extensions:

- Incorporate the failover functionality provided by carp and pfsync to the shell's commands. This configuration takes place outside of the shell environment at present.
- Introduce a privileged mode of operation in lbsh that provides functionality to view/edit low level PF rules (for use by power users).
- Load/save the device configuration from/to a remote storage, with tftp or sftp.
- Provide a web interface that leverages the shell.
- Implement optimizations in the poller component.
- Explore the possibility of using SNMP in our architecture.

– Add support for CIDR network blocks in lbsh commands.
– Investigate the possibility of adding support to lbsh for other open source packet filters that have network address manipulation capabilities.
– Introduce different load-balancing algorithms in the distribution of connections to servers. This is something that requires work at the PF level as well.

Acknowledgments: we are very grateful to Pantelis Papanikolaou and Alexander Bassakidis of OTENET for their active and ongoing support as well as several colleagues at OTENET for informative discussions. This work was also partially supported by a grant from the University of Athens Research Foundation.

References

1. Cisco CSM Architecture White Paper. http://www.cisco.com.
2. Cisco White Paper: Business Case for Global Server Load Balancing. http://www.cisco.com.
3. F5 Networks BIG-IP product. http://www.f5.com/products/bigip/.
4. Hot Standby Router Protocol. http://www.ietf.org/rfc/rfc2281.txt.
5. Open 24 Hours: Load Balancing. http://www.f5.com/communication/articles/2005/article040505.html.
6. Virtual Router Redundancy Protocol. http://www.ietf.org/rfc/rfc3768.txt.
7. Jacek Artymiak. *Building Firewalls with OpenBSD and PF.* devGuide.net, 2nd edition, 2003.
8. Firewall failover with pfsync and carp. http://www.countersiege.com/doc/pfsync-carp/.
9. Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, NY, 1982.
10. The FreeBSD Project. http://www.freebsd.org.
11. The Linux High Availability Project. http://www.linux-ha.org/.
12. The Linux Virtual Server Project. http://www.linuxvirtualserver.org/.
13. Christos K.K. Loverdos. The oyster Project. http://www.di.uoa.gr/~loverdos/oyster, 2005.
14. FreeBSD man pages. http://www.freebsd.org/cgi/man.cgi.
15. The OpenBSD Project. http://www.openbsd.org.
16. PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/index.html.
17. The Python Programming Language. http://www.python.org.