

mod_backhand

use your resources

INTERNALS EXPLAINED

THEO SCHLOSSNAGLE <jesus@cnds.jhu.edu>

APRIL 5TH, 2001

Abstract

A technical explanation of the inner workings and implementation of mod_backhand will help better illustrate the various extents to which it can be configured and used. Details on optimizing Apache in conjunction with mod_backhand will be discussed. The intelligent configuration of mod_backhand's transparent protocol upgrades can be used to accelerate multi-tiered application architecture. Some out-of-the-box content distribution methods and examples will be presented.

1 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.

2 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.

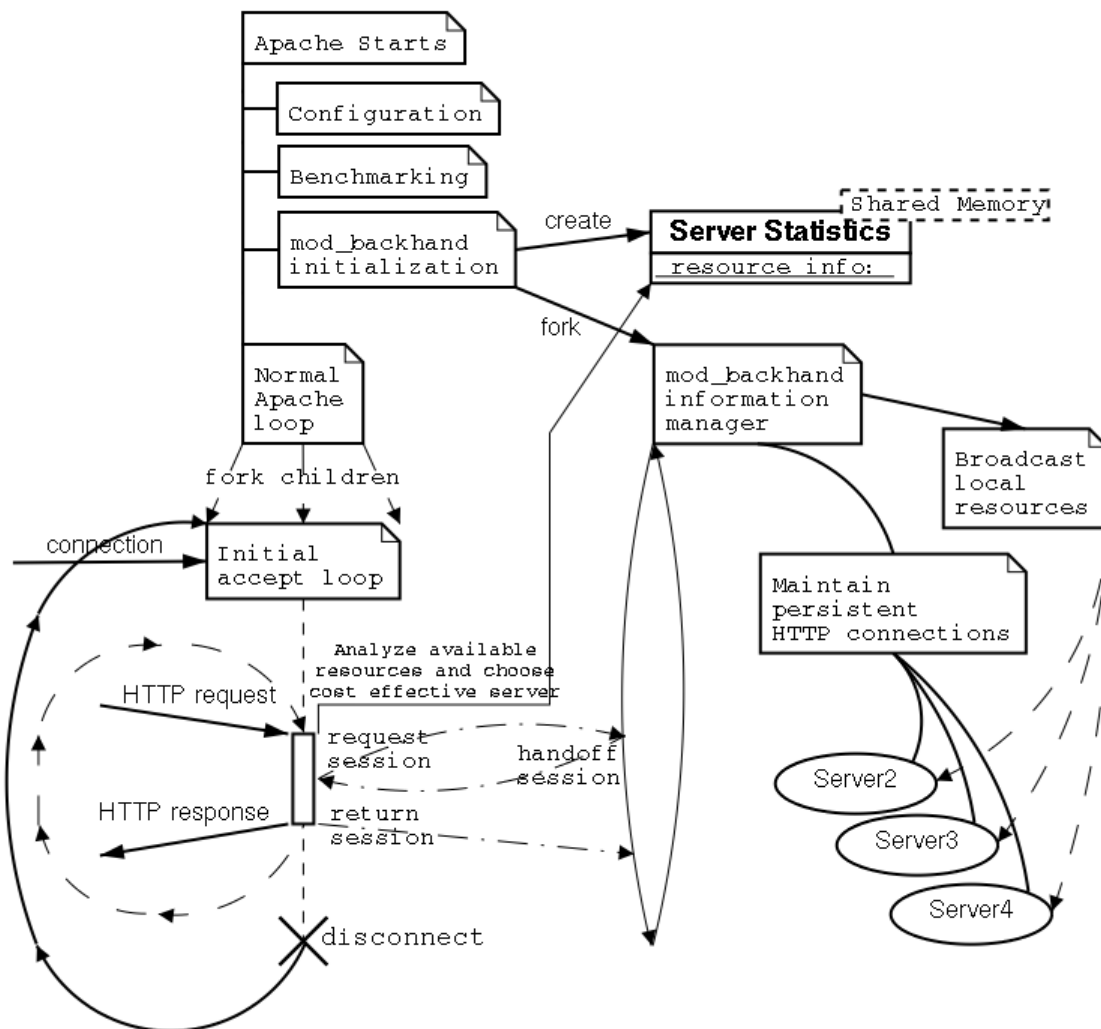
mod_backhand starts a separate moderator process at module initialization that will be responsible for several tasks, one of which is resource collection and distribution. This process also manages connection pools to peer web servers in the clusters.

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.

3 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.

3.1 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

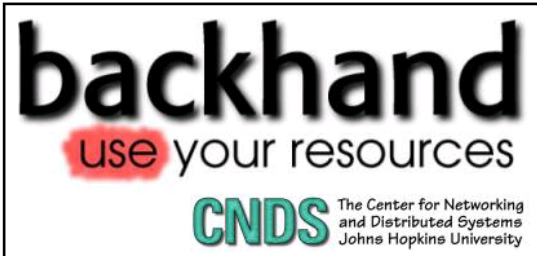
module initialization and is continually populated by the moderator process. As it is created at module initialization, all of the Apache children are able to see it – they inherit the attachment.

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”



Local Machine Name: fog13.cnds.jhu.edu
Apache Version String:
 Apache/1.3.12 (Unix) balanced_by_mod_backhand/1.1.1pre3
Server built: Feb 7 2001 17:00:33
REMOTE_ADDR: 216.0.51.145

Entry	Hostname	Age	Address	Total Mem	Avail Mem	# ready servers/ # total servers	~ms/req [#req]	Arriba	# CPUs	Load/HWM	CPU Idle
0	fog13.cnds.jhu.edu	0	128.220.221.113:80	263700480	161514688	150/150	0 [0]	859809	2	0.000000/256	0.996000
1	fog9.cnds.jhu.edu	0	128.220.221.109:80	263700480	166274240	148/150	0 [0]	858204	2	0.000000/256	0.996000
2	fog14.cnds.jhu.edu	0	128.220.221.114:80	263700480	159118528	150/150	0 [0]	858312	2	0.000000/256	0.996000
3	fog10.cnds.jhu.edu	0	128.220.221.110:80	263700480	165725376	150/150	0 [0]	858557	2	0.000000/256	0.996000
4	fog16.cnds.jhu.edu	0	128.220.221.116:80	261623808	193549504	150/150	0 [0]	859386	2	0.000000/256	0.996000
5	fog15.cnds.jhu.edu	0	128.220.221.115:80	263700480	168273088	150/150	0 [0]	854075	2	0.000000/256	0.996000
6	fog11.cnds.jhu.edu	0	128.220.221.111:80	263700480	172516544	150/150	0 [0]	859232	2	0.000000/256	0.996000

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.

3.2 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.

3.3 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 already exist in the list. If you are using a peer-based topology, you want to consider yourself most often, and some configurations (`byRandom`, `byLog-Window`) 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 On-line 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.

3.4 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(!regexexec(&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.

3.5 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`.

4 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.

4.1 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 `MulticastStats` directive. There are four forms of this directive:

1. `MulticastStats broadcastaddr:port`
2. `MulticastStats myaddr:port broadcastaddr:port`
3. `MulticastStats IPmulticastaddr:port,tll`
4. `MulticastStats myaddr:port IPmulticastaddr:port,tll`

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.

4.2 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.

4.3 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.

4.4 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 than 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.

There exists a patch to Apache 1.3.x that borrows heavily from the implementation of `mod_backhand` which provides a moderator that will alleviate the above described problems with keep-alives. It accepts connection via a single process and once information is available to read (the request), it is allocated to a child. After the request is serviced and no new request is readily available, the child will pass the connection back to the parent for "holding." This patch is called the `PARENTAL_ACCEPT` patch and is available from <http://www.omniti.com/~jesus/projects/>.

5 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 in 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.

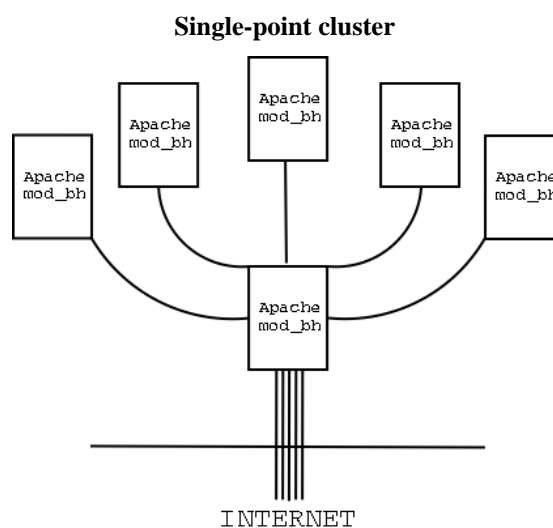
Issue 4 is precisely the reason that the byLoad function and the byBusyChildren function have a “bias” ar-

gument. It will allow a server to prefer serving the request to proxying to a server with the same or slightly more available resources.

6 Example configurations

6.1 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.



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_backend` 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_backend` makes it very easy to balances dif-

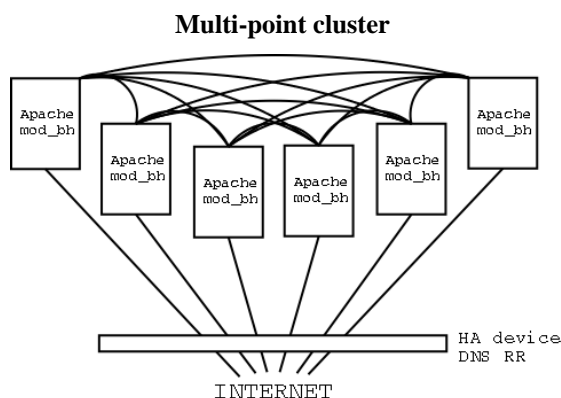
ferent content using different rules. Assume we use `mod_perl+Apache::ASP` and `mod_php` and we wish to balance that content based on system load. All other content should be served by the local machine. The following configuration will balance all “.php” and “.asp” to one of the back-end servers based 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.

6.2 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.



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:

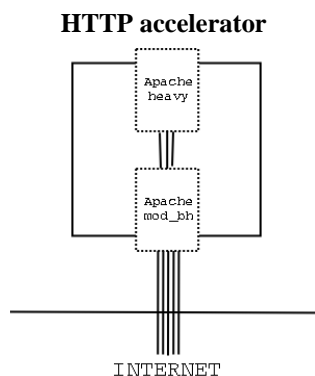
```
<Files ~ "(\\.asp|\\.php)$">
  Backhand byAge
  Backhand byRandom
  Backhand byLogWindow
  Backhand addSelf
  Backhand byLoad 2.0
</Files>
```

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.

6.3 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.



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 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 7) 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.

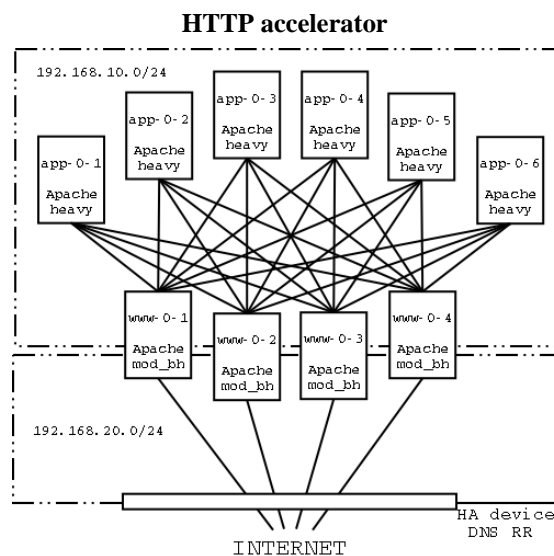
6.4 Two-tier, multi-point cluster

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 6.1 and 6.2, but adds the concept behind separate instances for acceleration presented in section 6.3 to create a flexible and efficient two-tier web cluster architecture.



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 interchangeable. 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>
```

7 Conclusion

Web users are demanding more complicated services and web sites are becoming more powerful to meet the demand. Only so much of that evolution can employ the end-users' resources. More and more server-side resources are required to provide these new services to clients. As more resources are required, high availability and manageability must remain in focus. Hardware load-balancing devices are still expensive and not as flexible as software solutions. As multi-tier architectures are more frequently deployed, both high availability and manageability become more difficult due to the inherent separation of the tiers. High availability solutions must be deployed between each tier, or the applications must be robust and fault tolerant. Even then, load-balancing each tier becomes a serious challenge.

`mod_backhand` provides a reliable, robust, flexible and cost-effective infrastructure for deploying large

Apache clusters. `mod_backhand` is young, but has been deployed successfully in several relatively large installations. As `mod_backhand` evolves, it stands to become a core component in large web cluster installations.

8 Credits

The Apache web server is developed by the Apache Software Foundation by a team too large to list here.

The `mod_backhand` Apache module is developed at The Center for Networking and Distributed Systems at the Johns Hopkins University by Yair Amir, Baruch Awerbuch, Ryan Borgstrom, and Theo Schlossnagle.

The parental accept patch to the Apache web server is developed at OmniTI, Inc. by Theo Schlossnagle.

Thank you to all of the `mod_backhand` user community and other supporters.