

# backhand

use your resources

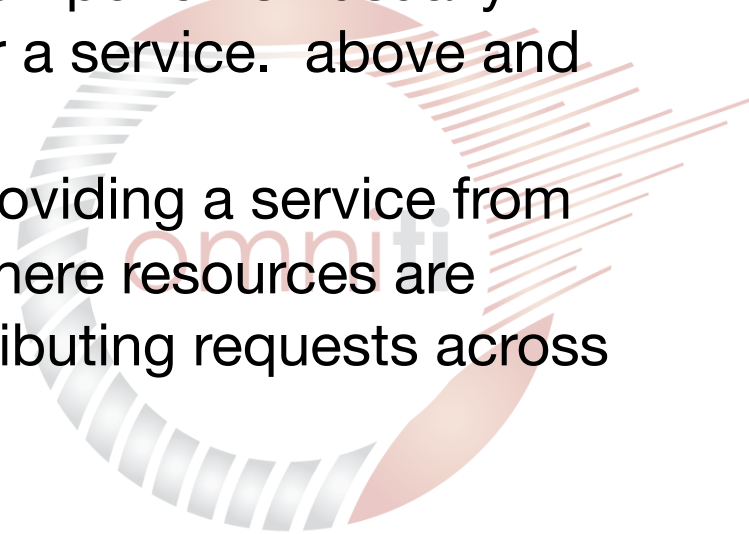
## Understanding and Building HA/LB Clusters

mod\_backhand, wackamole  
and other such freedoms



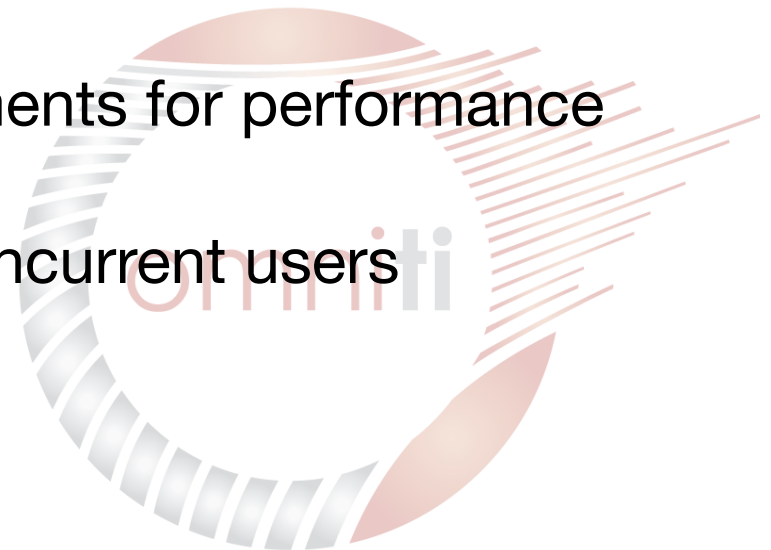
Theo Schlossnagle <jesus@omniti.com>  
Principal Consultant  
OmniTI Computer Consulting, Inc.

# Defining “Cluster,” “HA,” and “LB”

- 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 and/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 separate systems where resources are effectively combined by distributing requests across these systems.
- 

# Building Clusters... Why Bother?

1. Online applications used for internal productivity  
(Business Continuity)
2. Online applications used for revenue generation  
(Financial Continuity)
3. Strict service-level agreements for availability  
(Availability)
4. Strict service-level agreements for performance  
(Performance)
5. Services used by more concurrent users  
(Scalability)



# Clustering problems

- Fault tolerance and high availability  
`wackamole`
- Effective resource allocation  
`mod_backhand`
- Content replication  
`custom solution (NFS, rsync, etc.)`
- Server proximity optimizations  
`DNS shared IP or smarter DNS`
- Manageability  
`mod_backhand`
- Resource monitoring  
`mod_backhand`
- Seamless scalability  
`mod_backhand, mod_log_spread`
- Unified log collection  
`mod_log_spread`



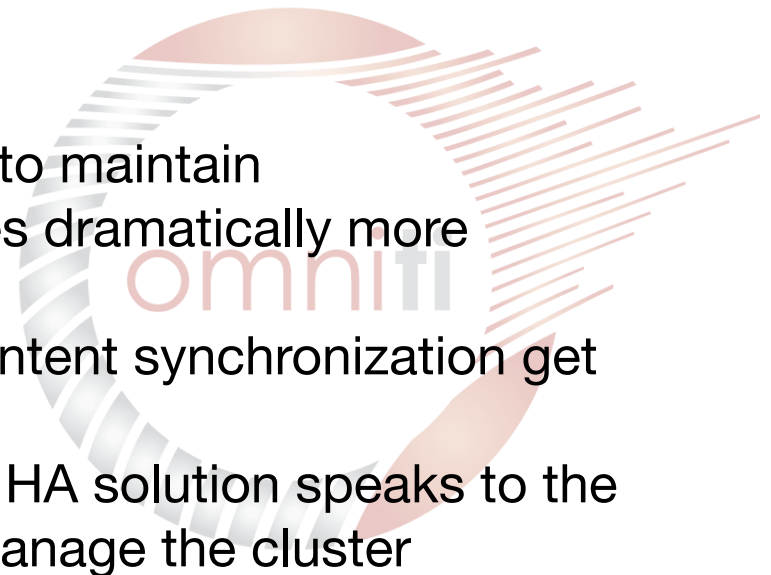
# High Availability

## Pros:

- ☺ Taking down components of your architecture for maintenance or upgrades is simple
- ☺ Architectures often lend themselves to higher capacity
- ☺ After tackling the 3 node cluster, scaling to n nodes is more a “process” than an “experiment”

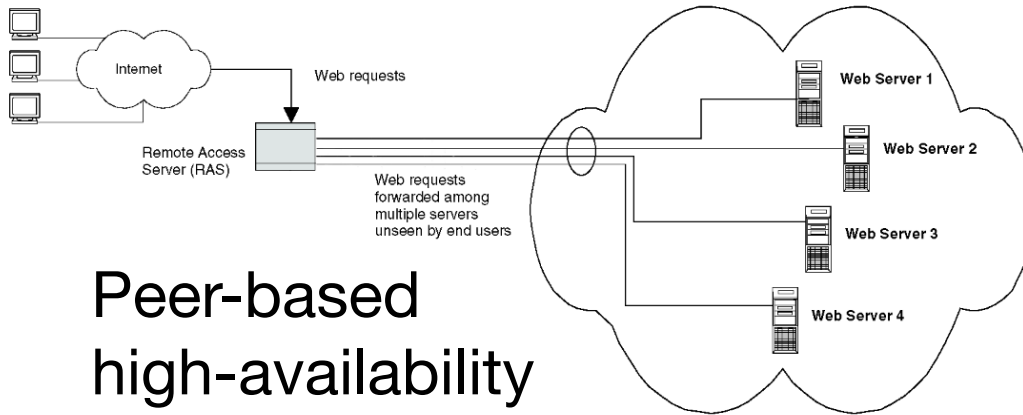
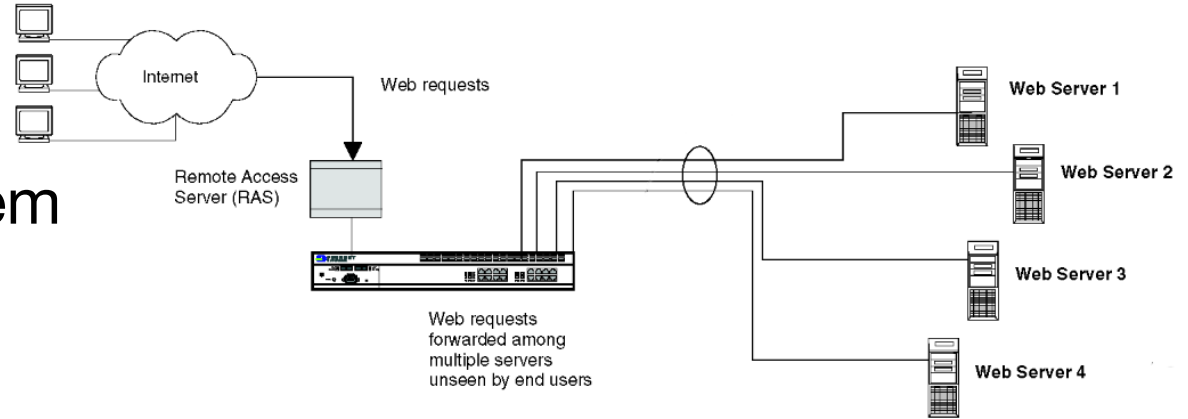
## Cons:

- ☹ More equipment, services, etc. to maintain
- ☹ Troubleshooting issues becomes dramatically more difficult
- ☹ Making your application and content synchronization get along
- ☹ The mere fact that you need an HA solution speaks to the attention required to properly manage the cluster



# High Availability — Two Approaches

Front-end high-availability distribution system

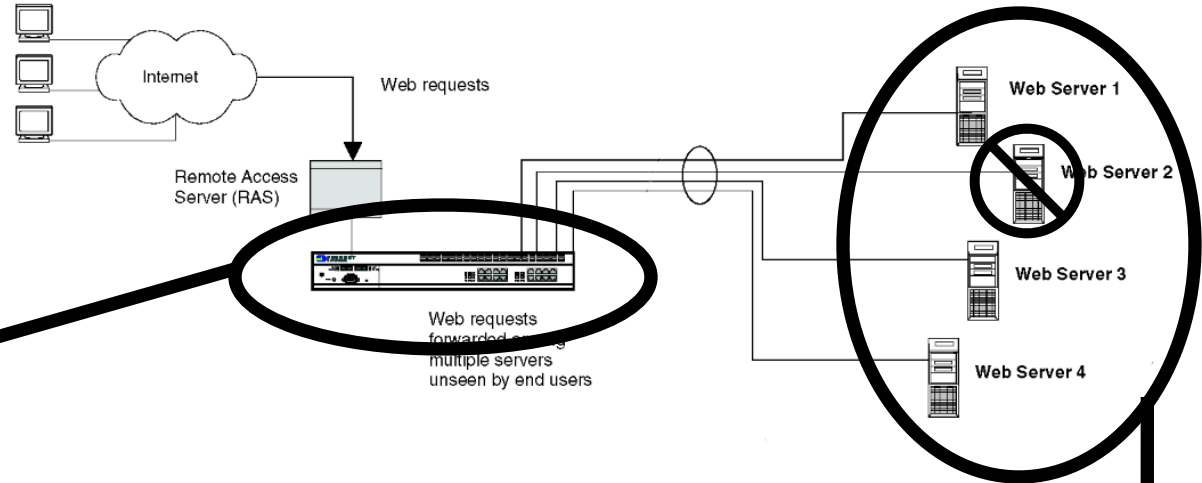


Peer-based high-availability system



# High Availability — First Approach

Front-end  
high-availability  
distribution system

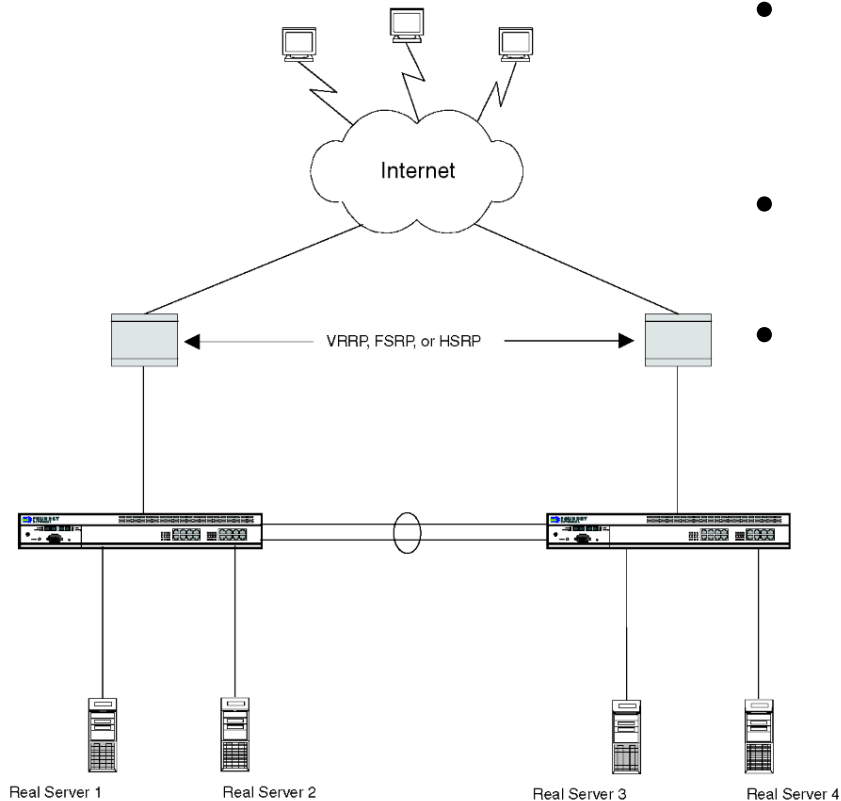


- HA device is a single point of failure
- HA device costs money and isn't **doing** anything
- Application servers are not a single point of failure

# High Availability — First Approach

(more robust)

- Single point of failure can be eliminated
- Two or more devices must be used and they must support some mechanism of failover
- All commercial solutions provide such built-in failover mechanics
- Some open source solutions for this machine-to-machine failover exist



Thought:

You have to use an HA solution to make your HA solution HA!

# Implementations



## Pair-based systems (master/slave):

Veritas Cluster Server

Linux-HA

Pirahna



## Front-end distribution systems:

LVS

F5 BIG/ip

Cisco LocalDirector

Cisco Arrowpoint

Foundry ServerIron

Alteon ACESwitch

Coyote Point Equalizer



# High Availability — A New Paradigm

## Goals:

- No specialized or dedicated hardware
- Every party involved is “responsible” for the availability of the service
- As long as one machine in the cluster is functioning it is able to survive the service

## How?

**Wackamole**  
<http://www.backhand.org/wackamole/>



# Wackamole: N-way IP failover

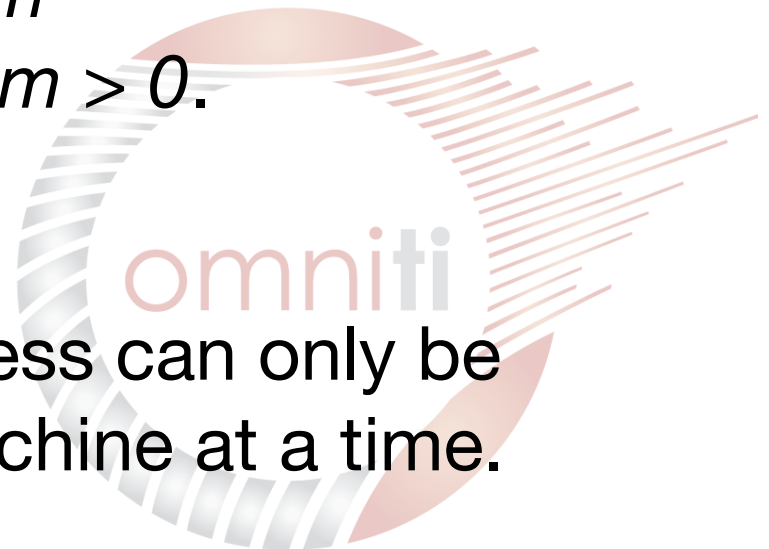
Allows  $n$  machines to provide service from  $m$  IP address with  $n-1$  faults.

Good things:

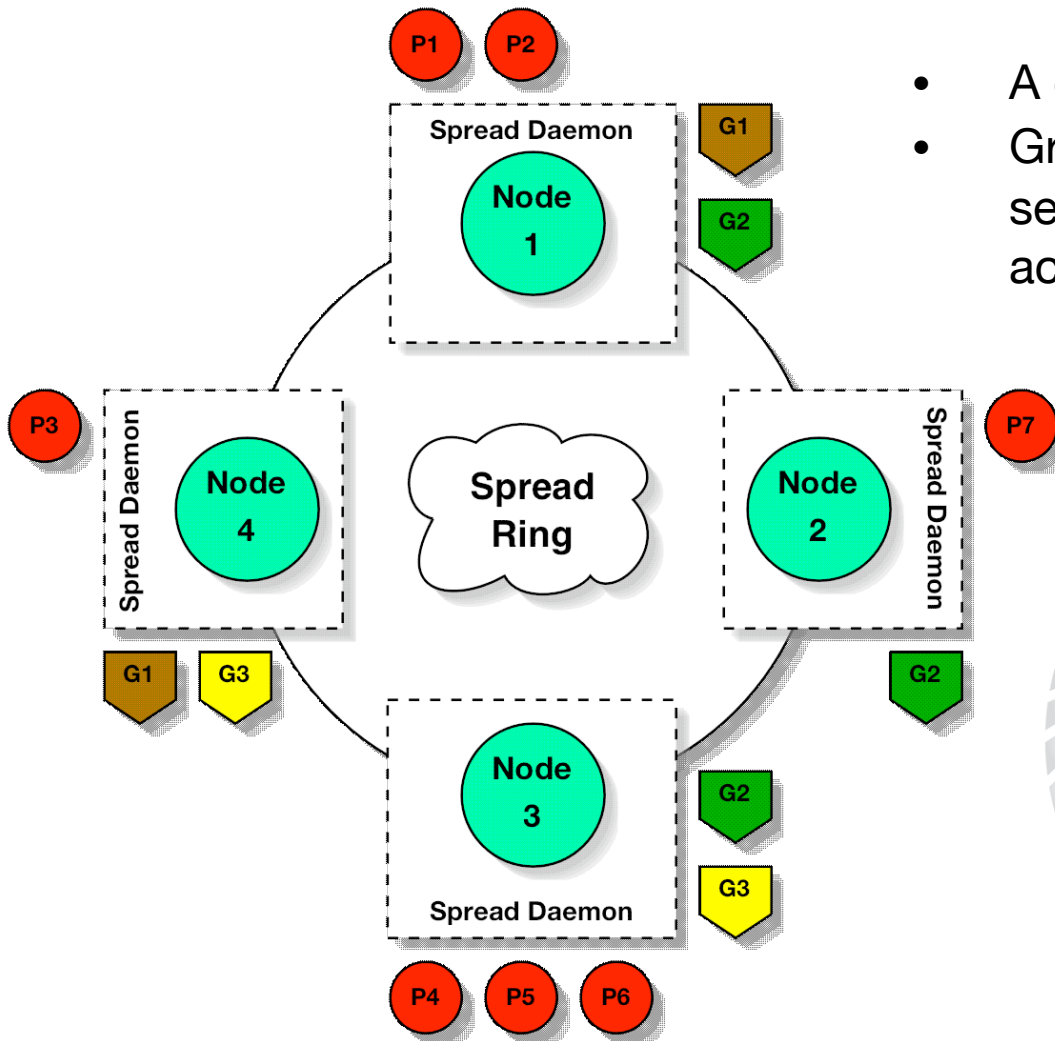
$n > m$ ,  $n = m$ , or  $n < m$   
as long as  $n > 0$  and  $m > 0$ .

Limitations:

A publicized IP address can only be managed by one machine at a time.



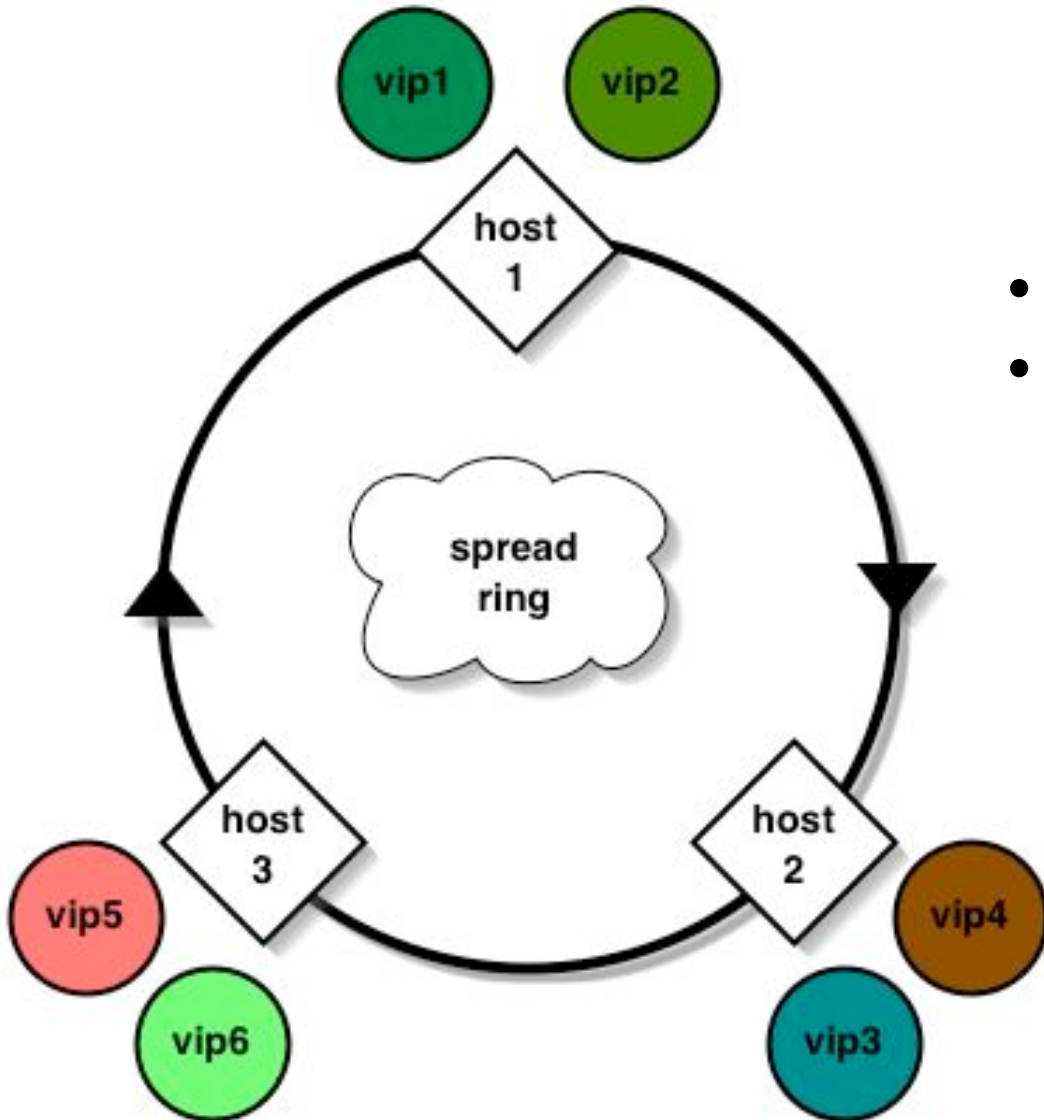
# Spread: The Concept (.1)



- A daemon on each machine.
- Groups with guaranteed delivery semantics and membership views across the cluster.



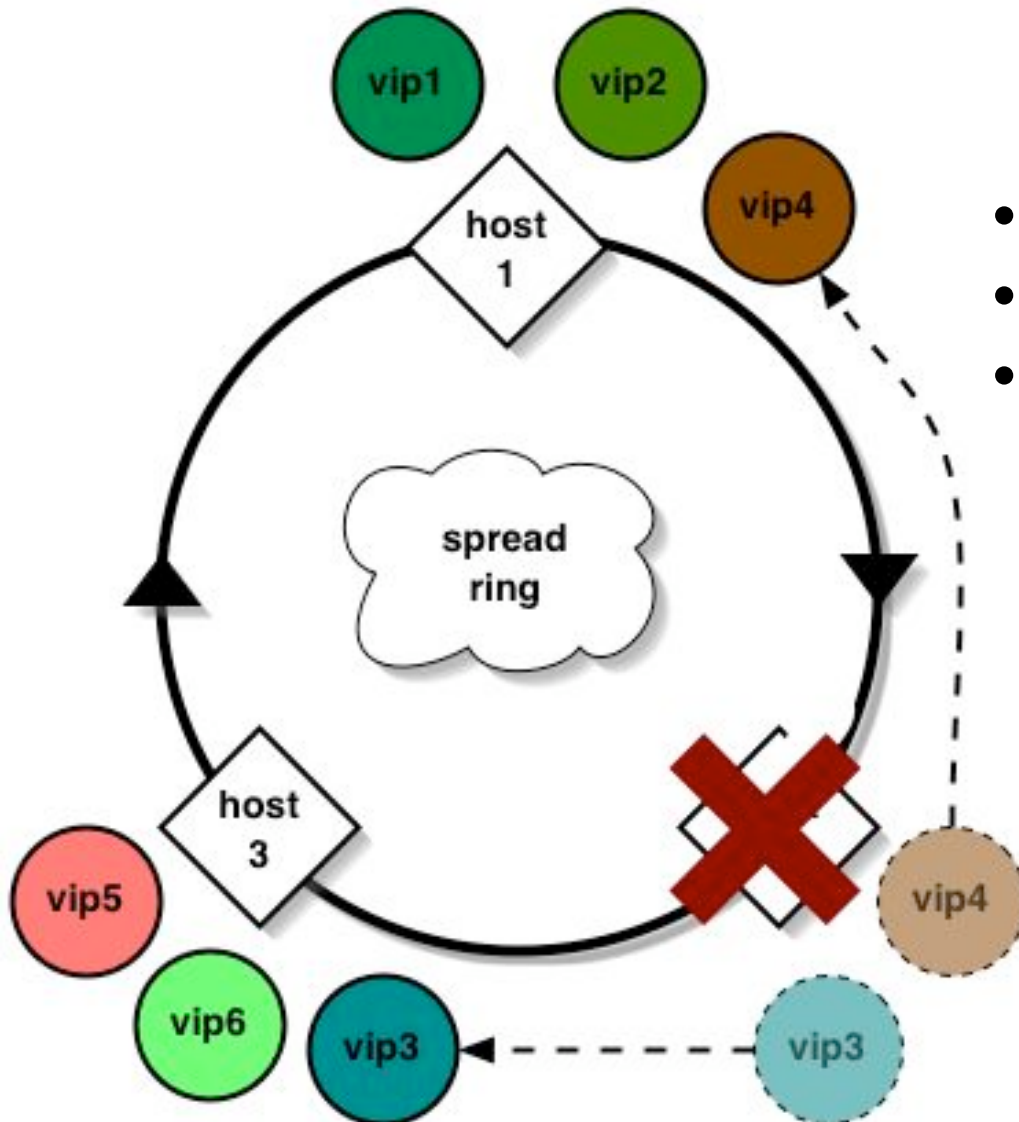
# Wackamole: The Concept (.1)



- 3 machines hosting
- 6 virtual IP addresses



# Wackamole: The Concept (.2)



- 1 machine failed
- 2 machines hosting
- 6 virtual IP addresses



# Wackamole: Installation and Configuration

No steps can be taken to install Wackamole until we first satisfy its requirements:

A supported operating system:

- Solaris
- FreeBSD
- Linux
- Mac OS X
- It's Open Source, PORT IT!

Underlying system requirements:

- Spread ([www.spread.org](http://www.spread.org))



# STEP 1:

## Installing Spread

The Spread Group Communication System is an integral toolkit used by Wackamole to make decisions:

1. Download Spread:

<http://www.spread.org/>

<http://mirrors.omniti.com/spread/>

2. Configure/Compile/Install:

```
gzip -dc spread-src-3.17.0.tar.gz | tar xvf -
```

```
cd spread-3.17.0
```

```
./configure
```

```
make
```

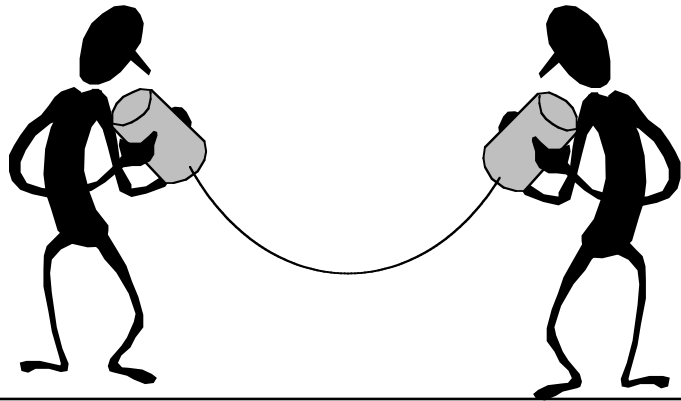
```
make install
```

(due to a bug in Spread-3.17.0, you must manually install the header files)

```
cp sp.h sp_*.h /usr/local/include/
```



# STEP 2: Configuring Spread



spread.conf:

Spread configuration for 2  
machines using IP Broadcast

```
DebugFlags = { EXIT CONFIGURATION }  
EventLogFile = /var/log/spread/mainlog  
EventTimeStamp  
DangerousMonitor = false  
  
Spread_Segment 10.77.53.255:3777 {  
    moria-1      10.77.52.2  
    moria-2      10.77.52.3  
}
```



# STEP 2 continued: Configuring Spread Bigger

spread.conf:

Spread configuration for 10  
machines using IP Multicast



```
DebugFlags = { EXIT CONFIGURATION }
EventLogFile = /var/log/spread/mainlog
EventTimeStamp
DangerousMonitor = false
Spread_Segment 225.77.52.128:4803 {
    samwise                10.77.52.62
    peregrine              10.77.52.18
    meriadoc               10.77.52.19
    www-s1-1               10.77.52.65
    www-s1-2               10.77.52.66
    www-s1-3               10.77.52.69
    stage-s1-1             10.77.52.68
    rsync                  10.77.52.67
    sslproxy-s1-1          10.77.52.70
    sslproxy-s1-1          10.77.52.71
}
```

# STEP 3:

## Installing Wackamole

1. Download Wackamole:

<http://www.backhand.org/wackamole/>

<http://mirrors.omniti.com/wackamole/>

2. Configure/Compile/Install:

```
gzip -dc wackamole-2.0.0.tar.gz | tar xvf -  
cd spread-wackamole
```

```
./configure
```

```
make
```

```
make install
```



# Wackamole Configuration Basics (.1)

## **Spread = <string>**

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.

## **Group = <string>**

Specifies the Spread group that should be joined. This group should only be used by Wackamole instances in the same cluster.

## **SpreadRetryInterval = <time>**

Specifies the amount of time between attempted but failed reconnects.



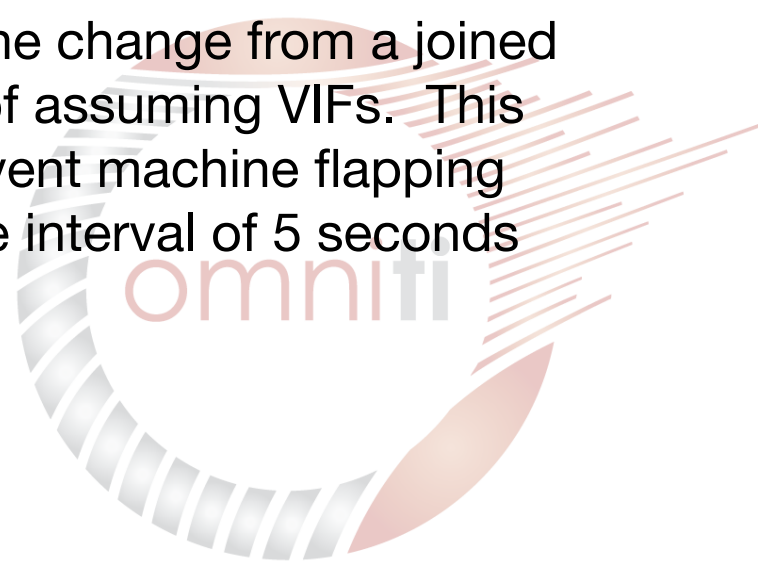
# Wackamole Configuration Basics (.2)

## **Control = <filename>**

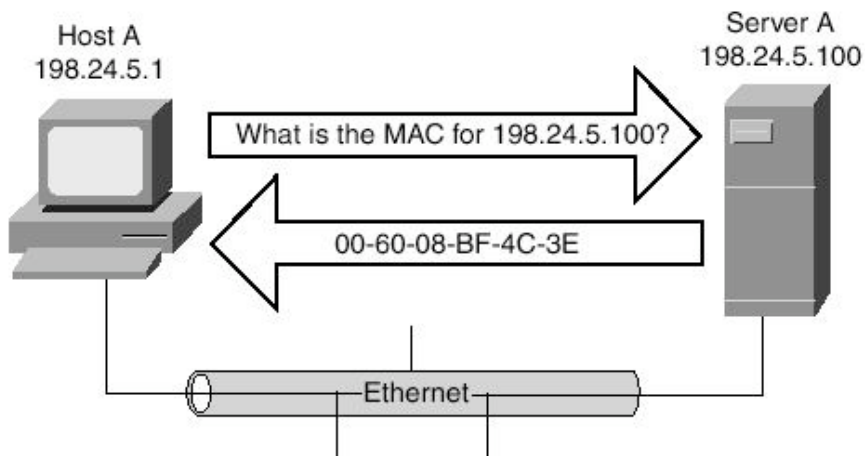
Specifies the location of the Wackamole control socket. Wackamole creates and listens on a unix domain socket. The program **wackatrl** connects via this socket.

## **Mature = <time>**

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



# Wackamole Configuration Basics (.3)



## Address Resolution Protocol

Is the protocol employed to resolve a given IP address on a local subnet to a hardware address for use on the datalink layer.

**The answer is cached.**

**That cache is your enemy.**

## Why?

- Machine A performs ARP to find 198.24.5.100
- Machine A finds 198.24.5.100 at MAC 00:60:08:bf:4c:3e (machine B)
- Machine A caches this answer in its ARP cache
- Machine B crashes
- Machine C takes 198.24.5.100
- Machine A has machine B's MAC in its ARP cache.
- Machine A can't talk to 198.24.5.100

# Wackamole Configuration Basics (.4)

## How do we fix other machines' ARP caches?

Arp-spoofing.

This is technically easy. It involves answering unsolicited ARP requests. Wackamole manufactures “fake” ARP response packets and sends them to important hosts.

## Which hosts are “*important*”?

- Routers?
- Peers in the cluster?
- Can we be smarter than this?

**YES!**



# Wackamole Configuration Basics (.5)

## How do we know who is important?

Any machine that has this particular IP address in its ARP cache.

## Problem:

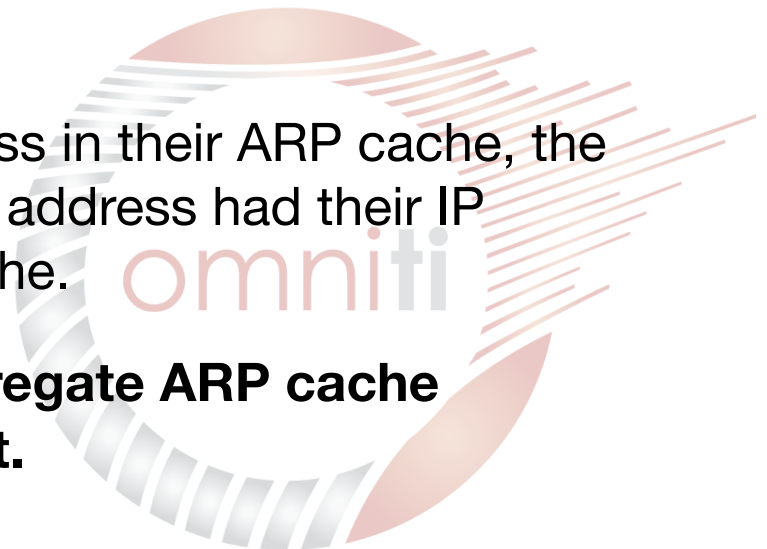
We cannot go look in every machine's ARP cache without special client installed code -- it is infeasible and unmanageable.



## Solution:

If they have this IP address in their ARP cache, the previous owner of this IP address had their IP addresses in its ARP cache.

**Maintain a shared, aggregate ARP cache and notify everyone in it.**



# Wackamole Configuration Basics (.6)

## **Arp-cache = <time>**

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 addresses 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.



# Wackamole Configuration Basics (.7)

```
Notify {  
    <network>  
    <network> throttle <number/second>  
    arp-cache  
}
```

The notification section describes which machines should be sent arp-spoofs when an IP address is acquired.

```
<network>:    <ifname>:<ip>/<mask>  
eth0:192.168.52.0/23  
fxp0:192.168.52.1/32
```



# Wackamole Configuration Basics (.8)

## What is this */n* thing?

Wackmole understands IP addresses CIDR form.  
(Classless Inter Domain Routing)

The *n* refers to the number of set (1) bits in the netmask.

/8	:	255.0.0.0
/16	:	255.255.0.0
/23	:	255.255.254.0
/24	:	255.255.255.0
/27	:	255.255.255.224
/30	:	255.255.255.251
/32	:	255.255.255.255



# Wackamole Configuration Basics (.9)

```
VirtualInterfaces {  
  <interface>  
  { <interface> <interface> ... }  
}
```

Lists all virtual interfaces for which wackamole is responsible.

All wackamole instances in the cluster **must** have all of the same interfaces in the same order.

```
<interface>:  <ifname>:<ip>/<netmask> (CIDR form)  
                eth0:10.77.52.2/32  
                fxp1:10.77.52.3/32
```



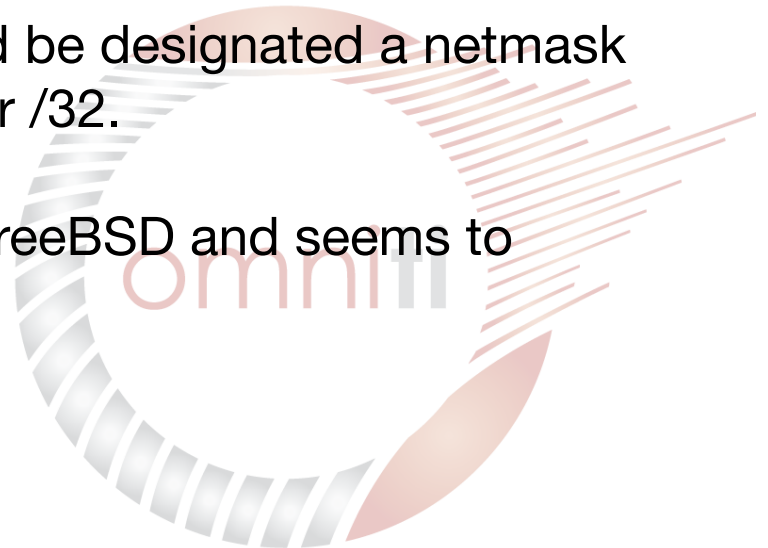
# Wackamole Configuration Basics (.10)

## Why do the examples use /32 as the netmask?

The first IP address configured on a machine in a given subnet should have the netmask and broadcast address representing that subnet.

All subsequent IP addresses in the same subnet on that machine are “virtual” and should be designated a netmask of 255.255.255.255 or 0xffffffff or /32.

This is clearly documented on FreeBSD and seems to hold true for all other OSs.



# Wackamole Configuration Basics (.11)

```
Balance {  
    Interval = <time>  
    AcquisitionsPerRound = <number>  
}
```

**Interval = <time>**

Specifies the length of a balancing round.

**AcquisitionsPerRound = <number>**

Specifies the number of IP addresses that can be assumed in a single round of balancing.



# Wackamole: Example Uses (.1.1)

## Goals:

- Static image and document server for intranet
- 1500 concurrent connections
- 120 Mbs of sustained traffic

## Solution:

- 4 commodity machines
  - 1 x 100Mbs interface card
  - 4 x Ultra2 SCSI drives in RAID 0
- Linux
- Apache (lean and mean)
- Wackamole



# Wackamole: Example Uses (.1.2)

4 Linux machines running Apache and Wackamole, advertising 4 virtual IP (VIP) addresses through DNS.

- If all 4 machines are up, each will acquire and provide service on a single VIP.
- If one machine (A) crashes, one of the 3 remaining will acquire and provide service on the VIP previously managed by A.
- If only one machine remains, it will have acquired and be providing service on all 4 VIPs.



# Wackamole: Example Uses (.1.3)

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

Mature = 5s
Balance {
    interval = 4s
    AcquisitionsPerRound = all
}
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
}
```



# Wackamole: Example Uses (.1.3.1)

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

Spread is running on port 4803 and Wackamole will connect over a local socket.

Wackamole will retry every 5 seconds in the event of a Spread failure.

Once connected to Spread, Wackamole will use the group called **wack1** as a control group.

Wackamole will listen on the local socket called **/var/run/wack.it** for control commands from **wackatrl**.

# Wackamole: Example Uses (.1.3.2)

Wackamole is eligible to assume virtual interfaces after 5 seconds.

Balancing rounds are 4 seconds long.

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

Upon a new group membership, Wackamole will acquire all the virtual interfaces in the final configuration at once.

Every 90 seconds Wackamole will examine the system's local ARP cache and announce all IP addresses found to the cluster through Spread.



# Wackamole: Example Uses (.1.3.3)

Wackamole will not prefer any virtual interface. Wackamole will attempt to balance the virtual interfaces across the currently available machines.

These 4 virtual interfaces are the IP addresses exposed in DNS for the service in question. These are the addresses clients connect to.

```
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 }
}
```



# Wackamole: Example Uses (.1.3.4)

Notify our router 192.168.12.1.

We have to notify our router as it needs to direct traffic to the IP addresses that are brought up by Wackamole. If we effectively notify our router, then all traffic originating from outside the subnet will be able to establish connections to these IP addresses.

Notify the contents of the shared ARP-cache. This helps all the machines on the local subnet “notice” the change.

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



# Wackamole: Example Uses (.2.1)

## Goals:

- Mission-critical router
- IPSEC tunnels for secure VPN
- Firewall rulesets
- NAT for attached private networks

## Solution:

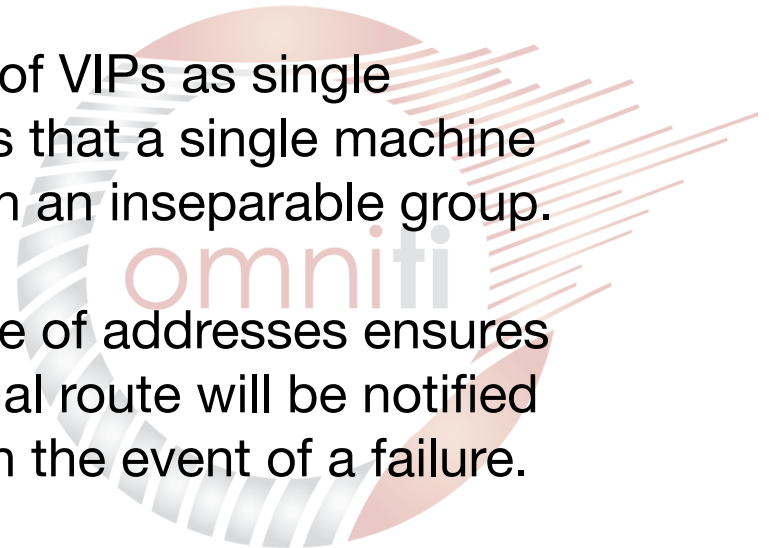
- 2 commodity machines
  - 3 x 100Mbs interface card
- FreeBSD
- Racoon/IPSEC
- ipfw
- Wackamole



# Wackamole: Example Uses (.2.2)

2 machines running FreeBSD, routing between subnets and running ipfw.

- As there is only a single IP address advertised/used as a default route on each subnet, only one machine will “own” the address at any given time.
- Wackamole manages “groupings” of VIPs as single “VirtualInterface” which guarantees that a single machine will acquire all of the routing VIPs in an inseparable group.
- Arp-spoofing to a shared arp-cache of addresses ensures that all machines that use this virtual route will be notified (arp-spoofed) to the change over in the event of a failure.



# Wackamole: Example Uses (.2.3)

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

mature = 5s
balance {
    Interval = 4s
    AcquisitionsPerRound = all
}
arp-cache = 90s
# There is no "master"
Prefer None
# Only one virtual interface
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
    fxp0:255.255.255.255/32
    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
}
```

# Wackamole: Example Uses (.2.3.1)

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

Spread is running on port 3777 and Wackamole will connect over a local socket.

Wackamole will retry every 5 seconds in the event of a Spread failure.

Once connected to Spread, Wackamole will use the group called **wack1** as a control group.

Wackamole will listen on the local socket called **/var/run/wack.it** for control commands from **wackatrl**.

# Wackamole: Example Uses (.2.3.2)

Wackamole is eligible to assume virtual interfaces after 5 seconds.

Balancing rounds are 4 seconds long.

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

Upon a new group membership, Wackamole will acquire all the virtual interfaces in the final configuration at once.

Every 90 seconds Wackamole will examine the system's local ARP cache and announce all IP addresses found to the cluster through Spread.



# Wackamole: Example Uses (.2.3.3)

Wackamole will not prefer any virtual interface. In this configuration that means that no instance *wants* to be “master”. This means that when a new Wackamole instance comes online and matures, it will not “steal” the VIF that is already being controlled by another machine.

There is a single virtual interface with three IP addresses. These IP addresses are the “default routes” on all the respective attached networks.

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



# Wackamole: Example Uses (.2.3.4)

Notification of routers is tricky at times. Many routers do not react the same way to ARP-spoofing.

```
Notify {  
  # Let's notify our upstream router:  
  fxp0:0.0.0.0/32  
  fxp0:255.255.255.255/32  
  fxp0:63.236.106.97/32  
  fxp0:63.236.106.98/32  
  fxp0:63.236.106.99/32
```

The static default route (the uplink) is 63.236.106.97, but that is a virtual router interface of two Cisco Catalysts running HSRP whose real IPs are 63.236.106.98 and 63.236.106.99.

0.0.0.0 is a universal network address and 255.255.255.255 is a universal broadcast address -- both can be used for ARP requests. Some machines will listen to those for ARP responses.

```
}
```



# Wackamole: Example Uses (.2.3.5)

```
Notify {
```

Perhaps the most important machines that we have on our network are our DNS servers.

Any interruption in their availability can cause cascading effects on other caching name servers.

```
# 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
```

66.77.52.4 and 66.77.52.5 are the virtual IP addresses advertised by a hardware load balancer. 10.77.52.18 and 10.77.52.19 are their real static IP addresses.

```
}
```

# Wackamole: Example Uses (.2.3.6)

```
Notify {
```

Notify the contents of the shared ARP-cache.

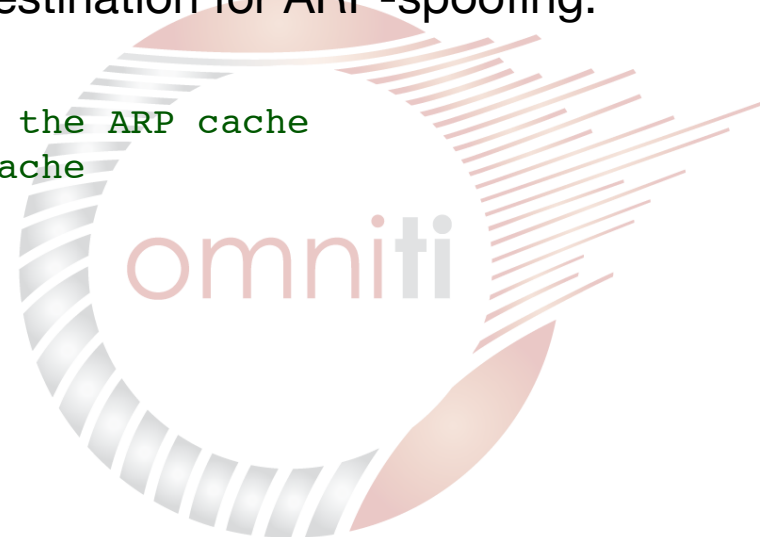
Every time this Wackamole and peer Wackamole instances announce the IP addresses in their respective systems' local ARP-caches, their IP addresses are placed in an aggregate pool.

Each IP in this aggregate pool is used as a destination for ARP-spoofing.

In theory, this should notify **every** machine that needs its ARP-cache refreshed.

```
# And the ARP cache  
arp-cache
```

```
}
```



# Wackamole: Example Uses (.2.3.7)

```
Notify {
```

The addresses specified here are actually IP blocks.

63.236.106.96/28 notifies the 16 addresses in that IP block.

66.77.52.0/24 notifies the class C (256 addresses).

10.77.52.0/23 notifies the 512 addresses  
(10.77.52.0 through 10.77.53.255)

```
# 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  
}
```

# Wackamole: Cool Uses

When pushing out new perl modules and configuration changes, we must restart Apache. These restarts can cause momentary interruption in service.

How can wackamole help?

```
#!/bin/sh

wackatrl -f
apachectl stop
# sync content
rsync -a stage::webroot/ /web/
apachectl start
wackatrl -s
```



# Wackamole: It works

- Wackamole is an effective HA tool
- It has been used in production for 24/7 services
- Wackamole can make any service highly available:
  - Routers
  - Firewalls
  - Web servers
  - FTP servers
  - DNS servers



# HA. Now what?

- Now that we have dissected HA we can approach LB without the itching question: “What if it crashes?”
- The approach of load balancing is **not** to make a system available in spite of failures.
- Load balancing is used to effectively combine the resources of several machines contributing to a single service.
  - To increase performance
  - To increase capacity



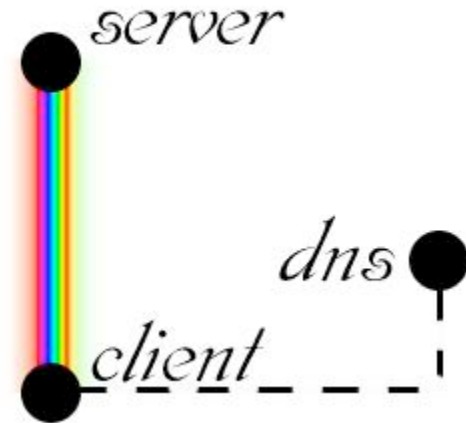
# Load balancing: Practiced Solutions

- Standalone Server
  - ✓ Vanilla Apache server
- DNS Round Robin
  - ✓ Multiple A records in DNS
- Single point proxies
  - ✓ BIG/ip
  - ✓ Alteon, Foundry, Arrowpoint, and Extreme content aware
- Peer-based proxies
  - ✓ mod\_backhand



# Load balancing: or not

- Apache, plain and simple:
  - One server
  - One IP
  - Single point of failure



# Load balancing: DNS RR

- The most simple “load balanced” configuration:
  - Multiple web servers with identical services and content.
  - There is an A record in DNS for that web server for each IP.
  - If a server fails, a portion of the service is unavailable.

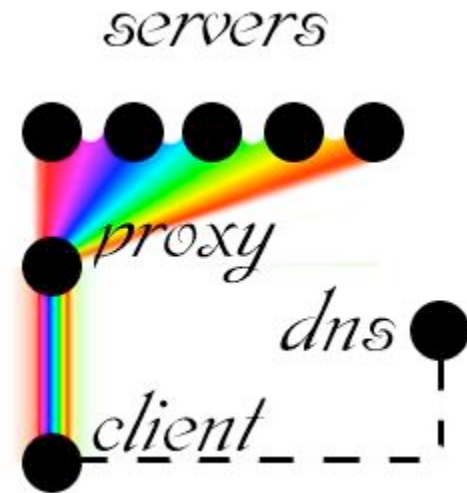


## Sample BIND configuration:

www	300	IN	A	10.0.0.1
	300	IN	A	10.0.0.2
	300	IN	A	10.0.0.3
	300	IN	A	10.0.0.4
	300	IN	A	10.0.0.5

# Load balancing: Single Point Proxy

- The most common “load balanced” configuration:
  - A singular entity represents the cluster as a whole and traffic is allocated by that entity.
  - Most commercial solutions conform to this paradigm.
  - Single point of failure unless an HA solution is applied.



# Load balancing: Peer-based Proxy

- Each node in the cluster is privileged to proxy requests to other nodes within the cluster.
- Extensive resource utilization information is available to each node allowing for more intelligent reallocation decisions.
- Like DNS RR, a single failure will cause a service availability issue.
- This can be solved with Wackamole.



# What is mod\_backhand?

- Effective utilization of resources in a web cluster.
- Handling heterogeneous environments:
  - Resources (Memory, Processing Power, I/O)
  - Hardware (Intel, Alpha, Sun)
  - Platforms (Linux, BSD, Solaris)
- Flexible architecture allowing increased manageability.
- Seamless cluster scaling.
- A convenient means for resource monitoring.
- Open source solution that integrates with Apache.
- Wide area and local area support.



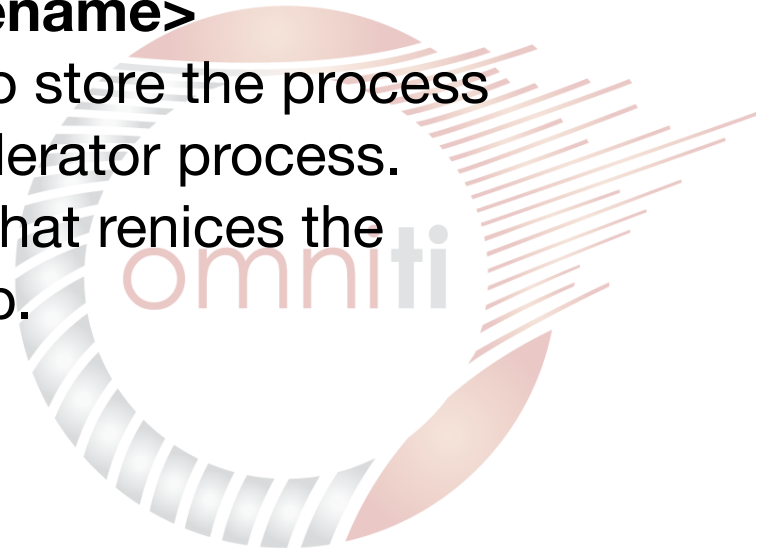
# mod\_backhand: configuration (.1)

## **UnixSocketDir <directory>**

The directory which will house the moderator and children unix domain sockets over which file descriptors are passed (for connection pooling).

## **BackhandModeratorPIDFile <filename>**

Specifies a filename in which to store the process ID of the currently running moderator process. This is useful for a start script that renices the moderator after Apache startup.



# mod\_backhand: configuration (.2)

**MulticastStats <broadcast IP:port>**

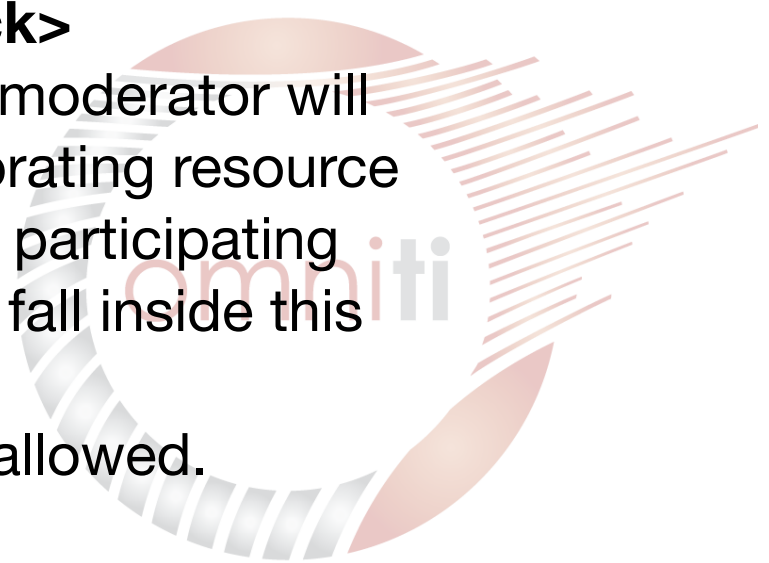
**MulticastStats <multicast IP:port,ttl>**

The IP address and port to which the moderator will announce local resource information.

**AcceptStats <CIDR-form IP block>**

Describes an IP block that the moderator will “trust”. Accepting and incorporating resource information from these IPs. All participating machines in the cluster should fall inside this block.

Multiple AcceptStats lines are allowed.



## mod\_backhand: configuration (.3)

**MulticastStats <myIP: port> <broadcast IP:port>**

**MulticastStats <myName:port> <multicast IP:port,ttl>**

The IP address and port to which the moderator will announce local resource information. However, “local” information will be announced as if from myIP:port or myName:port.

**BackhandSelfRedirect <On|Off>**

Tells mod\_backhand to use proxying even if it finds “itself” to be the best host to service a request. This is useful for running two Apache instances and using the two argument MulticastStats directive.

# mod\_backhand: configuration (.4)

## **BackhandConnectionPools <On|Off>**

Tells mod\_backhand that it should allow the moderator to manage connection to the other machines in the cluster and reuse them from client request to client request.

If set to **Off**, the Apache child will not contact the moderator and the connection will be initiated from the child and terminated immediately after use.

This requires more time per connection, but allows for more concurrency. It should be **On** for low traffic sites and **Off** for high traffic sites.

# mod\_backhand: configuration (.5)

## **BackhandLogLevel** <(+|-)(dcsn|net|mbcs)(all|1|2|3|4)>

This is an advanced feature for troubleshooting problems with the underlying plumbing of mod\_backhand. It enables verbose logging from all the different subsystems of mod\_backhand:

- **dcsn** The decision making subsystem
- **net** The networking subsystem
- **mbcs** The mod\_backhand control system used for child-moderator communications

By default, all logging is disabled.

# mod\_backhand: configuration (.6)

## **Backhand <function> [args]**

Passes the current list of eligible servers (candidates) through the candidacy function **function** which can examine the request and the available resources in the cluster and change that list accordingly.

## **BackhandFromSO <file> <function> [args]**

Same as above, except that **function** is dynamically loaded from the shared object **file**. This allows candidacy functions to be written completely outside of mod\_backhand.

More than one **Backhand** directive can be specified, providing a cascading effect.

# mod\_backhand: configuration (.7)

Directives can pertain only to a specific VirtualHost, Directory, or Location clause.

Allows for decisions based on:

- Document location
- Virtual host
- File extension

## Example:

```
<Files ~ "(\.php)$">
```

```
Backhand byAge
```

```
Backhand byBusyChildren
```

```
</Files>
```

```
<Directory /www/64bitapps/>
```

```
Backhand byAge
```

```
BackhandFromSO libexec/byHostname.so byHostname (alpha|sun4u)
```

```
Backhand byCost
```

```
</Directory>
```

## Machines in cluster:

```
www1.intel.domain.com
```

```
www2.intel.domain.com
```

```
www3.intel.domain.com
```

```
www1.sun4u.domain.com
```

```
www2.sun4u.domain.com
```

```
www3.sun4u.domain.com
```

```
www1.alpha.domain.com
```

```
www2.alpha.domain.com
```

```
www3.alpha.domain.com
```



# mod\_backhand: configuration (.8)

## **Loading the module:**

```
LoadModule backhand_module libexec/mod_backhand.so
AddModule mod_backhand.c
```

## **Main module configuration:**

```
<IfModule mod_backhand.c>
  UnixSocketDir /var/apache/backhand
  BackhandModeratorPIDFile /var/apache/backhand/moderator.pid
  MulticastStats 10.77.53.255:5445
  AcceptStats 10.77.52.0/23
  BackhandConnectionPools Off
</IfModule>
```

## **Status URL:**

```
<IfModule mod_backhand.c>
  <Location "/backhand/">
    SetHandler backhand-handler
    Backhand off
  </Location>
</IfModule>
```



# mod\_backhand: configuration (.9)

## **Balancing a directory:**

```
<IfModule mod_backhand.c>
  <Directory /path/to/directory>
    Backhand byAge
    Backhand byRandom
    Backhand byLogWindow
    Backhand byBusyChildren
  </Directory>
</IfModule>
```

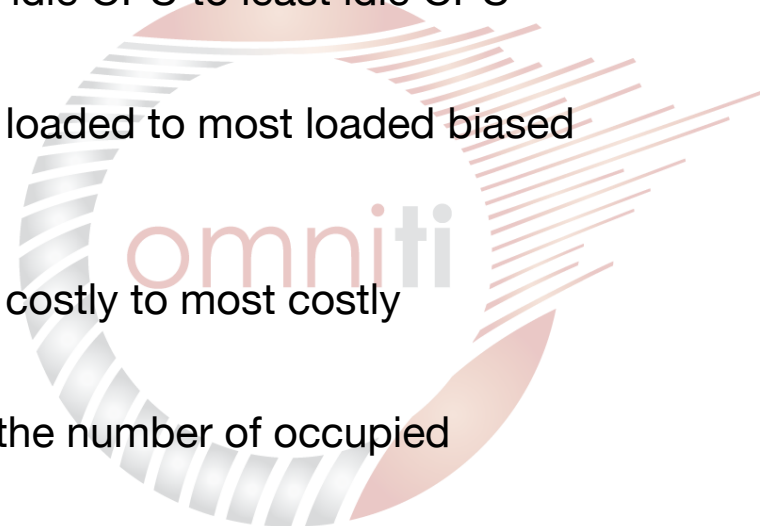
## **Balancing specific file types:**

```
<IfModule mod_backhand.c>
  <Files ~ "(\\.jsp|\\.do)$">
    Backhand byAge
    Backhand libexec/byHostname.so byHostname java
  </Files>
</IfModule>
```

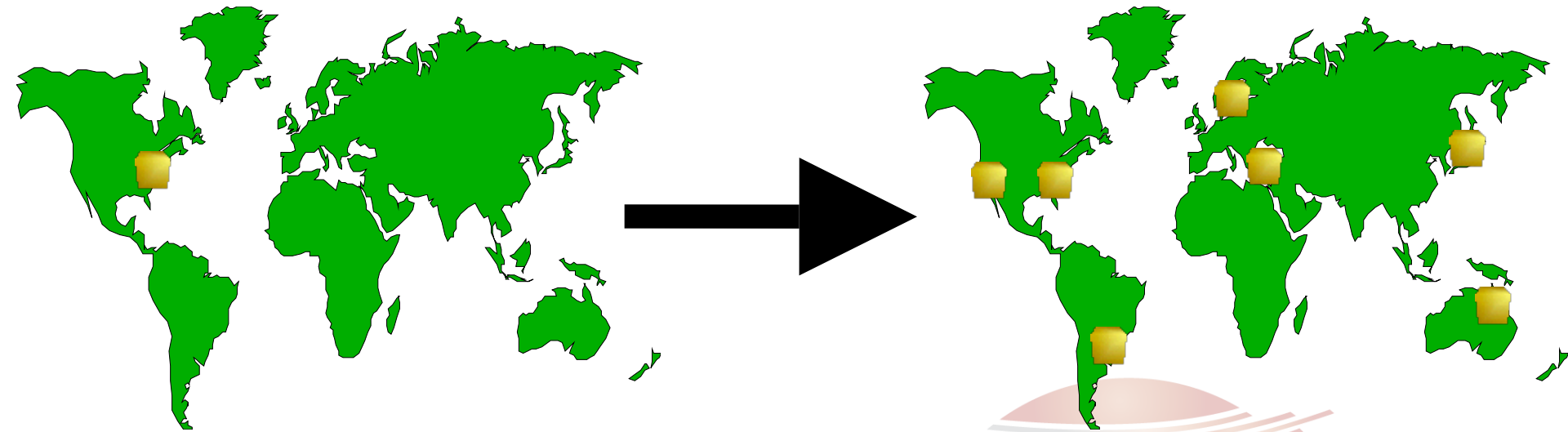


# Flexible Framework: making decisions (.1)

- **byAge [# seconds]**  
remove all servers from whom we have not heard resource information
- **byRandom**  
reorder the existing server list randomly
- **byLogWindow**  
reduce the list of  $n$  servers to the first  $\log_2(n)$
- **byCPU**  
reorder the existing server list from most idle CPU to least idle CPU
- **byLoad [# bias]**  
reorder the existing server list from least loaded to most loaded biased toward yourself
- **byCost**  
reorder the existing server list from least costly to most costly
- **byBusyChildren**  
reorder the existing server list based on the number of occupied Apache children from least to most.



# Flexible Framework: making decisions (.2)



Candidacy functions that cause mod\_backhand to redirect:

- **HTTPRedirectToIP**  
change the proxying mechanism for the candidates to HTTP redirect and redirect to a server's IP address
- **HTTPRedirectToName "format"**  
change the mechanism for the candidates to HTTP redirect and build the name from the format string

# Flexible Framework: making decisions (.3)

How these candidacy functions do their job:

- 1 A list of candidacy functions is specified in the Apache configuration file
- 2 For each request, a complete server list is passed into the first function.
- 3 This function optionally modifies the list of servers and the resulting list is passed to the next function.
  - The Apache request structure can be modified.
  - The method (proxying/redirection) can be selected.
- 4 Repeat step 3 for next function until no more candidacy functions remain.
- 5 The first server in the resulting list is chosen for proxying or redirection.

# Flexible Framework: making decisions (.4)

The API for a decision function:

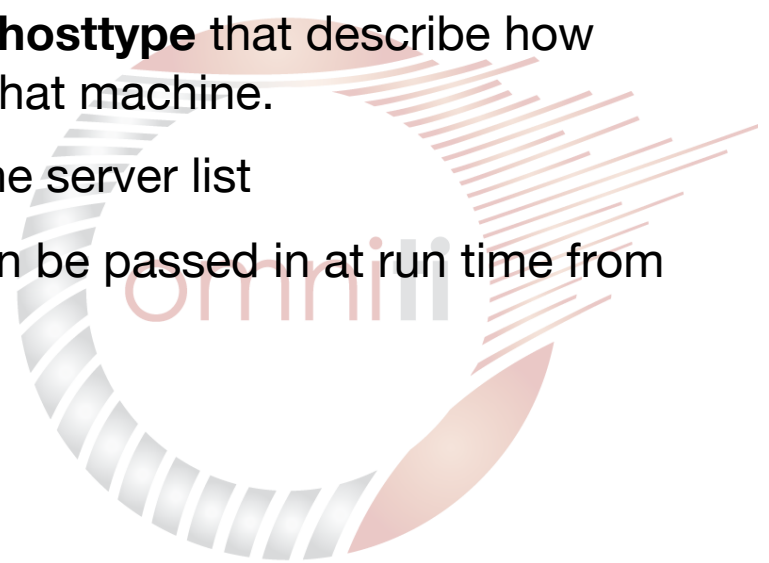
```
int Foo(request_rec *r, ServerSlot *s, int *n, char *arg);
```

**r** is the Apache request record structure

**s** is the server list as a list ServerSlot structures. Each ServerSlot contains an **id** that is an index into the resource information table called serverstats and **redirect** and **hosttype** that describe how mod\_backhand will direct traffic to that machine.

**n** is the number of valid servers in the server list

**arg** is the optional argument that can be passed in at run time from the Apache configuration file

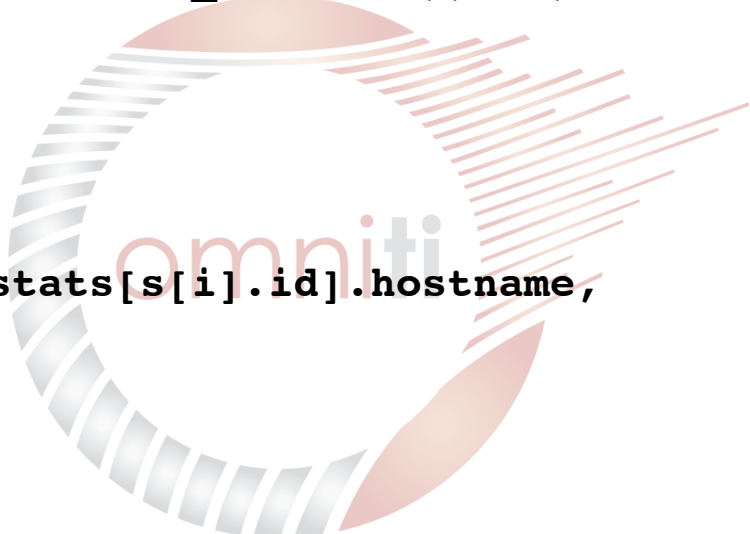


# Flexible Framework: making decisions (.6)

The byHostname candidacy function

```
#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, int *s, int *n, char *a) {
    int ret, i, mycount;
    if(!a) return -1;
    if(!lastarg || strcmp(arg, lastarg)) {
        if((ret = regcomp(&re_matching, arg, REG_EXTENDED))!=0)
            return -1;
        if(lastarg) free(lastarg);
        lastarg = strdup(lastarg);
    }
    for(i=0;i<*n;i++)
        if(!regexexec(&re_matching, serverstats[s[i].id].hostname,
                    0, NULL, 0))
            s[mycount++] = s[i];
    *n = mycount;
    return mycount;
}
```

A large, semi-transparent watermark logo for 'OmniTi' is positioned on the right side of the slide. The logo consists of a circular emblem with horizontal lines on the left and a stylized 'OmniTi' text on the right, all in a reddish-pink color.

# Flexible Framework: internals (.5)

- Resource information includes:

- Hostname
- IP:port for service
- Total/Available memory
- System load
- CPU utilization
- Number of CPUs
- Speed of machine
- Age of more recent information
- Number of available/total Apache children
- Average turnaround time for remote requests



```
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



# Flexible Framework: internals (.6)

- Resource information is collected from the local system every second.
- That information is multicasted to all peers.
- An event loop is used to collect information from peers and it is consolidated in the serverstats structure.
- That event loop is used to manage persistent connection pools assuming BackhandConnectionPools is on.



# Flexible Framework: internals (.7)

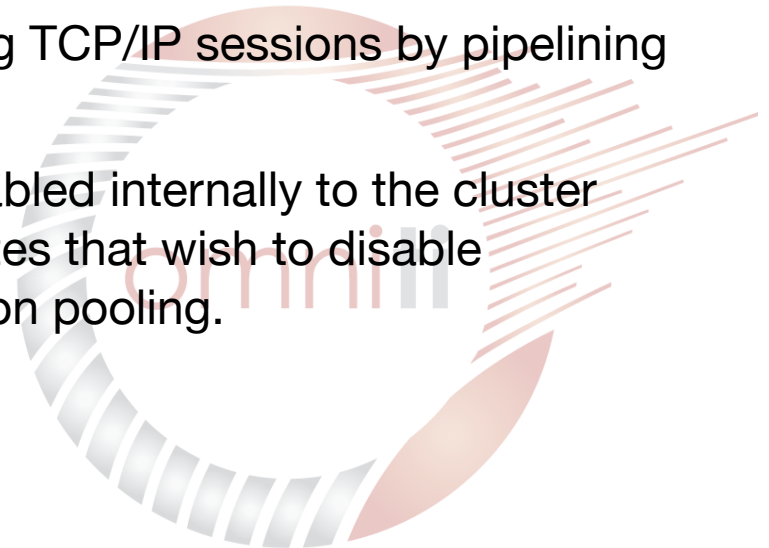
- Connection pooling

Reuse of connections to other servers reduces the overhead of TCP/IP establishment and possible Apache forking.

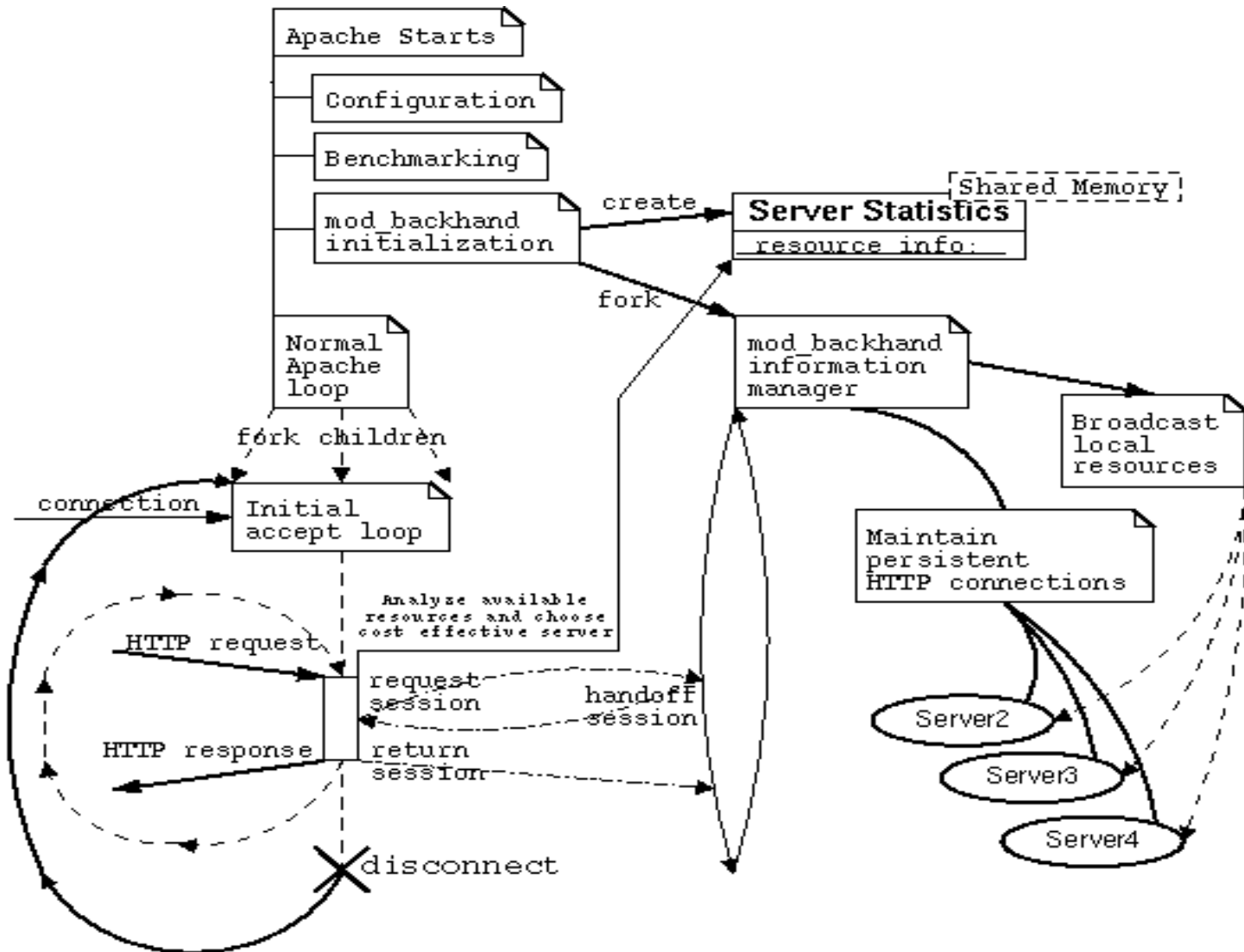
- Keepalive exploiting

Keepalives allow reuse of an existing TCP/IP sessions by pipelining HTTP requests.

If keepalives are unconditionally enabled internally to the cluster (Apache source patch available), sites that wish to disable keepalives can still exploit connection pooling.

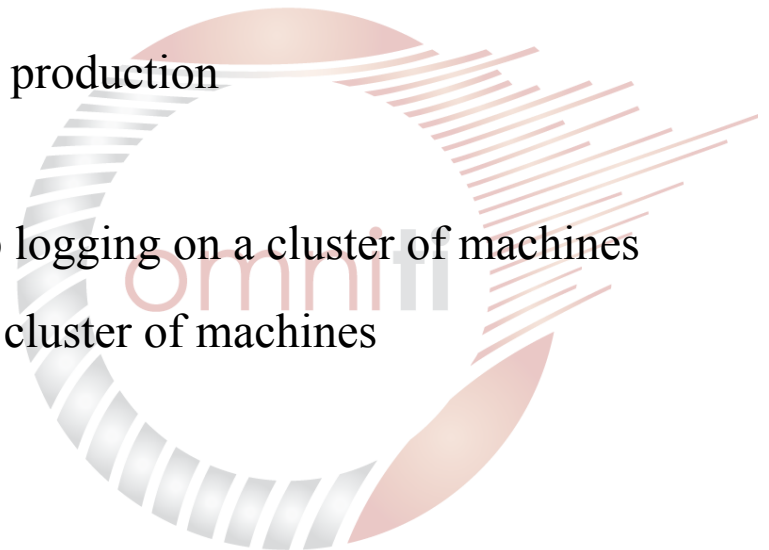


# Flexible Framework: internals (.8)



# HA. LB. Now what?

- Now that we have dissected HA and LB, what remains to be tackled?
- Outstanding problems:
  - Content distribution
    - Production pushes of code
    - Efficiently placing static content in production
  - Logging
    - Unifying/consolidating simple web logging on a cluster of machines
    - Error logging and job logging on a cluster of machines



# Content Distribution

- Most content distributions are based on:
  - A combination of CVS export and rsync
  - Front-end file system-based caches
    - Can be smarter with intelligent cache placement and mod\_rewrite

The details of content distribution systems are out of the scope of this talk.



# Distributed logging

- Distributed logging systems are in their infancy
  - Some systems employ a passive network “sniffing” approach. This is flawed as packets can be dropped, and this behaviour is difficult to diagnose and fix.
  - Some systems use TCP/IP to connect to a logging server. This has a high overhead, can “hang” in the event of a systems failure, and has inherent scalability limitations.
  - Using a multicast message bus is the “right” approach. `mod_log_spread` is such an implementation.



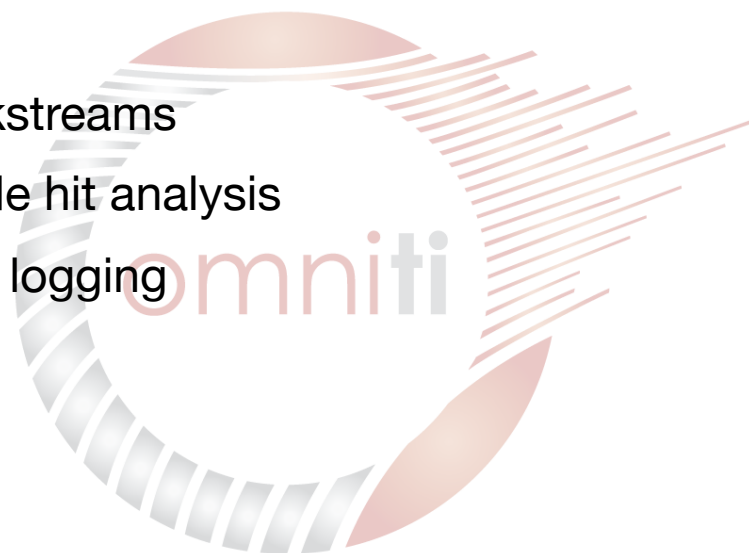
# Distributed logging: what we need

- What we need:
  - Apache integration and a flexible and **easy** API
  - Reliable and ordered delivery of logs
  - Efficient network utilization
  - Efficient handling of thousands of concurrent publishers
  - Multiple subscribers with minimal overhead
- What we use:
  - A group communication system (Spread)
  - A log journaling program (spreadlogd)
  - Spread client API (C, python, ruby, PHP, Perl)
  - An Apache module (mod\_log\_spread using C API)
  - mod\_perl logging handler (Apache::Log::Spread)

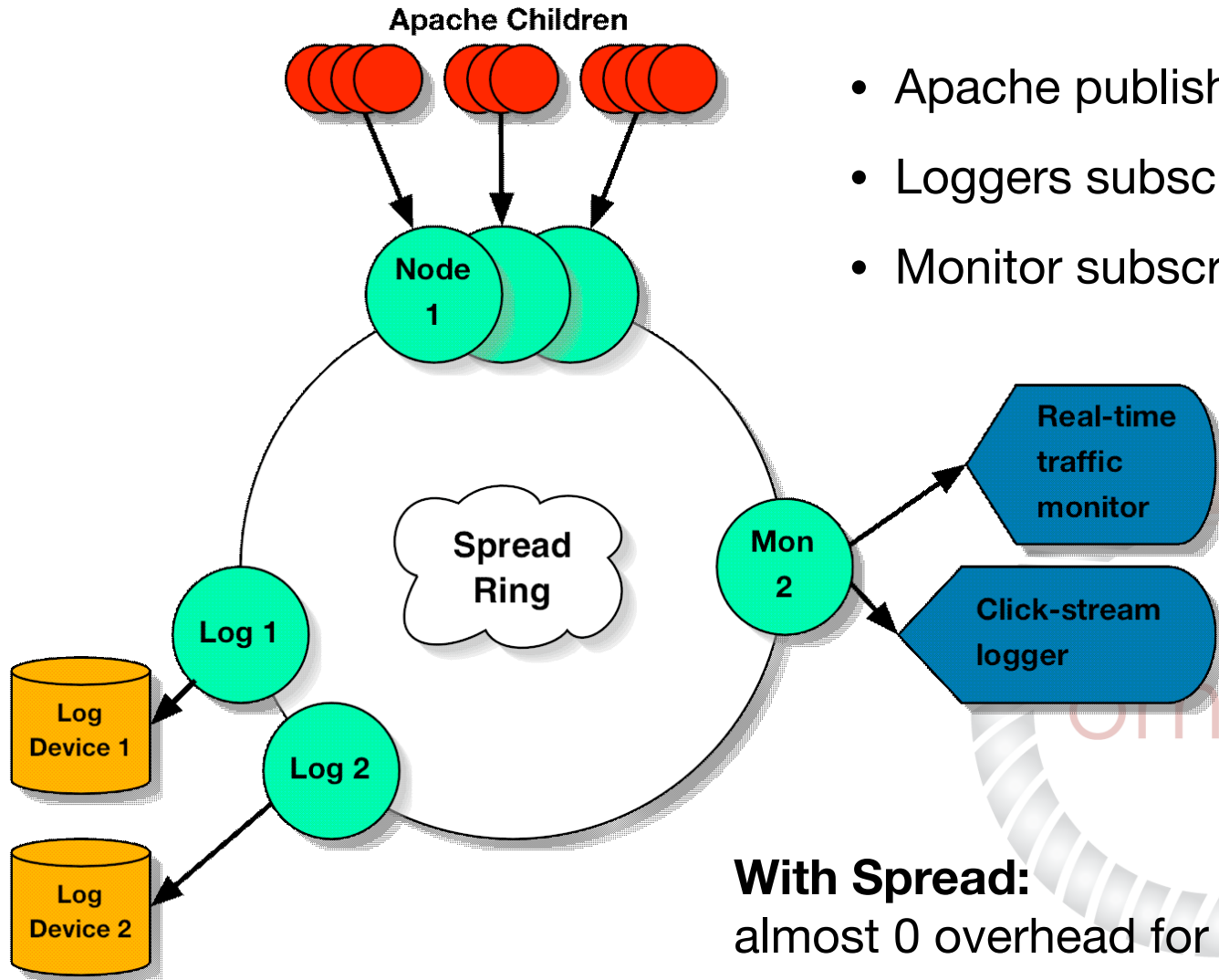


# Distributed logging: what we get

- Unified logging for local area clusters
- No drop semantics
- Ultra-convenient integration into an existing configuration
- Multiple subscribers with little overhead
  - Fault-tolerant log writers
  - Process real-time cross-cluster clickstreams
  - Real-time per server and cluster-wide hit analysis
  - Real-time individualized mass vhost logging
  - ALL AT THE SAME TIME!



# Distributed logging: concept (.1)



- Apache publishes logs to groups
- Loggers subscribe to those groups
- Monitor subscribes to groups

**With Spread:**  
almost 0 overhead for multiple subscribers

# Distributed logging: how its done

- Spread is installed on every web server and every log server
- Download and install mod\_log\_spread
  - [http://www.backhand.org/mod\\_log\\_spread/](http://www.backhand.org/mod_log_spread/)
  - [http://mirrors.omniti.com/mod\\_log\\_spread/](http://mirrors.omniti.com/mod_log_spread/)
- Compile/Install mod\_log\_spread
- Add to httpd.conf:

```
LoadModule log_spread_module libexec/mod_log_spread.so
AddModule mod_log_spread.c
SpreadDaemon 4803
CustomLog $site1 combined
```
- Install spreadlogd (comes with mod\_log\_spread)
- Configure spreadlogd

*... and let 'er rip!*



# Distributed logging: configure spreadlogd

```
# This is a sample spreadlogd.conf file
```

```
BufferSize = 65536
```

```
Spread {
```

```
  Port = 4903
```

```
  Log {
```

```
    RewriteTimestamp = CommonLogFormat
```

```
    Group = "site1"
```

```
    File = /data/logs/apache/www.site1.com/combined_log
```

```
  }
```

```
  Log {
```

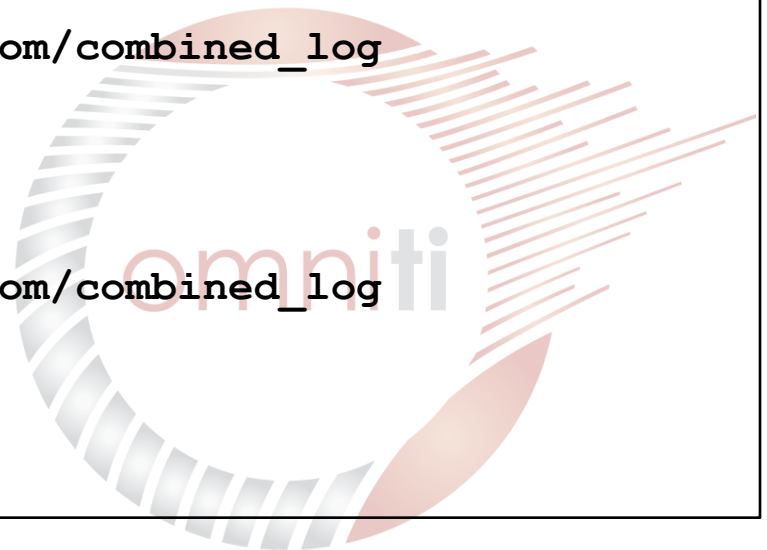
```
    RewriteTimestamp = CommonLogFormat
```

```
    Group = "site2"
```

```
    File = /data/logs/apache/www.site2.com/combined_log
```

```
  }
```

```
}
```



# Distributed logging: other cool things

- You can run 1, 2 or 10 spreadlogd instances on separate logging servers without incurring more network traffic.
- Writing other applications that listen to the logging stream for real-time statistics:
  - Hit rates
  - Click-thru rates
  - Who's online lists



# Summary

- LB is complicated and very specific to your application. If someone claims to offer the best solution, they are lying -- or misrepresenting the truth.
- HA clustering solutions are often combined with LB solutions. This isn't bad, but can confuse the topics.
- Peer-based HA solutions are powerful alternatives to the classic front-end device. They may be better for many applications.
- Distributed logging is not a truly complicated problem, but `mod_log_spread` is the best solution available. Period.
- Content distribution is complicated -- good luck.

# Conclusion

The Backhand Project provides solid, tested solutions for many of the complications found when building clusters to serve today's Internet Applications.

As open source technologies, they all lend themselves to further extension, robustification and multi-platform support.

The tools presented provide solutions that have been used **successfully** to build enterprise, mission-critical architectures.



# Credits

## **The Johns Hopkins Center for Networking and Distributed Systems**

who funds the core development of tools such as:

Spread, mod\_backhand and wackamole.

## **OmniTI Computer Consulting, Inc.**

who funds development of:

mod\_log\_spread, spreadlogd, wackamole, mod\_backhand, tons of cool Perl and PHP stuff and countless other open source projects.

and for providing an excellent work environment with a constant stream of challenging and exciting problems.

