backhand

use your resources

# UNDERSTANDING AND BUILDING HA/LB CLUSTERS

MOD_BACKHAND, WACKAMOLE AND OTHER SUCH FREEDOMS

THEO SCHLOSSNAGLE <jesus@omniti.com>

SEPTEMBER 12TH, 2002

## Abstract

*Today's web applications are more demanding than ever. Many companies rely on their online applications for internal productivity and revenue generation. Faced with 24/7 uptime requirements and strict service-level agreements for performance as well as reliability, today's online applications require higher availability and smarter resource allocation than ever before.*

*We will discuss in detail a variety of high-availability, content-intelligence and load-balancing solutions that are commonly employed by today's leading sites, focusing on the correct application of each technology, the differences between them, as well as dispelling common misconceptions regarding these terms.*

*The discussion will bridge both open-source and proprietary technologies and will include detailed tutorial-style examples for various open-source components from the Backhand Project. This paper aims to give you all the tools necessary to assemble your own highly available, content-intelligent, resource-balanced application cluster complete with the skills to create custom load-balancing logic to best handle the idiosyncrasies of your specific architecture.*

## 1 Definitions

*”My site can’t go down, and it is too much work for one machine.”* In order to devise a successful solution, we must have a satisfactory definition of the problem. When most people think of ”HA” they also think of ”LB,” when they have very little to do with one another. This mistake is, however, understandable due to commercial clout on the subject. Most commercial solutions provide HA/LB in a single product, and while this may be convenient, we will investigate the shortcomings of this approach. Separating HA and LB is vital to understanding the advantages and disadvantages of their various implementations.

| | |
|---|---|
| **Cluster** | (*n.*) a bunch; a number of things of the same kind gathered together. in computers, a set of like machines providing a particular service united to increase robustness or performance. |
| **HA** | High Availability (*adj.*) remaining available despite the failure of one’s components. usually used to describe a system or a service. above and beyond fault tolerant. |
| **LB** | Load Balanced (*adj.*) providing a service from multiple points where resources are equally utilized. |

## 2 High Availability

In the most fundamental approach, to make a cluster of web servers highly available we simply must ensure that the failure of a single web server will not sacrifice the availability of the service provided from that cluster. Simply put, if one machine crashes, no one notices.

There are two basic approaches to high availability in common usage. One addresses two machine failover and the other addresses HA in larger clusters.

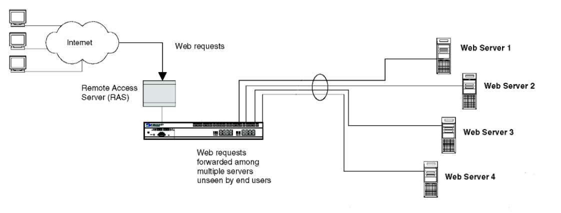
### 2.1 Pair-wise failover

Two-machine failover can be implemented trivially with heartbeart style programs. Two machines are prepared to provide the same service, and a heartbeat is established between the machines. One machine is designated active and the other standby. If the standby machine detects a failure in the heartbeat, it assumes active control of the service. There are several available heartbeat solutions on the market and in the open source world. These methods that inherently preclude their usage in $N$ node configurations are antiquated. Wackamole provides a more generalized solution that will be described in further detail later.

### 2.2 Front-end failover device

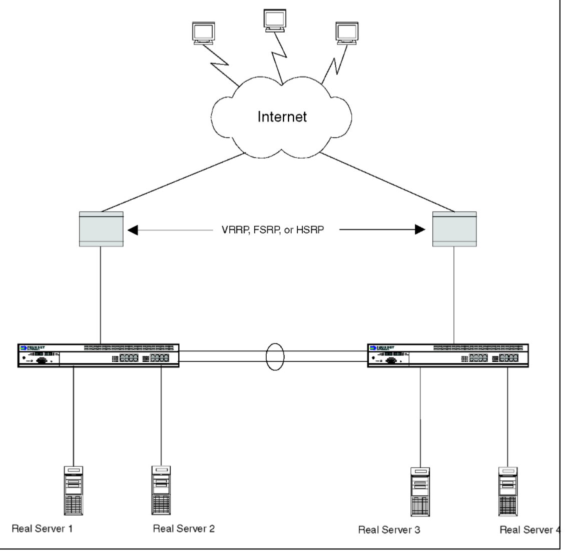
The multi-machine availability solution is more appropriate for this discussion. This is the single-point transaction distribution device. The device is placed between the cluster of machines and the clients performing transactions. The device provides a virtual IP address, and all transactions with an endpoint of that IP will be connected instead to one of the machines in the cluster. The device monitors the cluster machines in some fashion and will not attempt to connect clients to a machine that is unavailable.

**Single-point transaction distribution device**


Internet
Web requests
Remote Access Server (RAS)
Web requests forwarded among multiple servers unseen by end users
Web Server 1
Web Server 2
Web Server 3
Web Server 4

Upon a moment of inspection, the apparent flaw of the single-point transaction distribution device is evident. How does one make the HA device highly available? If a single distribution device is used to provide this service, what happens to the overall architecture when it fails? All (or sufficiently many) commercial solutions provide the above described two-machine failover solution between two of their identical devices. Problem solved? Yes. End of discussion? No.

**Failover transaction distribution device**



The above described solution requires the purchase and integration of two specialized pieces of machinery. There are a plethora of reasons why this is frustrating and often even unacceptable. While these reasons have varying degrees of applicability dependant on the situation, most likely one applies. A short list of reasons could inlcude:

- a single HA device is expensive, let alone two,
- two more components to manage, which require additional expertise to be on-call 24/7,
- introduces a new bandwidth limitation,
- occupies cage/cabinet space that could be otherwise occupied by web servers.

To speak only of an architecture's shortcomings is to do it a disservice. The transaction distribution device is an elegant and simple solution. Due to its common usage in production configurations, the implementations tend to be well-designed, well-tested, stable and functional.

While all available commercial solutions tout load balancing as their main function, and while that is arguable, most *do* provide rock-solid high availability. A list of solutions that exist in this category would include: LVS (Linux Virtual Server), Foundry's ServerIron, F5's BIG/ip, Cisco's LocalDirector and Arrowpoint product line, and Alteon web switches. All of these devices work on the network level with only casual inspection of the actual nature of the transaction. This allows them to be extremely efficient high-availability devices.

However, this operational approach inherently inhibits their ability to load balance. The black-box nature of these products can frustrate attempts to implement an intelligent content allocation scheme.

## 2.3 Peer-based high availability

To say that front-end HA devices are ”the wrong approach” is a strong sweeping statement, and we won’t make such rash generalizations here. However, asserting that it is ”the right approach” in every architecture is pretentious. While peer-based approaches are still considered by many to be schismatic, a paradigm shift in thought often solicits an unorthodox design and implementation.

The concept of peer-based technologies is not complicated. Each machine in the cluster that provides a service is responsible for making itself available. Of course, this is the ”mission” of any machine in any production environment. However, in a peer-based HA system, each machine has some added responsibility. Any given machine must be capable of assuming the functional responsibilities of any other machine in the cluster in the event of a failure.

Today’s clusters are often heterogeneous in configuration as machine addition, upgrade, and retirement cycles never align completely. However, the vast majority of clusters are homogeneous with respect to their function. Each machine within the cluster serves the same purpose. This is integral to being capable of assuming the service responsibilities of a fellow machine. In an Apache web-clustering configuration, each machine would need to be running a functionally identical Apache installation. If the machines are functionally equivalent, it directly follows that any machine is able perform the function of a fallen comrade.

The technology involved in peer-based HA solutions consists of both the technique employed to decide when and by whom a machine’s responsibilities need to be assumed and the mechanics that drive the actual transference of responsibility. This peer-based system, as a distributed system, suffers from the most fundamental problem that plagues all distributed systems: it is hard to design and implement while maintaining correctness. This is the main reason for the lack of alternative implementations that use this approach in their design.

As a peer-based HA solution is a fundamentally divergent approach than a single front-end HA device, it embodies a distinctly different set of advantages and disadvantages. Perhaps the most apparent difference is the presentation of service. With a front-end device, a service is presented over a single IP address through which connections are distributed over a set of back-end machines. Peer-based HA systems separate the functions of HA and LB completely. Assuming that LB and resource allocation is not being performed, a peer-based system must present the service over a number of IP addresses greater than the number of machines. This is typically accomplished with DNS round-robin. Very basically, if a machine in a peer-based system wishes to do work, it must be available to receive queries. With the addition of a load balancing technology, a multi-tier architecture is possible with only a single IP advertised for a given service.

N-way failover mechanisms are inherently more fault tolerant than 2-way failover systems for all N greater than 2. We can suffer N-1 failures and the service as a whole will remain available. This robustness combined with freedom from dedicated HA devices is a clear advantage.

On the side of disadvantages, the most prominent is its restrictive nature in clusters whose purpose it is to serve secure HTTP (SSL) traffic. HTTP over SSL requires an immediate client-server SSL negotiation. Unlike TLS enabled SMTP, which allows you to upgrade an unencrypted and unauthenticated session after some simple interactions, HTTP requires the server to present the SSL certificate immediately. This does not allow the client to indicate to whom it wishes to talk (via a `Host:` header or the like), which means that only one named host can be served over any given IP (on port 443[1]).

If the front tier of machines in the cluster wish to each expose themselves to client-originating requests (which is necessary if all machines are to contribute and the front-end machines are not capable of distributing requests over the other machines in the cluster), then each must have an IP address, and those addresses need to be exposed via DNS. Adding this up yields a single hostname pointing to multiple IP addresses, each of which must be functionally equivalent and therefore must produce the same SSL cert upon connection. If your goal is to serve 250 SSL sites through your cluster, it only makes sense to expose a single VIP per SSL service.

This disadvantage isn't a fundamental flaw in an HA system. As we discussed, there are many HA systems, but to be effective, they themselves must not be a single point of a failure. A peer-based HA system can be used as a complement to eliminate the single-point of failure that exists in many front-end HA systems. The sole purpose of any HA solution is to eliminate a single point of failure, we must not loose site of that objective or confuse it with load balancing.

## 3 Wackamole: a peer-based HA implementation

Wackamole[2] is an implementation of the above described peer-based high-availability technology. It resides on all of the machines in a cluster and ensures that they collectively represent all of the IP addresses advertised for the service.

### 3.1 Design

Wackmole is designed to run on a set of machines providing a unified service over an IPv4 network. The design only guarantees that the set of IP addresses that are sponsored by the cluster will be completely ”covered” completely by the available machines in the configuration. When a wackamole instance fails, the other wackamoles shortly become aware of this change and elect a machine in the current working set to take each IP address that had been controlled by the failed machine prior to its failure.

A group communication (GC) service provides information to its subscribers concerning membership and provides a communications bus over which messages can be sent. These message are delivered is a fasion that allows sender to trust the order of delivery and the receiver to know the exact membership of the group at the time of message delivery. Wackamole uses this GC service to determine availability and to trasfer state information.

Each wackamole instance connects to the GC service and joins a common, previously determined management group. In the event of a system fail-

[1]Port 443 is assigned to HTTPS (Secure hypertext transfer protocol.

[2]wackamole is available `http://www.backhand.org/wackamole/` and is a product of the Center for Networking and Distributed Systems at the Johns Hopkins University

ure, the GC service on that machine will become unavailable. This unavailability will trigger a change in the active membership of this management group will change and each member will be notified of this change. The GC system provides a consisting view to all members in the same working set. This means that wackamole can simply act on their current persepctive of the active membership – as the GC system guarantees that the other members see the same thing.

This membership state is used as a moment to moment ”roll call.” Wackamole makes an immediate decision on which member in the current membership will take which IP addresses in the cluster's configuration.

The GC system is used to propagate the current state of the cluster – which machine is advertising which IP address(es). Based on this state information and a consistent view of the current membership, each wackamole can make the fast, accurate decisions on who will release/acquire a given IP address.

## 3.2 Implementation

Wackamole is written in C for UNIX and there is a clean separation between application logic and platform dependent plumbing. Combined with the use of autoconf, this provides a proven framework for multiplatform support and easy porting to new platforms.

For a group communication system, wackamole relies on Spread. Spread is supported on several UNIX platforms and its use of autoconf allows for easy porting. Spread is a group communication system provided as a system service. A Spread daemon is run and client application connect to it over either a unix domain socket or a TCP/IP socket to communicate.

For wackamole, we wish to have a Spread daemon run on every machine in the cluster and connect to it only via unix domain sockets. This guarantees us that there is no IP component between the client and the GC daemon and it guarantees that the GC daemon is actually running on the machine that provides the service to be made available.

As wackamole is designed to be a high availability system for IPv4 services, it is important that we don't rely on IPv4 for the application itself. This ensures that no IPv4 problems can influence that mechanism of Wackamole. It is also vital that the Spread system use the same ethernet network and IPv4 subsystem for inter-daemon communications so that any changes in the IPv4 availbility of the machine immediately impact the visibility of the Spread daemons with respect to each other; thus impacting the group membership view it presents to Wackamole.

Wackamole is given a set of virtual interfaces (VIFs) to manage, this list of VIFs must be consistent in both the IP address(es) the represent and their ordering. Wackamole operates on this array of VIFs by index making sure that each index in the array is actively represented by at least one and at most one participating machine.

Wackamole's implementation allows for the cluster to consist of completely heterogeneous machines, differing in speed, power, architecture and even platform. Each machine must be capab le of assuming all the responsibilities of the IP addresses advertised by the service.

When the first machine in a cluster comes online, that Wackamole instance is responsible for all of the VIFs in the configuration. When the next machine comes online, there are no IPs that **need** taking as they are currently being handled by the first machine. However, this sort of inbalance is undesirable. Some balancing of IP addresses, while not required, is a feature that brings a tremendous benefit. Assuming a DNS round robin configuration where all IP addresses are advertised to the public, by balancing the IP addresses managed by the clustered machines we effectively balance the connection rate sustained to each individual machine from end clients.

The actual mechanics to releasing control of a VIF is quite platform specific, but usually entails simply making a few `ioctl()` calls in order to down the virtual IP address(es). On the other hand, acquiring a VIF is more complicated. Not only do we perform the necessary system calls to plumb and instantiate the virtual IP address(es) associated with a VIP, we also need to ensure that everyone else on the network is aware of the new home of those IP addresses. On ethernet networks, this is accomplished by a technique called arp-spoofing.

ARP (address resolution protocol) is the protocol used on ethernet networks to resolve IP addresses to ethernet MAC addresses. ARP is initiated by a machine making an IP connection to another machine on the same subnet that do not have the IP in question in the local ARP cache. One the ARP request as been made, the response is waited for and upon reception, the MAC address to IP address mapping is stored in the system's local ARP cache so that subsequent ARP interactions regarding that IP address do not require network activity.

This cache is the root of problem that plagues and IP failover solution. When an IP address is released by one machine and acquired by another, the MAC address will change. The immediate concern is all of the machines that have the now incorrect MAC address in their ARP caches. This is where arp-spoofing comes into play. If an unsolicited ARP response is received by a machine, the new association is replaced into the systems ARP-cache. In order to perform a successful IP acquisition, we need to establish the new IP address on that machine and then annouce this to all of the necessary machines as unsolicited ARP responses.

Now, which are the necessary machines? The first obvious machine that needs to be arp-spoofed is the router. By doing this, we satisfy availability to all machines on other subnets (as they pass all traffic though the router). However, this doesn't handle machines on the same subnet – and yes, these machines are important too! So, how can decide which machines these are? These machines are all machines on the network that contain in their ARP cache the IP address just acquired. However, we don't have access to these machine's ARP caches.

Most IP traffic is part of a two-way interaction, so it stands to reason that all machines that have said IP address in their ARP cache will exist in the ARP cache of the previous owner of said IP. Obviously there is a problem with retrieving the content for the ARP cache (or any other information) from a machine that has just failed. Wackamole provides a mechanism for periodically distributing the contents of the the ARP cache local to each instance. These caches are aggregated for a cluster-aggregate ARP cache. This cluster-aggregate ARP cache can be used by wackamole as a destination for arp-spoofing.

## 3.3 Why a VIF and a VIP are different

It seems at first glance that a VIF and VIP would be strictly 1 to 1. In a web cluster of Apache servers, this logic holds. All VIFs of an Apache cluster will most likely consist of only externally addressable virtual

IP addresses. In this case, a VIF only adds the physical interface on which the VIP is instantiated.

There are specific cases when different IP addresses in different subnets are intrinsicly related and must be considered a single unit. The most obvious case is that of a router. A router must possesses an IP address on all subnets over which it intends to route. In this case, a VIF consists of a set of inseparable set of physical interface/virtual IP address pairs.

## 3.4 Configuration

There are several important configuration options for Wackamole.

### 3.4.1 Spread settings, operation and control

**`Spread`** specifies the location of the Spread daemon. A number like 4803 should be specified to connect to the locally running Spread daemon. 4803 is the default port that Spread uses, if you have installed it differently the Spread daemon should be reflected here.

**`SpreadRetryInterval`** specifies the amount of time between attempted but failed reconnects. If the Spread connection is lost of any reason (including initial start up) Wackamole immediately attempts to make a connection to the Spread daemon specific by the `Spread` configuration option. If this connection attempt should be unsuccessful, Wackamole will reattempt after this specified time interval.

**`Group`** specifies the Spread group that should be joined. This group should only be used by Wackamole instances in the same cluster.

**`Control`** specifies the location of the Wackamole control socket. Wackamole creates and listens on a unix domain socket. The utility program `wackatrl` that comes with Wackamole connects to the Wackamole daemon connect via this control socket.

**`Mature`** specifies the time interval required for the instance to mature. Maturation describes the a change from a joined member to a member capable of assuming VIFs. This can specified in order to prevent machine flapping causing service flapping. A time interval of 5 seconds (5s) is a reasonable value.

**`Arp-cache`** specifies the time interval between recurrent ARP cache updates. Every $n$ seconds, Wackamole will peruse the system's ARP cache and announce all the IP address contained therein to the Spread group. All members of the group will receive this list and integrate it into the shared ARP pool that can be used as a source for the **Notify** list.

### 3.4.2 The `Balance` section

**`Interval`** specifies the length of a balancing round.

**`AcquisitionsPerRound`** specifies the number of VIFs that can be acquired be round. After each round, this number of VIFs are acquired by machines that are still out of balance. In a cluster with 10 VIFs with one active machine, when a second machine matures it will attempt to balance with the VIFs/machine. The two machines will mutually decide that the recent joiner should acquire 5 of the VIFs. Is `all` is specified here, then there is no limit to the number of VIFs that can be shifted in a single round and the balancing will all happen immediately. If a number $n$ is specified, Wackamole will reorganize the ownership of VIFs such that no machine acquires more than $n$ VIFs will be acquire by this instance per round.

### 3.4.3 The `VirtualInterfaces` section

This section defines all of the virtual interfaces that the cluster is to be responsible for. A virtual interface can be of the following forms:

1. `ifname:ipaddr/mask`
2. { `ifname:ipaddr/mask` }
3. 
```
{
ifname:ipaddr/mask
ifname:ipaddr/mask
...
}
```

ifname should be the name of the ethernet interface as seen from `ifconfig -a` without any virtual numbering. On operating systems like Linux, addition interfaces are labeled as `eth0:1` and `eth0:2`. In this case, only `eth0` should be used as Wackamole will dynamically determine an available virtual interface number on which to instantiate the given IP address. The mask is the number of set bits in the netmask, so that the ip address is in CIDR form.

This list of VIFs is stored in an array by wackamole and the ordering matters. Every Wackamole instance in the cluster must have the same VIFs in the same ordering. The only reasonable difference in configuration is the interface name as different machines may have different ethernet cards or be on a platform with differing nomenclature.

**Note:** on most operating systems the first IP address in a given subnet should be configured with the appropriate netmask and broadcast address for that subnet. However, all subsequent virtual IP address in the same subnet should be given a 32 bit netmask (255.255.255.255).

### 3.4.4 The `Prefer` directive

This directive is used to mark specific addresses in the virtual interface list as ”preferred.” When an address is marked as preferred for a given instance, it is guaranteed that if this machine is operation, it will acquire these addresses. It is important that more than one machine not mark the same addresses as preferred.

`Prefer` takes a single IP address as an argument, or a list of IP addresses in braces. These IP addresses are in CIDR form and though the netmask is not used, it is recommended that it match the mask used in the `VirtualInterfaces` section. If the a VIF has multiple IP addresses associated with it, the first IP address in the association should be used here. The argument `None` signifies that the machine should allow Wackamole decide which VIFs to acquire. Using an argument of `None` is recommended unless a configuration specifically requires a master/slave configuration.

### 3.4.5 The `Notify` secion

The notification section describes which machines should be sent arp-spoofs when an IP address is acquired. IP addresses are specified in the same fasion as in the `VirtualInterfaces` section – `ifname:ipaddre/mask`. The CIDR form IP is not treated as an IP address, but rather an IP network. The address 10.0.0.0/16, signifies that the 65536 address in the range 10.0.0.0 through 10.0.255.255

should be notified. A single IP address is can be specified by using a mask of /32.

In addition to a CIDR form IP, the special token `arp-cache` can be specified. If this is used, every IP address in the shared ARP cache pool described above is arp-spoofed.

As these IP spaces could be potentiall large, it is likely that sending all those arp-spoofed packets will flood the network. Thus, the ability to throttle the rate at which these packets are sent is needed. Any address (including `arp-cache`) can be followed by `throttle` $n$, where $n$ is the maximum number of ARP response packets to be sent per second.

## 3.5 Sample configurations

In this section we will discuss a few sample `wackamole.conf` files used in production environments.

### 3.5.1 Linux Apache servers

We have four Linux servers running Apache for a service providing image serving for a large web site. We have three virtual IP addresses allocated for this service and all are exposed via DNS for the name `images.example.com`. These IP addresses are: 192.168.12.15, 192.168.12.16, 192.168.12.17, and 192.168.12.18. The defatul route for each machine is 192.168.12.1. Spread is installed on all three machines and run on port 4803. The machines only have one interface in each and it is labeled `eth0`. The three machines have the permanent IP addresses 192.168.12.201, 192.168.12.202, 192.168.12.203, and 192.168.12.204, respectively.

The `wackamole.conf` on each machine will appear as follows:

```
Spread = 4803
SpreadRetryInterval = 5s
Group = wack1
Control = /var/run/wack.it

Mature = 5s
Balance {
  AcquisitionsPerRound = all
  interval = 4s
}
Arp-Cache = 90s

Prefer None
VirtualInterfaces {
  { eth0:192.168.12.15/32 }
  { eth0:192.168.12.16/32 }
  { eth0:192.168.12.17/32 }
  { eth0:192.168.12.18/32 }
}

Notify {
  eth0:192.168.12.1/32
  arp-cache
}
```

### 3.5.2 FreeBSD routers

We have two FreeBSD machines with three physical interfaces each. Spread is installed on each running on port 3777. They are plugged into the following subnets 63.236.106.104/28, 66.77.52.0/24 and 10.77.52.0/23. The machines are responsible for routing information between all of the networks and providing NAT services for machines operating on the 10.77.52.0/23 network. In order to perform the task, a single machine must possess all of the correct default routes, which in this case are: 63.236.106.102, 66.77.52.1 and 10.77.52.1. Each machine has real IP addresses other than the default router IPs which are ”virtual” in this scenario. The goal is that if any machine is alive, at most one will possess all three virtual router IP address and perform routing functions.

The `wackamole.conf` on each machine will appear as follows:

```
Spread = 3777
SpreadRetryInterval = 5s
Group = wack1
Control = /var/run/wack.it

mature = 5s
balance {
  AcquisitionsPerRound = all
  interval = 4s
}
arp-cache = 90s

# There is no "master"
Prefer None
VirtualInterfaces {
  {
     fxp2:10.77.52.1/32
     fxp1:66.77.52.1/32
     fxp0:63.236.106.102/32
  }
}

Notify {
  # Let’s notify our upstream router:
  fxp0:0.0.0.0/32
  fxp0:255.255.255.255/0
  fxp0:63.236.106.97/32
  fxp0:63.236.106.98/32
  fxp0:63.236.106.99/32
  # And out DNS servers
  fxp1:66.77.52.4/32
  fxp1:66.77.52.5/32
  fxp2:10.77.52.18/32
  fxp2:10.77.52.19/32
  # And the ARP cache
  arp-cache
  # The networks we are attached to.
  fxp0:63.236.106.96/28
  fxp1:66.77.52.0/24 throttle 16
  fxp2:10.77.52.0/23 throttle 16
}
```

## 4 Load balancing

Load balancing is a fundamentally complicated issue and it isn’t simplified by the chronic misrepresentation and confusion instigated by commercial propaganda. Most often, load balancing is used interchangably with request distribution. If a system is capable of distributing requests over a cluster of machines, it is performing load balancing. However, simply ditributing requests doesn’t mean that they are being effectively balanced. Load balancing is a resource allocation problem and resource allocation problems are hard.

Specifically in web systems, where requests are fast arriving and extremely short lived, determining the best machine to service a request is a challenge. As we do not have information about the future, it is theoretically impossible to make the optimal decision. Instead we have to make a best effort to assign requests to machines to most effectively utilize their resources. Most products use adhoc approaches to this as they don’t have specific details concerning the available resources on each machines.

There isn’t anything necessarily wrong with naive resource allocation techniques used by most commercial products. The reason for this is that most sites don’t really need intelligent resource allocation and the reason more than one machine is employed is for availability reasons rather than to handle excessive load.

In the event that the load incurred by client requests exceeds the capacity for a single machine, it needs a load balancing, but deserves an intellegent resource allocation framework. In order to perform truly intelligent resource allocation, a system must have detailed information about the request being made as well as the the available resources on all of the machines contributing horsepower to the cluster.

## 5 What is mod_backhand?

mod_backhand is a module for the Apache web server. It provides several facilities that enable load-balancing of HTTP requests over a cluster of machines:

- allocation of inbound requests to peer machines within a cluster. The allocation can happen via an HTTP redirect or via internally proxying the HTTP request.
- collection and distribution of resource statistics for machines within the cluster including memory utilization, CPU utilization, system load, and much more.
- an infrastructure to make allocation decisions based on cluster-wide resource utilization information, as well as information in the request itself.

Although these are the keys of mod_backhand's infrastructure, its flexible, extensible and efficient implementations supersede these simple components.

Given the resource utilization information in the cluster, intelligent decisions can be made as to where a request should be allocated. These decisions can account for clusters that are heterogeneous not only with respect to resources, but to platform and architecture as well.

## 6 An implementation overview

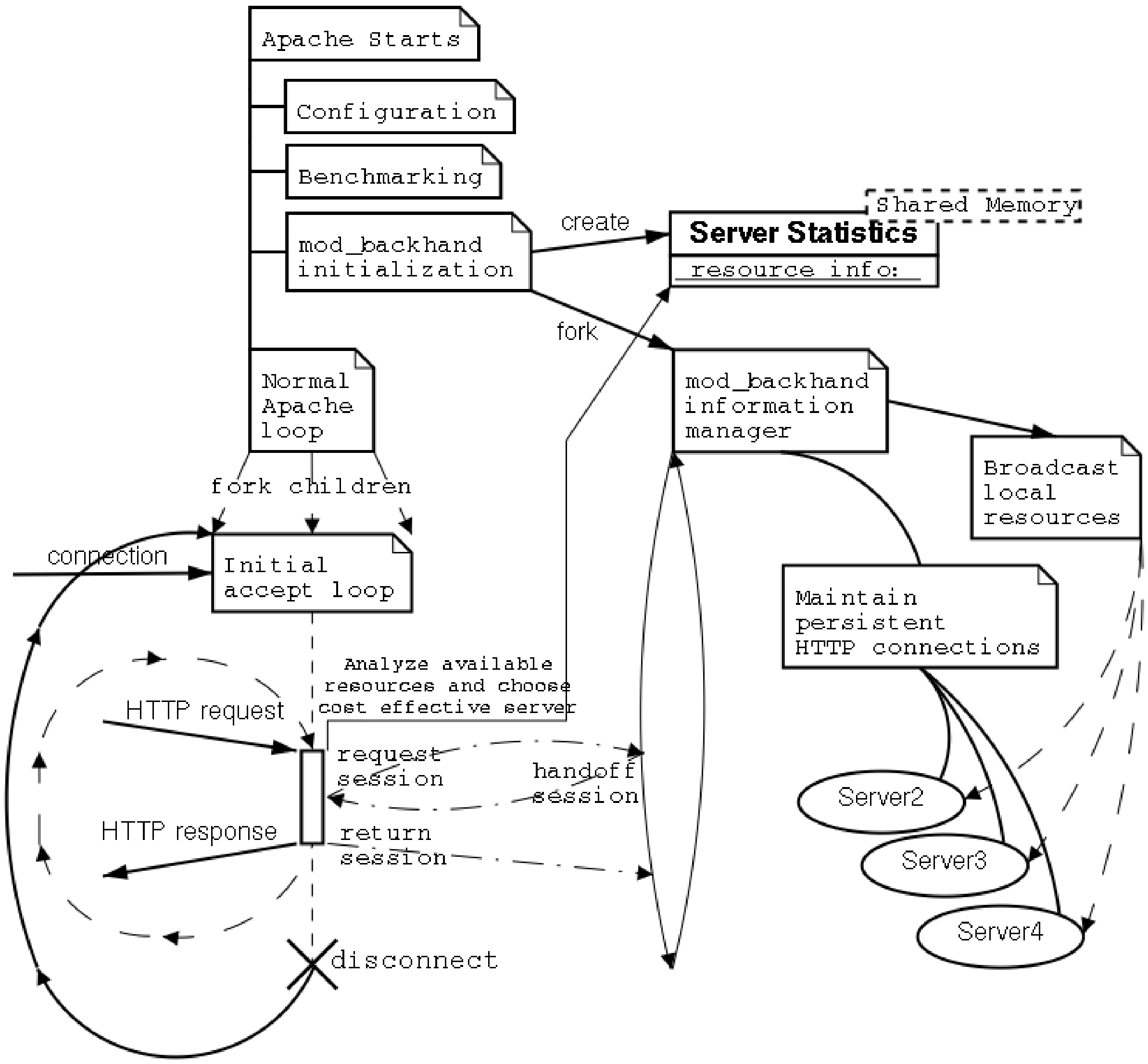
Apache is a complicated beast. mod_backhand attempts to integrate with Apache, disturbing as little of the rest of the system as possible. The Apache module interface supplies all of the necessary hooks to intercept requests and reroute them if necessary.

Apache does not, however, provide a built-in resource collection mechanism flexible enough to store the cluster-wide resource information that we need as input to our decision-making algorithms.

mod_backhand starts a separate moderator process at module initialization that will be responsible for several tasks, one of which is resource collection and distribution. This process also manages connection pools to peer web servers in the clusters.

When a request first arrives, mod_backhand first determines what candidacy rules to apply to the URI. This is defined in either a <Directory>, <Location>, or <Files> statement in the httpd.conf or an .htaccess file. If the request is to be "backhanded," then the request and the resources are provided as inputs to each candidacy function in the order specified. Each candidacy function is allowed to reorder and change the cardinality of the set of candidate servers and change the method of request reallocation (either HTTP redirect or proxy.) After the last candidacy function executes, the first server in the resulting list is chosen as the server to which we will reallocate the request.

The mod_backhand redirection handler is then invoked and the reallocation occurs. If the reallocation method is specified to be HTTP redirect, mod_backhand immediately issues an HTTP redirect and releases control back to Apache. If the method is set to proxy, then more work is done.



**mod_backhand implementation overview flow diagram**

The Apache child handling this request asks the moderator for an open file descriptor to the server to which it would like to proxy the request. The moderator hands an open file descriptor (creating and connecting first if necessary), to the Apache child, and the child attempts to proxy the request. The request is automatically upgraded to an HTTP protocol that supports keep-alives, if necessary. This ensures that pooled connections will have several pipelined HTTP requests during their lifetime even if the connection to the client doesn't want or support them.

The transparent upgrading of HTTP sessions from the front end to the back end allows for a more streamlined operation. If mod_backhand servers are configured as a front tier (HTTP-accelerator), then a considerable performance advantage can be seen when compared to basic HTTP proxies. The building and tearing down of TCP/IP sessions is avoided almost completely, and thus resources are conserved on the second tier web servers.

## 7 Decision making

The decision-making algorithms have access to both resource utilization information and the details of the request in question. The algorithm will analyze these inputs and augment a list of "candidate" servers that can satisfy the request.

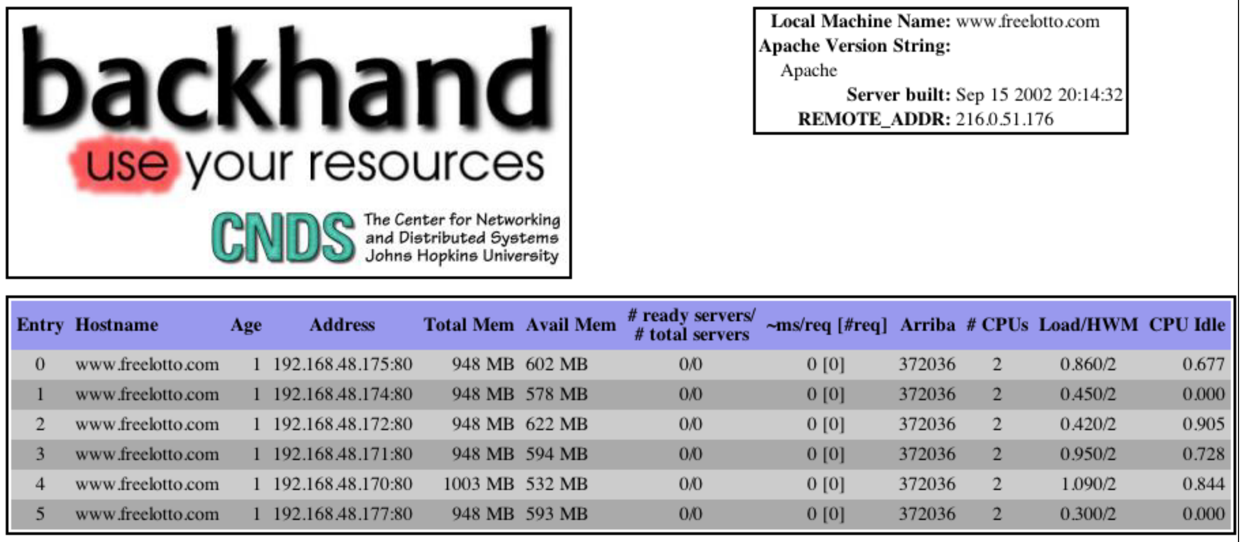
### 7.1 Resource information

Let us assume we have 10 servers. They are called www-0-1, www-0-2, ..., www-0-10. These servers all run mod_backhand and thus broadcast their resource information over the network. Each server should have an identical or close to identical view of the cluster-wide resources that are available at any given point in time. These statistics are held in a shared memory segment on each machine in a similar fashion to the Apache scoreboard. This shared "serverstats" structure is created at module initialization and is continually populated by the moderator process. As it is created at module initialization, all of the Apache children are able to see it – they inherit the attachment.

The serverstats structure is a `MAX_SERVERS` element array of `struct serverstats`. As of release 1.1.1, this `struct` looks as follows:

| struct serverstats |
|---|
| char hostname[40]; |
| time_t mtime; |
| struct sockaddr_in contact; |
| int arriba; |
| int aservers; |
| int load; |
| int load_hwm; |
| int cpu; |
| int ncpu; |
| int tmem; |
| int amem; |
| int numbacked; |
| int tatime; |

**Sample output of a <Location> with "SetHandler backhand-handler"**


backhand
use your resources
CNDS The Center for Networking and Distributed Systems Johns Hopkins University

**Local Machine Name:** www.freelotto.com
**Apache Version String:**
Apache
**Server built:** Sep 15 2002 20:14:32
**REMOTE_ADDR:** 216.0.51.176

| Entry | Hostname | Age | Address | Total Mem | Avail Mem | # ready servers/ # total servers | ~ms/req [#req] | Arriba | # CPUs | Load/HWM | CPU Idle |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | www.freelotto.com | 1 | 192.168.48.175:80 | 948 MB | 602 MB | 0/0 | 0 [0] | 372036 | 2 | 0.860/2 | 0.677 |
| 1 | www.freelotto.com | 1 | 192.168.48.174:80 | 948 MB | 578 MB | 0/0 | 0 [0] | 372036 | 2 | 0.450/2 | 0.000 |
| 2 | www.freelotto.com | 1 | 192.168.48.172:80 | 948 MB | 622 MB | 0/0 | 0 [0] | 372036 | 2 | 0.420/2 | 0.905 |
| 3 | www.freelotto.com | 1 | 192.168.48.171:80 | 948 MB | 594 MB | 0/0 | 0 [0] | 372036 | 2 | 0.950/2 | 0.728 |
| 4 | www.freelotto.com | 1 | 192.168.48.170:80 | 1003 MB | 532 MB | 0/0 | 0 [0] | 372036 | 2 | 1.090/2 | 0.844 |
| 5 | www.freelotto.com | 1 | 192.168.48.177:80 | 948 MB | 593 MB | 0/0 | 0 [0] | 372036 | 2 | 0.300/2 | 0.000 |

These fields are presented in a convenient and easy to read HTML format via mod_backhand's "backhand-handler" content handler. The default install activates this handler for the /backhand/ location. Visiting `http://hostname/backhand/` will display its contents. See the *Sample output of a <Location> with "SetHandler backhand-handler"* image for more detail.

Notice that the servers are listed in no particular order. The only guarantee given is that the local server is the first in the list – serverstats[0]. The servers are listed in the order they are discovered. The moderator is responsible for collecting resource information from peers within the cluster and, as it does this, it updates the information in this table. If the moderator receives resource information for a server it has not yet heard of, it will insert the information in the next available (unused) serverstats row.

Every candidacy function can see this information and use it to make decisions. In addition to this, the candidacy functions can see the `request_rec` created by Apache upon interpreting the request. This includes the method, URL, protocol, and all headers including cookies.

## 7.2 Candidacy functions

Let us define candidacy functions since we skimmed over them earlier, but didn't really describe how they work in detail. They decide which server(s) are the candidates to serve a request, in order of preference.

Given the previously described configuration of 10 web servers, we will refer to the servers by their indices in the serverstats table; 0 through 9, inclusive. Suppose that 2 and 4 have crashed and have not been heard from for several minutes. The list of servers, if viewed from the perspective of server 0 (www-0-1), could look as follows (simplified):

| Entry | Hostname | Age | Load |
|---|---|---|---|
| 0 | www-0-1 | 0 | 1.340 |
| 4 | www-0-5 | 544 | 6.100 |
| 3 | www-0-4 | 1 | 2.540 |
| 7 | www-0-8 | 1 | 0.280 |
| 6 | www-0-7 | 0 | 1.730 |
| 9 | www-0-10 | 0 | 1.280 |
| 2 | www-0-3 | 1582 | 1.000 |
| 8 | www-0-9 | 0 | 5.230 |
| 1 | www-0-2 | 1 | 4.890 |
| 5 | www-0-6 | 0 | 8.030 |

In the Apache configuration, we could specify that we want to direct all URI's that end with '.php' to the least loaded server that is currently available. To do that, a <Files> directive could be specified as follows:

```
<Files ˜ "\.php$">
  Backhand byAge
  Backhand byLoad
</Files>
```

If a request arrives that is a PHP script (ending in .php), mod_backhand will trigger and pass the list of servers through each candidacy function in order.

First, the Apache request structure and the list $[0, 4, 3, 7, 6, 9, 2, 8, 1, 5]$ is passed into the byAge candidacy function. The byAge candidacy function will eliminate all servers that have not been heard from within the last 5 seconds. The resulting list will be $[0, 3, 7, 6, 9, 8, 1, 5]$. This list and the Apache request structure will be passed into the byLoad function, which will sort the list from least loaded to most loaded. Upon the return of the byLoad function, the list will be $[7, 9, 0, 6, 3, 1, 8, 5]$. As there are no more candidacy functions specified, mod_backhand will enter the backhand-redirection handler with the intent to redirect the request to server 7 (www-0-8.)

So how is the method of reallocation chosen? Well, the list of servers that is passed into each candidacy function for augmentation is not really a list of numbers; it is actually a list of structs. Each struct contains an "id", "redirect", and a "hosttype" field. The id is the number alluded to in the above lists. The redirect is either `MB_HTTP_PROXY` or `MB_HTTP_REDIRECT`. If it is `MB_HTTP_PROXY`, then the hosttype field is ignored, and if that server is ultimately chosen, the request will be proxied there. If the redirect field is set to `MB_HTTP_REDIRECT`, then mod_backhand will attempt to issue an HTTP redirect to reallocate the request to that server if it is chosen in the end. The hosttype field specifies whether the hostname (set to `MB_HOSTTYPE_NAME`) or IP address (set to `MB_HOSTTYPE_IP`) of the server should be used when constructing the URL for HTTP redirection.

A candidacy function that sets the redirect field to `MB_HTTP_REDIRECT` and the hosttype field to `MB_HOSTTYPE_NAME` can optionally add an entry into the request's "note" table with the key "Backhand-Redirect-Host." If this is done, the value of that entry will be used instead of the hostname from the serverstats structure when constructing the URL for redirection.

## 7.3 Built-in candidacy functions

The mod_backhand distribution comes with several built-in candidacy functions:

- **off** – this disables mod_backhand for a directory.
- **addSelf** – adds the local server to the end of the list of candidates if that server does not al-

ready exist in the list. If you are using a peer-based topology, you want to consider yourself most often, and some configurations (byRandom, byLogWindow) could remove you.

- **byAge [time in seconds]** – eliminates servers that you have not received resource information for in a certain amount of time. The default is 20 seconds, but you can pass a parameter that is interpreted as an integer number of seconds.

- **byLoad [bias]** – reorders the list of candidate servers from least loaded to most loaded. The bias (a floating point number) is used to prefer yourself over proxying the request and can be used to approximate the effort involved in proxying a request. It is the amount of load that is added to all other servers' loads before sorting takes place.

- **byBusyChildren [bias]** – reorders the list of candidate servers from the server with the least to the most number of Apache children in state `SERVER_BUSY`. The bias (a floating point number) is used to prefer yourself over proxying the request and can be used to approximate the effort involved in proxying a request. It is the number of children that are added to all other servers' number of children before sorting takes place.

- **byCPU** – eliminates all servers except for those with the absolute highest CPU idle. This is, for the most part, useless. Don't use it unless you really know *why* you are using it.

- **byLogWindow** – eliminates all but the first log base 2 of the n servers passed in. So, if 17 servers are passed in, the first 4 remain.

- **byRandom** – this function randomly (pseudo, of course) reorders the list of servers given as input.

- **byCost** – this function attempts to assign a cost to the assignment of a request to each machine in the cluster. It then chooses the assignment that costs the least. The method of cost assignment is based on a cost-benefit framework as discussed in the paper titled "A Cost-Benefit Framework for Online Management of a Metacomputing System" by Amir, Awerbuch, and Borgstrom available at http://www.cnds.jhu.edu/pub/papers/dss99.ps

- **HTTPRedirectToIP** – instructs mod_backhand to reallocate clients' requests to the servers within the cluster via an HTTP redirect of the form http://1.2.3.4/request/uri rather than the default method of proxying.

- **HTTPRedirectToName [format string]** – instructs mod_backhand to send clients to servers within the cluster via an HTTP redirect of the form http://format string/request/uri rather than the default method of proxying. The format string, if omitted, will simply be the ServerName for the Apache server chosen. As this is not always a desirable choice, format string provides a means for a more intelligent hostname creation. It allows the construction of the new hostname based on the left portion of the ServerName (the Hostname on the backhand-handler page) and the right portion of the hostname provided from the *Host:* header. This facilitates clustered name based virtual hosting setups.

- **bySession [identifier]** – this function will attempt to find a cookie named [identifier] or a query string variable named [identifier]. It will then attempt to hex decode the first 8 bytes of its content into an IPv4 style IP address. It will attempt to find this IP address in the list of candidates and, if it is found, it will make the server in question the only remaining candidate. If any of the above steps fail, it will not augment the candidacy list. This, plus a bit of server-side application code, can be used to implement sticky user sessions – where a given user will always be delivered to the same server once a session has been established. [identifier] defaults to “PHPSESSID=” as it was originally written to support PHP sessions by Martin Domig. For more information see the README.bySession file that is included in the mod_backhand distribution.

## 7.4 External candidacy functions

In addition to these built-in functions, mod_backhand provides a flexible and convenient infrastructure that facilitates the creation of alternative candidacy functions. These candidacy functions are coded and executed exactly as built-in functions are, but they are loaded dynamically at run time. These functions can be added, removed, or changed without recompiling mod_backhand or Apache.

A simple candidacy function that performs no operations on the list of candidates could be written in a file `mycfs.c` as follows:

```
#include <httpd.h>
#include <mod_backhand.h>

int noop(request_rec *request,
         ServerSlot *candidates,
         int *numcandidates,
         char *optionalarg) {
  return *n;
}
```

We then compile this into a shared object. This is extremely platform dependent, but the following works on several platforms:

```
gcc -fPIC -I/usr/apache/include \
   -I/path/to/mod\_backhand \
   -c mycfs.c;
gcc --shared -o mycfs.so mycfs.o
```

To use this with mod_backhand we can add the line: `BackhandFromSO /path/to/mycfs.so noop`

If this function were to use an argument, one could be specified directly after the word `noop`, but as the noop routine will not use it, we shall not specify one.

**Source listing for byHostname.c**

```
#include "httpd.h"
#include "http_log.h"
#include "mod_backhand.h"

static char *lastarg = NULL;
static regex_t re_matching;

int byHostname(request_rec *r,
               ServerSlot *servers, int *n,
               char *arg) {
  int ret, i, mycount;
  if(!arg) return -1;
  if(!lastarg || strcmp(arg, lastarg)) {
    /* This will compile the regex only once
     * per string of consecutive expressions */
    if ((ret = regcomp(&re_matching, arg, REG_EXTENDED))!=0) {
      char line[1024];
      ret = regerror(ret, &re_matching, line, sizeof line);
      ap_log_error(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, NULL,
   "Internal error: regcomp(\"%s\") returned non-zero (%s)",
   arg, line);
      return -1;
    }
    if(lastarg) free(lastarg);
    lastarg = strdup(arg);
  }
  mycount=0;
  for(i=0;i<*n;i++)
    if(!regexec(&re_matching,
                serverstats[servers[i].id].hostname,
                0, NULL, 0))
      servers[mycount++] = servers[i];
  *n=mycount;
  return mycount;
}
```

This “noop” function is a trivial example of a candidacy function. A more complicated example is the byHostname candidacy function that is distributed with mod_backhand, which is designed to be a tutorial on how to build your own dynamically-loadable candidacy functions. byHostname will cull all servers from the candidacy list whose hostnames do not match the regular expression supplied as the argument. If it is compiled and installed in /path/to/apache/libexec (alongside all of the other modules), it can be specified as a candidacy function as: `BackhandFromSO` `libexec/byHostname.so byHostname (sun|alpha)`. This will allow only servers whose hostnames contain "sun" or "alpha" to remain as candidates.

### 7.5 Other candidacy craziness

In addition to augmenting the candidacy list that is passed in, a candidacy function can do other things. It is also passed the apache request_rec structure, and changing the values contained therein will affect the rest of the request. A candidacy function has the ability to modify the URL and the inbound headers, including cookies

If you were using the HTTPRedirect candidacy functions, the request would be reallocated using an HTTP redirect. If you specify an IP address or a hostname that is in another domain, the client's browser will not send the cookies during the subsequent request. It would be quite easy for a candidacy function to take the cookie in question and append it in some form to the query string for this request. If this is done, when mod_backhand issues the HTTP redirect, it will include the query string.

You can do anything you like in your candidacy function, but remember that it is executed *before* the response is given to the user; so if it has the potential to block, it is probably a very bad idea to perform that operation. Setting notes or headers in the request can be used during the logging phase by mod_log_config.

## 8 The moderator

The moderator process is essentially the plumbing of mod_backhand. It provides all of the necessary facilities to make load-balancing within a cluster possible, sans proxying which is provided in the child itself. It has several responsibilities, all of which are satisfied in a single event-driven process.

### 8.1 Resource acquisition and distribution

The moderator process analyzes the OS to determine the available resources on a second-by-second basis. This information is placed in the serverstats structure. This is the most non-portable portion of mod_backhand as each operating system provides entirely different APIs to interface with the internal kernel structures. To give a few examples: Solaris provides the kstat API, BSD provides access through the sysctl system call, and Linux exposes its resource information through the proc filesystem. All of these systems and future ports will require constructing the code segments that populate the serverstats structure from scratch. All of this code is separated into the platform.c source file. Many ports will require only modifications to this file.

Once the resource information is collected, it is immediately multicasted to the cluster. The address(es) to which it multicasts are specified in the Apache configuration file using the MuticastStats directive. There are four forms of this directive:

1. MulticastStats *broadcastaddr*:*port*
2. MulticastStats *myaddr*:*port* *broadcastaddr*:*port*
3. MulticastStats *IPmulticastaddr*:*port,ttl*
4. MulticastStats *myaddr*:*port* *IPmulticastaddr*:*port,ttl*

If *myaddr* is specified, then the contact field in the serverstats structure will be set to that value, otherwise mod_backhand will determine the local machine's IP address based on the machine's hostname.

## 8.2 Resource collection

The moderator will listen on a set of broadcast and/or multicast ports for resource information from other machines in the cluster(s). Upon receiving these resources, the source IP address is checked against an IP access control list and is either ignored or integrated into the local serverstats table that represents the cluster-wide resource utilization. The serverstats structure is a System V shared memory segment and is available to all Apache children and thus the candidacy functions that will be executed within them.

## 8.3 Connection pooling

The moderator is responsible for providing "live" connected sessions to Apache children on demand. Apache children, if executing in the backhand-redirection routine, may ask the moderator for an active connection to a particular server within the cluster. If an already established connection does not exist, the moderator must construct one.

Upon the creation of an Apache child process, the child will connect to the moderator process through a Unix domain socket named "bparent" that resides in the directory specified by the UnixSocketDir mod_backhand configuration directive.

The moderator accepts requests from Apache children using the mod_backhand control system (MBCS) protocol. Using this protocol, an Apache child will request a file descriptor for a socket that is connected to a particular server in the serverstats table, and the moderator will respond by passing an appropriate file descriptor back to the child.

Once the child has this file descriptor, it attempts to proxy the request over the connection. If it fails in a fashion that could be remedied with a fresh file descriptor, then the child closes the "bad" file descriptor and requests a new one. If the proxying fails in an unrecoverable fashion, the request falls through to the Apache server as if mod_backhand "chose" not to backhand the request at all.

The moderator maintains a separate pool of open TCP/IP sessions to each server for which it has been collecting resource information. After a child has successfully proxied an HTTP request over the file descriptor provided by the moderator, that file descriptor is returned to the moderator using MBCS, and the moderator replaces the connection into the pool for the server to which that file descriptor is connected.

This allows mod_backhand to utilize HTTP keep-alive semantics more extensively than the client ever could. Imagine hundreds of clients connecting to the front tier of mod_backhand enabled web servers while all of their requests are pipelined over a relatively small number of connections to the second tier. The front servers can pipeline hundreds to thousands of HTTP requests over a single TCP/IP session and thus eliminate a vast majority of the TCP/IP construction and deconstruction that would otherwise be necessary. This "acceleration" optimization hardly constitutes referring to the front-end and back-end web servers as different tiers, but it helps to clarify the picture.

### 8.4 Keep-alives – a blessing and a curse

All recent HTTP protocols provide the notion of "keep-alives," which allow a client to perform multiple HTTP transactions over a the same TCP/IP sessions (without necessitating the establishment of a new TCP/IP session for subsequent requests). TCP/IP session construction requires several round trips between the two endpoints. If there is any substantial latency between these endpoints, then construction and deconstruction of the session can be quite expensive with respect to both local resources and end-user experience.

It should be clear that using keep-alives for sessions would benefit both the end-user and reduce resource utilization across the web server(s). Unfortunately, there are other factors that can make this an unwise decision. The design of Apache 1.3.x allocates an Apache child to service each request. If keep-alives are respected, the Apache child will be allocated to that TCP/IP session for multiple subsequent HTTP requests even if there is a substantial pause between each of them.

A vast majority of the content served by web servers is small enough (less that 64 or 128 kilobytes) for it to be perfectly reasonable to increase the `SO_SNDBUF` socket option to completely hold the data destined for the client. If this is done and keep-alives are disabled entirely, each Apache child can write its response to the user and close the socket immediately without the risk of extensive blocking. Essentially, the OS can be made responsible for "spoon feeding" the client instead of the Apache child. This frees the Apache child to accept and service another client's request immediately.

A simple example will put this into perspective. Let us assume that it requires a server an average of 10ms to service a request while maintaining a load of 10. This means that in a single second, you could potentially service 100 requests with a single child, but you have an average of 10 children in use at any time, so you are servicing 1000 requests/second. These numbers are completely reasonable to achieve on commodity hardware.

This setup makes a bold assumption. We are assuming that once a request is serviced (10ms) we can immediately start servicing the next request with that child. If keep-alives are enabled, this is not so. Keep-alives allow a client to hold a connection open for some period of time so that it can make its next request over the existing established TCP/IP session.

For the sake of argument, let us assume that our keep-alive timeout (the time after which we close an idle connection to a client) is set to one second. *Note that the default Apache keep-alive timeout is 15 seconds!!!* If we were to support the same load with keep-alives enabled and each client only requests one document, then each request would take 1.010 seconds from the perspective of the Apache child. This would mean a sustained 1010 Apache children!!!

It is unfair to suppose that each client will only issue a single request. Let us suppose that a child requests 5 objects per session; that is still a minimum of 202 sustained Apache children. The problem is that the clients that reuse the same session often pause between requests due to latency and leave the session open in case the user decides to visit another URL that could utilize this session. On extremely high traffic sites, you will see that it simply will not work to enable keep-alives on the first tier web servers.

Although the end user experience is slightly degraded (usually not noticeable by humans, only by benchmarks), the fact that you can support an order of magnitude more concurrent users clearly outlines the necessity to disable keep-alives. The argument that users *always* request multiple documents over each session is often a poorly constructed one. The user may request several objects when first visiting a site, but the extremely aggressive nature of browser caching often causes that user's subsequent sessions to request a single object.

## 9 Resource allocation decisions

We have outlined the various built-in candidacy functions that can be combined together to create a powerful and intelligent decision-making algorithm. We have also outlined that one can easily create candidacy functions of their own if those provided do not meet the system requirements. However, we have not discussed what the "right" decision-making function is.

This is a very hard problem. From an academic perspective, it would be nice to say, "my algorithm for load-balancing will perform within a constant factor of optimal." The general concept may sound good, but the assumptions are not clear. The cost-benefit framework provides an algorithm that will meet this constraint. However, its implementation is very awkward and thus testing it accurately is complicated. This framework tends to shine under heavy load, but in our limited experiments, this load far exceeds any load that would ever be placed on a cluster. This is probably due to the inability to track resource availability in a manner that approaches the granularity on which we make decisions.

This points to the core of the problem: *the resource information is not particularly useful*. It may seem that an intelligent decision could be made given the load of the machines, their power relative to the rest of the cluster and their available memory. This information is updated on a one second period, but resources are allocated hundreds of times per second. Our resource allocation granularity is two orders of magnitude off our resource information collection granularity!

In addition to the poor granularity, system load is a one minute rolling average update on a five second period. If you allocate 1000 requests to a machine, it will be several seconds before you see its load vary at all! The system load information is not a good picture of the current state of a machine.

This does not mean that the information is useless. The nature of the data merely must be accounted for in the algorithms. A simple example can demonstrate a "poor" algorithm that does not account for the nature of system load.

Choosing the least loaded server may sound like an intelligent decision, but consider that everyone in the cluster sees the same thing and, as mentioned, the system's load will take a considerable amount of time to reflect the immediate stress on a system. In a cluster of 10 machines with each receiving 100 requests per second, 9 of those servers will be directing **all** of their requests to the same machine for the next several seconds. This is starting to sound like a very bad idea. Considering that requests take relatively little time to respond to (relative to a few seconds), you have 9 servers that are performing no actions but proxying and 1 server that is being pulverized.

The key points to remember when analyzing mod_backhand load-balancing schema are:

1. The resources you are seeing right now could be stale (like load).
2. The resources you see do not account for all of

the requests that have been reallocated across the cluster since the last update.

3. All servers see the same data, so if a server is chosen based solely on the resources, there will be contention issues.

4. The resource information does not account for the overhead that the local server must incur in order to proxy the connection.

Handling issue 1 is a serious problem. However, actually attempting to compensate for the stale information (by weighting it less as it gets older or several other techniques) would be far too intensive computationally. We would spend all of our time calculating how important the information is and no time serving web pages – let's not lose track of the goal! Currently, the most obvious offender of the stale information problem is load. The reason for this is that we are not really interested in the system load over the last minute. Rather, we would like to know the average system load over the last second. This information is not readily available on most UNIX systems, so we must use what *is* available.

Since load is defined as the average length of the run queue, it may make sense to use the length of the run queue at a given point in time in place of system load. This too is not a standard metric and thus is not readily available. However, assuming that the content spread across you cluster is relatively homogeneous (meaning you attempt to service all content everywhere), the number of Apache requests currently on the run queue should be fairly representative of the system run queue. We can simply find the server with the least number of "busy" Apache children. This is available in the byBusyChildren candidacy function.

To handle contention (issues 2 and 3), there are several approaches. One easy to implement solution is the approach that each server will limit its choices to a random subset of the available servers. This will ensure that not all servers will select the same server during a given update interval. If a random window of size $n$ of the candidates is chosen and then the least loaded within that window is selected, there will be some distribution of selections *concentrated* on the least loaded server and completely excluding the $n-1$ most loaded servers.

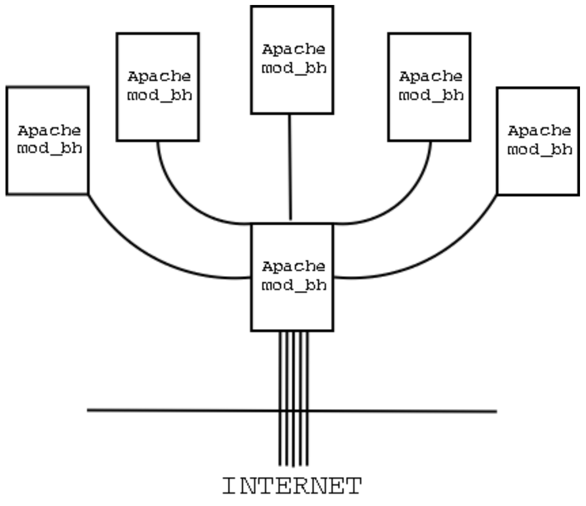
Issue 4 is precisely the reason that the byLoad function and the byBusyChildren function have a "bias" argument. It will allow a server to prefer serving the request to proxying to a server with the same or slightly more available resources.

## 10 Example configurations

### 10.1 Single-point cluster

This model best describes a cluster of machines in which a single is advertised to the public. That machine accepts requests and either services them locally or proxies them to a "peer" machine within the cluster.

**Single-point cluster**



This is a very simple configuration that allows clean and fairly efficient control over the cluster-wide resources. Contention issues are avoided entirely as the only server that reallocates requests is the single point of entry. Of course, it presents a single point of failure and a single bottleneck. The entire cluster can only push traffic as fast as the single front-end machine can proxy the requests and their responses. mod_backhand does operate above layer 4, so it incurs dramatically more overhead than a layer 3 proxy. A layer 3 proxy can't proxy different requests on the same TCP/IP session to different servers.

This configuration lends itself to heavy dynamic content. If the content is static and requires few resources to serve, it does not make sense to try to funnel through an application level proxy. If the content requires substantial resources, assigning resources intelligently can help stabilize the cluster and speed web serving overall.

mod_backhand makes it very easy to balances different content using different rules. Assume we use mod_perl+Apache::ASP and mod_php and we wish to balance that content based on system load. All other content should be served by the local machine. The following configuration will balance all ".php" and ".asp" to one of the back-end servers based on load using a randomized log-window to give a decent distribution.

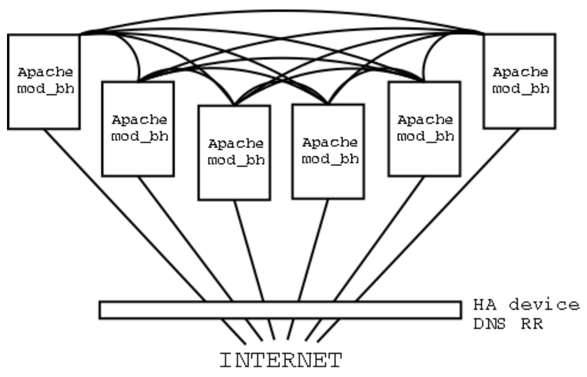
```
<Files ~ "(\.asp|\.php)$">
  Backhand byAge
  Backhand removeSelf
  Backhand byRandom
  Backhand byLogWindow
  Backhand byLoad
</Files>
```

That's all! All requests to the front-end machine that match the above <Files> statement will be directed to a back-end machine with a low load (probabilistically). All other requests will be serviced locally as there are no mod_backhand-related configuration directives.

## 10.2 Multi-point cluster

Much like the previous example, the multi-point cluster's configuration simply exposes more machines (in this case all) to origination client connections. There is the clear advantage that now all machines are capable of servicing clients directly. Now it is only mod_backhand's responsibility to "fix" poor allocations.

**Multi-point cluster**



It is hard to justify reallocating the serving of static images and static HTML content as it is expensive to proxy them in a user space process like mod_backhand.

In addition, we should take this overhead into account when choosing to reallocate requests. Keep in mind we should *always* consider the local server, so if it is eliminated at some point probabilistically, it should be added back in. A more appropriate rule set than the one presented in the single-point scenario would be:

```
<Files ~ "(\.asp|\.php)$">
  Backhand byAge
  Backhand byRandom
  Backhand byLogWindow
  Backhand addSelf
  Backhand byLoad 2.0
</Files>
```

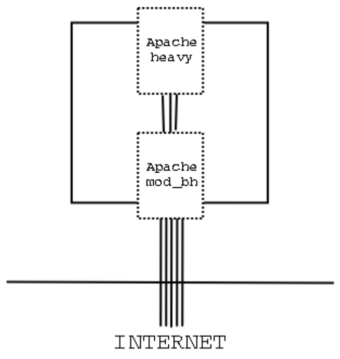
The reason we always want to consider the local instance and give the local instance a preference over its peers is because the intent is to serve content, not proxy connections. The proxying done should be done only to correct poor allocations. If it is done without enough potential gain, we will find that the cluster is spending much of its resources proxying and little actually performing its job.

In this configuration mod_backhand must choose itself over all others. The local server is guaranteed to be the first in the list of candidates; so enabling it just requires writing a rule that will not rearrange the list and not eliminate the local server. The byAge rule accomplishes this. Simply adding `Backhand byAge` as the only mod_backhand rule for the content that needs to be proxied to the heavy instance will achieve the desired effect.

## 10.3 Simple HTTP accelerator

A common use of Apache's mod_proxy module is to provide HTTP acceleration for a bulkier Apache instance on the same machine. An Apache instance with a large mod_perl or mod_php application can be heavy. By this we mean that it can have a memory footprint that is substantially larger than that of a "lean and mean" Apache instance.

**HTTP accelerator**



It is nonsensical to serve static pages and images with an instance that is so unwieldy. It should be used only to serve the content for which it is required. The classic method of accomplishing this is to have an Apache instance with only mod_proxy and a few other light-weight modules listening publicly. The Apache instance that is "heavy" will be listening only on an internal or private interface. The public instance will serve all requests except those that require the heavy instance. The remaining requests are reverse proxied to the heavy instance. This means that a heavy Apache child isn't needed to serve each request and if the documents served by the heavy instance are large, the proxy will buffer the document so the heavy child will be freed quickly to serve the next request.

mod_backhand can be used in a similar fashion with a particular advantage – it uses connection pooling so that no time needs to be wasted connecting between the instances. The `BackhandSelfRedirect` directive will tell mod_backhand that it if it chooses itself, it is to proxy the request instead of allowing the request to fall through to the underlying Apache instance.

Now the trick is to convince the front-end instance that it is actually the back-end instance. That way, when it attempts to proxy to "itself," it will actually be proxying to the back-end instance. The `MulticastStats` directive in form 2 or 4 (discussed on page 20) can effectively do this. If the back-end instance is bound to the loopback interface, then we will want to put 127.0.0.1 as the second first argument to the MulticastStats directive before the multicast address.

This means that the front-end mod_backhand-enabled instance will maintain a pool of open connections to back-end, heavy-weight Apache children and reverse proxy the necessary requests.

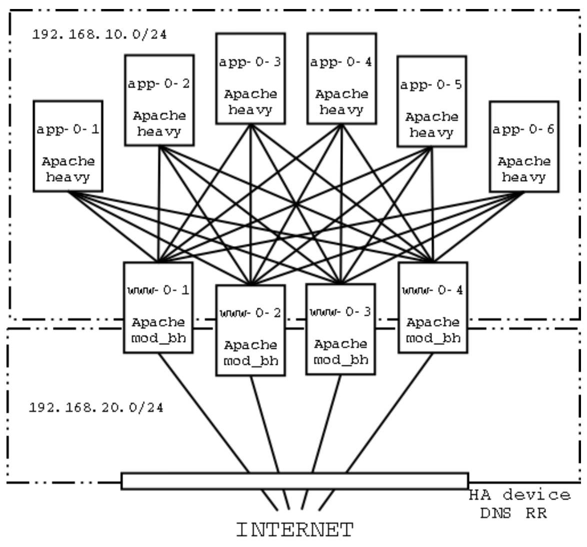
## 10.4 Two-tier, multi-point cluster

It is difficult to present a name to describe this architecture that is both descriptive and short. There are two tiers of web servers, each with a distinctly different purpose. The front tier is designed to provide all web content to the client (as the client can only connect to the first tier). These front-end machines do not have the capacity to generate the more complicated dynamic content. If the architecture was collapsed into a single tier, the heavy application servers would be serving static content – which is an inefficient use of resources.

Instead, we have a second tier of web servers whose sole purpose is to generate dynamic content for the front end to incorporate into the presented site.

This approach combines the models in sections 10.1 and 10.2, but adds the concept behind separate instances for acceleration presented in section 10.3 to create a flexible and efficient two-tier web cluster architecture.

**HTTP accelerator**



Let us look briefly at the disadvantage of an architecture like this before we describe why it is powerful and flexible. The obvious downside is that it is difficult to determine how many machines are required in each tier. In other words, as all machines are assigned to a tier, *a priori*, it is impossible to ensure that resources are optimally utilized.

But unless you know something about the future that the rest of us don't, it is impossible to allocate resources optimally anyway. We can only attempt to do a good job. So the only apparent disadvantage is deciding how many machines to place in each tier. Fortunately, in a well architected set up, the machines will be more or less interchangable. With relatively little effort, a machine can be shifted from one tier to another.

Notwithstanding this disadvantage, the architecture is quite powerful. The front tier of machines is effectively an accelerator for the back-end machines. All static content is completely offloaded to them and the more resource intensive dynamic content is serviced by the second tier.

The front tier will proxy the requests to the back end based on resource utilization. It will also ensure that the second tier can deliver content quickly and efficiently due to the low-collision, high-speed network that connects the two tiers.

All machines must run mod_backhand so as to announce themselves as candidates and to have the plumbing necessary to proxy. The rules on the second tier machines are irrelevant (and should be omitted) because they will never directly serve any client-originated requests. The front-end machines will only proxy requests that require running on the second tier and thus should never consider any machine on the first tier. Instead, they balance across the second tier servers attempting to minimize the active run queue on each machine. On the front tier machines, we want to have a configuration as follows.

**Front-tier Apache configuration**

```
UnixSocketDir /opt/apache/backhand
MulticastStats 192.168.10.255:4445
AcceptStats 192.168.10.0/24
<Location /backhand/>
  SetHandler backhand-handler
</Location>
<Files ~ "(\.php)$">
  Backhand byAge
  BackhandFromSO libexec/byHostname.so byHostname app
  Backhand byRandom
  Backhand byLogWindow
  Backhand byBusyChildren
</Files>
```

**Second-tier Apache configuration**

```
UnixSocketDir /opt/apache/backhand
MulticastStats 192.168.10.255:4445
AcceptStats 192.168.10.0/24
<Location /backhand/>
  SetHandler backhand-handler
</Location>
```

## 11 A consolidated example

In this section, we will present everything discussed thus far in a concise, real-world example.

The site is an auction site. The application is implemented in Java using Jakarta Tomcat as a backing technology. Apache serves the actually requests and is connected to Tomcat via mod_jk. The application is written in such a fashion that session state is store by the application in local memory – this poor design requires subsequent requests from the same user to be serviced by the same machine.

The site's traffic demands that at least 5 machines be used to provide adequate quality of service. Much of the site traffic is presented over HTTPS.

### 11.1 The back-end

The application layer is implementated in Java through Jakarta Tomcat through Apache. While the architecture is responsible for providing an SSL enabled connection, we will refrain from implementing this on the back-end requiring the front-end to completely handle SSL traffic serving.

Back-end machines runs Apache with mod_jk, mod_rewrite, and mod_backhand. mod_backhand is required here to advertise the machine's services to the cluster. These machines do not handle any client-originating requests.

In practice, it is good practice to not exceed 70%

capacity on any device or cluster. As the application requires 5 machines to preform its task without detriment to the required quality of service to its users, we will use 8 machines on the back-end. With 8 machines, the cluster will be at approximately 62.5% capacity.

As specified by the application designers, the back-end machines are to be dual-processor machines with at least 1GB of memory running Linux 2.4.18 and IBM's JDK 1.3.1.

**Back-end Apache configuration**

```
<IfModule mod_backhand.c>
UnixSocketDir /opt/apache/backhand
MulticastStats 192.168.10.255:4445
AcceptStats 192.168.10.0/24
<Location /backhand/>
  SetHandler backhand-handler
</Location>
</IfModule>
```

### 11.2 The front-end

The job of the front-end is the make the back-end operate as efficiently as possible and to present a highly-available front. The front-end machines would likely run on single processor machines with very little memory if the demand for SSL service wasn't placed on them. However, since these machines are responsible for providing all SSL negotiation, encyption and decryption, bolstering their raw CPU power is necessary. For front end machines, we deploy dual-AMD machines with 512MB of RAM running Linux 2.4.18. The server run relatively light instances of Apache only sporting mod_ssl, mod_rewrite and mod_backhand. Following the 70% capacity rule, we only require 2 machines for adequate QoS, so with 3 machines we are no more than 66% capacity. Hence, 3 front-tier machines are deployed.

The front-end machines do require a special session-aware load balancing that works whether the connection is an HTTP or HTTPS connection. The Java application sets a cookie named ”host” to the not fully qualified domain name of the machine it is on, in this case, is one of the following: `www-s1-n`, where `n` is between 1 and 8. So, a mod_backhand candidacy function needs to look for this cookie and account for it in its decision making phase.

The front end machines are only required to send back requests to the back-end unless we can't feasible service them. We are unable to service Java Server Pages and Servlet actions. These URLs end in `.jsp` and `.do`, respectively. All other requests are extremely lightwieght static content like images and static HTML pages. These lighter pages can be served easily from the front-end machines.

**Source listing for byAuctionSession.c**

```
#include "httpd.h"
#include "http_log.h"
#include "mod_backhand.h"

static char *id = "host=";
int byAuctionSession(request_rec *r, ServerSlot *servers,
                     int *n, char *arg) {
  const char *cookie = NULL;
  char *host = NULL;
  int len = 0;
  int i = 0;
  if(cookie = ap_table_get(r->headers_in, "Cookie")) {
    if(host = strstr(cookie, id))
      host += strlen(id);
  }
  if(!host) return *n;
  while(host[len] && host[len] != ’;’ && host[len] != ’ ’) {
    len++;
  }
  for(i=0;i<*n;i++) {
    if(!strncmp(serverstats[servers[i].id].hostname, host, len)) {
      /* This is the right server */
      servers[0] = servers[i];
      *n = 1;
      return *n;
    }
  }
  return *n;
}
```

**Front-end Apache configuration**

```
<IfModule mod_backhand.c>
UnixSocketDir /opt/apache/backhand
MulticastStats 192.168.10.255:4445
AcceptStats 192.168.10.0/24
BackhandConnectionPools Off
BackhandModeratorPIDFile \
    /var/run/backhand.pid
<Location /backhand/>
  SetHandler backhand-handler
  Backhand off
</Location>
</IfModule>

...

<IfModule mod_backhand.c>
<Files ˜ "(\.jsp|\.do)$">
  Backhand byAge
  BackhandFromSO \
      libexec/byAuctionSession.so \
      byAuctionSession
  Backhand FromSO \
      libexec/byHostname.so \
      byHostname www
  Backhand byRandom
  Backhand byLogWindow
  Backhand byBusyChildren
</Files>
</IfModule>
```

With this configuration the front tier of machines will perform the following actions on all requests ending in `.jsp` and `.do`:

1. If the user has an active session at this site, their `host` cookie is set and they will be directed to a machine that is alive and has that name.
2. If the cookie doesn't exist or the cookie does not match a live machine, then the possible machines are limited to those that are back-end web servers.
3. Then $\log_2$ of the machines will be randomly selected (3 if all 8 are available).
4. The server with the least occupied Apache children will be selected.

At this point, we have a load-balanced and highly available second-tier. However, the front-tier is exposed to the public over three IP addresses via DNS RR and we must still ensure that in the event of a front-tier system failure, the overall function and availability will not be compromised.

### 11.3 Wackamole on the first-tier

As demonstrated, Wackamole can solve the availability problem easily. Our front three machines have the following internal IP addresses 192.168.48.71, 192.168.48.72, and 192.168.48.73. We would like their external IP addresses to be 192.168.100.71, 192.168.100.72 and 192.168.100.73, but we need all external IP addresses available in the event of a system failure.

This particular configuration is a bit tricky as there are no existing IP addresses in the external IP space. So, the first IP address on each interface must have a full netmask and the subsequent should be instantiated with a /32 netmask. This requires a unique configuration on each machine with preferences.

Note that on each machine, we must have the ”preferred” IP address with the subnet's netmask and we must specifically prefer that IP address. With this configuration, each machine will know that it is immediately responsible for that primary IP address and that address will be added first. This approach eliminates a variety of technical problems with routing and interface removal.

#### 11.3.1 wackamole.conf on 192.168.48.71

```
Spread 4803
Group = wack1
Control = /var/run/wack.it
arp-cache = 90s
mature = 5s
balance {
  Interval = 4s
  AcquisitionsPerRound = 1
}

Prefer eth1:192.168.100.71/24
VirtualInterfaces {
  { eth1:192.168.100.71/24
    eth1:192.168.100.72/32
    eth1:192.168.100.73/32
  }
}
Notify {
  eth1:192.168.100.1/32
  arp-cache
}
```

### 11.3.2 wackamole.conf on 192.168.48.72

```
Spread 4803
Group = wack1
Control = /var/run/wack.it
arp-cache = 90s
mature = 5s
balance {
  Interval = 4s
  AcquisitionsPerRound = 1
}

Prefer eth1:192.168.100.72/24
VirtualInterfaces {
  { eth1:192.168.100.71/32
    eth1:192.168.100.72/24
    eth1:192.168.100.73/32
  }
}
Notify {
  eth1:192.168.100.1/32
  arp-cache
}
```

### 11.3.3 wackamole.conf on 192.168.48.73

```
Spread 4803
Group = wack1
Control = /var/run/wack.it
arp-cache = 90s
mature = 5s
balance {
  Interval = 4s
  AcquisitionsPerRound = 1
}

Prefer eth1:192.168.100.73/24
VirtualInterfaces {
  { eth1:192.168.100.71/32
    eth1:192.168.100.72/32
    eth1:192.168.100.73/24
  }
}
Notify {
  eth1:192.168.100.1/32
  arp-cache
}
```

## 12 Conclusion

Load balancing and high availability are demanded in most enteprise architectures. Sometimes, however, this technical demand is often issued by a non-technical mandate. With these mandates often come technical proposals. This is an effect of marketing stategies that target non-technical upper-level management. This should be cause for thorough investigation and evaulation of both need and appropriateness of the solution. Rarely will a mandate arrive that dictates a free solution be used to implement a business requirement such as availability and scalability – simply because there is no one to market them to upper-level managment.

As Apache itself demonstrates, open-source solutions are just as capable as commercial, close-source solutions. The Backhand Project is an initiative to provide capable and usable products that address the obstacles commonly encountered in clustered Internet architectures.

Wackamole is a breakthrough in open source software. It is the first open source, N-way, peer-based IP failover tool available. These qualities make Wackamole a fierce competitor in the HA industry. With exposure and community support, it will quickly gain many of the features found in other HA tools as they are needed by its users.

Load balancing is one of the most decpetive topics in Internet computing. There is simply no silver bullet. Each product (commerical and non-commercial) has its strong-points and clever approaches. Choosing the correct solution requires careful analysis of the existing infrastructure and the goals of the implementation. mod_backhand is unique in this field in that it is part of Apache. It requires no major augmentation of network topology and can be used in combination with other off-the-shelf HA/LB products. As it is part of Apache, it is capable of making decisions that are otherwise simply impossible using today's hardware load balancers.

The combination of Apache, mod_backhand and Wackamole provide the necessary software tools to build an infrastructure that is truly mission-critical.

## 13 Credits

Spread, mod_backhand, and wackamole are largely due to the creative design and forward thinking of The Johns Hopkins University's Center for Networking and Distributed Systems (http://www.cnds.jhu.edu/). OmniTI Computer Consulting, Inc. (http://www.omniti.com/) funds further development of all existing elements of the Backhand Project. OmniTI sponsored the authorship of this paper and accompanying presentation.

### 13.1 The Spread Group System

- Creators:
  - Yair Amir `<yairamir@cs.jhu.edu>`
  - Michal Miskin-Amir `<michal@spread.org>`
  - Jonathan Stanton `<jonathan@cs.jhu.edu>`
- Contributors:
  - Dan Schoenblum `<dansch@cnds.jhu.edu>`
  - John Schultz `<jschultz@cnds.jhu.edu>`
  - Theo Schlossnagle `<jesus@omniti.com>`
  - Ben Laurie `<ben@algroup.co.uk>`
- Thanks:
  - Ken Birman, Banny Dolev, Mike Goodrich, David Shaw, Robbert VanRenesse.

### 13.2 Wackamole

- Creators:
  - Yair Amir `<yairamir@cs.jhu.edu>`
  - Ryan Caudy `<wyvern@cnds.jhu.edu>`
  - Aashima Munjal `<munjal@jhu.edu>`
  - Theo Schlossnagle `<jesus@omniti.com>`

## 13.3 mod_backhand

- Creators:
  - Yair Amir <`yairamir@cs.jhu.edu`>
  - Theo Schlossnagle <`jesus@omniti.com`>
- Contributors:
  - Baruch Awerbauch <`baruch@cs.jhu.edu`>
  - Ryan Borgstrom <`rsean@cnds.jhu.edu`>
  - Alec Peterson <`ahp@hilander.com`>
  - George Schlossnagle <`george@omniti.com`>
  - Jonathan Stanton <`jonathan@cs.jhu.edu`>
  - Martin Domig <`md@mail.ims.at`>
  - Rob Butler <`rob_butler@hotmail.com`>

## 13.4 mod_log_spread

- Creators:
  - George Schlossnagle <`george@omniti.com`>
- Contributors:
  - Theo Schlossnagle <`jesus@omniti.com`>
  - Jonathan Stanton <`jonathan@cs.jhu.edu`>
- Principals Entities:
  - The JHU Center for Networking and Distributed Systems
  - Community Connect, Inc.
  - OmniTI, Inc.