



The Backhand Project: load-balancing and monitoring Apache Web clusters

Introduction

The Backhand Project was started at The Johns Hopkins University in the Center for Networking and Distributed Systems. Its main goal is to tackle some of the more pressing difficulties in the area of high-performance replication systems. Some of these areas include distributed databases, distributed file systems, resource allocation on both the local area and wide area clusters, and tools to allow seamless clustering.

The first and most prominent component of the Backhand Project is `mod_backhand`. `mod_backhand` is a dynamically loadable module for the Apache web server that aims to seamlessly allow for load-balancing within a cluster of machines. This cluster is not limited to a local area grouping of webservers. The purpose of `mod_backhand`, specifically, is to provide the infrastructure to reallocate HTTP requests to any machine within a cluster and the framework for effective and flexible decision-making.

The concept behind `mod_backhand` is the amalgamation of several important deviations from "standard" practice. Many network appliances used for balancing web clusters have their foundations in the networking world. This leads to a design drawn in the image of a router. This approach was purposely avoided in an attempt to avoid bottlenecks and single points of failure. The approach we use allows for maximum utilization of a network's egress points and survives link failures extremely well. The flexibility of the implementation provides a tool that can be used to build both a single and a multiple entry point cluster.

`mod_backhand`, in its current form, attempts to solve resource allocation and management issues within a cluster of machines on both relatively low latency, high-throughput networks and wide-area, high-latency networks.

Monitoring and analyzing specific performance metrics can be difficult on a single machine and near impossible on a cluster of machines. The problem is directly related to the inability to watch individual events happen in real-time on all machines in one central place. This motivation along with the effort involved recombining http log files from multiple machines led to the development of the `mod_log_spread` Apache module.

`mod_log_spread` is an augmentation of the core Apache `mod_log_config` that uses the Spread group communication system as a messaging bus to publish events instead of writing directly to a file or pipe. The advantage of using Spread is efficiency, simplicity (of use) and the ability to have multiple subscribers without increasing the network utilization.

In this paper, we will introduce the concepts behind load-balancing, monitoring clusters for performance, and two modules for Apache that will make these tasks easy.

Problem

There are many complicated problems involved in load balancing a web cluster. Just a few of the major issues include: administration complications, centralized logging, monitoring, regular file consistency amongst servers, algorithms for balancing load, and the actual mechanism for balancing load. `mod_backhand` does not attempt to deal with centralized logging or information consistency of regular file systems or databases. They are out of the scope of this module. The issues that `mod_backhand` does attempt to cope with are those of mechanics and algorithms.

Currently, most available tools for analyzing clusters of webservers are either offline or based on low-level network metrics. These tools are all useful, but the looking at events after they happen doesn't allow you to perform online, proactive performance tuning because the event has long since ended. Network metrics are very useful for finding networking problems, but they are at a disadvantage without

knowledge of what is happening on an application level. An efficient, high-performance, scalable architecture for announcing events in real-time adds the missing component. `mod_log_spread` attempts to cope with all of the obstacles involved in centralized logging and monitoring of local area clusters.

Solution

We will attack the problems of load-balancing and distributed logging in this paper, but the implementations that provide the solutions are individual and separable. `mod_backhand` and `mod_log_spread` are completely independent in operation and are fully functional when used individually. The reason for their joint presentation in this solution is because, more often than not, when multiple servers are required to drive a web real estate, both load-balancing and distributed log handling are required.

First, we must clarify a common misunderstanding. The terms "load-balancing" and "high-availability" are often used interchangeably to mean both. However, they are two completely separate and fundamentally different problems. However, in most situations, technologies must be employed to handle both problems. For this reason, industry attempts to tackle both problems with a single black-box product. Some examples include "load-balancing" products from Cisco, Arrowpoint, Alteon, Foundry, and F5. They all tackle the problems with varying methods. All of these products advertise themselves as load-balancing solutions, but they each handle high-availability much better.

High-availability assures an incoming request will be delivered to a server that is capable of responding. Load-balancing means that at any given point in time, the next request delivered to the system will be delivered to a machine such that the overall resources of the cluster are efficiently allocated. `mod_backhand` does not claim nor attempt to provide high-availability for a cluster of Apache servers.

This immediately leads to the question: "Product X gives me high-availability and load-balancing while `mod_backhand` only provides load-balancing. Why would I use `mod_backhand`?" The tone of this question implies that you can use product X or `mod_backhand`, but not both — wrong. As discussed, product X provides high-availability, which is absolutely necessary in your architecture. But, how well does it perform load-balancing? Load-balancing and, more directly, online algorithms for resource allocation are academically hard problem. `mod_backhand` can work in conjunction with product X to correct inappropriately allocated requests.

`mod_backhand` attempts to compensate for the weaknesses in earlier architectures through simple, selective combination. On top of previous architectures a flexible framework for decision making is used to intelligently reallocate requests.

One of the fundamental advantages that `mod_backhand` has over other TCP/IP based solutions is the understanding and processing of streamed HTTP requests. Even solutions that have the ability to "look into" the HTTP request enough to make "more intelligent" decisions (via 3-way TCP handshakes or other mechanisms) still make their decision for the TCP sessions. HTTP/1.0 and later specify the ability to pipeline HTTP requests over a single TCP sessions. The decision made for a TCP session based on the first HTTP request in the pipeline will also be made for the n HTTP requests following it without regard for the requests themselves or the currently available cluster resources.

The serious issue that `mod_backhand` attempts to tackle is a resource allocation problem. No web load balancing infrastructure, other than `mod_backhand`, uses both request content and resource utilization information to make decisions on a request-by-request basis.

When looking at the overall concept of web server, we see that by servicing a request, we dedicate resources on the network and on the server that responds. Once the transaction is complete, those resources become available. The more resources a server has, the more quickly it can respond to any given request. So, assigning requests to machines on the basis of information other than resource utilization information seems a bit confused.

Knowing the request and resources of every machine in a cluster, which machine is "best" to service a request? This is an open area of research, and we are still investigating various algorithms. We realized that `mod_backhand` would be the first web cluster load balancing solution to use resource utilization information in the decision making process, so we accounted for this in the decision making framework. This framework is vital for the ongoing research in the area of online resource allocation of HTTP requests.

As `mod_backhand` uses actual resource utilization metrics, there are several complications that

arise. One specifically challenging problem is the handling of stale resource information. For example, if resource utilization information is shared once every second and there is more than a single request per second then all subsequent requests (after the first) will be based on the same, now stale, resource information. So, in an extreme case, if you choose the least loaded machine and receive 1,000 requests in a single second, then all 1,000 requests will be delivered to the same machine. On the other hand, if you simply assign each request to the next machine in line (round-robin) then requests are being assigned to a machine without regard to its capacity.

Other methods include weighted round-robin, least connections, fastest response time and random. All of these methods have nothing to do with the request itself and nothing to do with the resource available on the various machines in the cluster. Though they use some metric that in some fashion reflects available resources, they do not account for individual available resources on the machines like memory and system load.

We need resource utilization information available on the same time scale as our request times. Resource allocation techniques for scheduling jobs on clustered servers have been researched for many years. Though the same theory applies directly to the problem at hand, the short job length makes previous practical solutions inapplicable. In particular, handling requests as dynamically migrateable entities is not appealing due to the short duration of the requests. On the other hand, the ability of `mod_backhand` to analyze the requests before they execute seem to allow a better use of profiling techniques. This, among other things, is an on going research.

This delivers us to our next fundamental problem. Are things working right? Are all servers in the cluster serving pages? Are any of the machines serving error pages more? Are they pushing equal bandwidth? All of these questions are very hard to answer without a real-time, distributed logging mechanism.

A distributed logging mechanism requires an efficient, high-throughput, and reliable messaging bus. If one wishes to log the events that are announced as well as monitor various aspects in real-time, then a system that uses publish/subscribe semantics is required. Also, it is very important that multiple subscriptions to the same event stream do not incur severe overhead. For this reason, Spread was chosen as the underlying messaging system for `mod_log_spread`. Spread is a high-performance and robust group communication system with strict semantics developed at The Center for Networking and Distributed Systems at The Johns Hopkins University.

`mod_log_spread` is a set of patches to `mod_log_config` that will allow the writing of apache log messages to Spread groups in addition to standard files, syslog and pipes. This allows any number of utilities to simply connect to the local Spread cluster and subscribe to the Apache logging events that are being published by the web servers running `mod_log_spread`. The programs can be as simple as `log_writer` that writes what it hears to disk producing a unified, time-sequential log for the entire cluster or more complicated monitoring suites that analyze the actual information to calculate real-time metrics.

Reliability and ordering are both of vital importance to companies that rely on their logs for revenue and analysis. Syslog is not sufficient for this purpose as guarantees neither reliability nor ordering. As web architectures become more complicated, understanding the order of events is vital when tracking down bottlenecks and bugs. In a clustered configuration, one request will be satisfied by one machine and the next request will be satisfied by another machine. This can make it impossible to determine the exact order of events.

`mod_backhand` and `mod_log_spread` provide an infrastructure to enable clusters of Apache web servers to operate as a clean and efficient distributed system. In addition, they both work as compliments to other web replication tools.

Implementation

The implementation of `mod_backhand` can easily be split into several components:

- The resource information manager (RIM),
- The resource maintenance assistant (RMA),
- The diagnostic tool (DT),
- The decision maker (DM),
- The reallocation mechanism.

Each of the components interacts closely with the others to provide a framework for informed resource allocation decisions. A technical overview follows; for more detailed information see *mod_backhand: A load balancing module for the Apache web server* in the ApacheCon 2000 (Orlando, FL) conference proceedings.

The resource information manager is responsible for acquiring the platform specific system resource utilization information and multicasting it to other machines within the cluster. The method of multicasting the information is done via unreliable UDP IP broadcast or IP multicast. The various announcements from machines in the cluster are combined by the resource maintenance assistant into a resource utilization matrix on each machine. This matrix should be close to identical on all of the machines participating in the *mod_backhand* load-balancing. The RMA is also responsible for maintaining a pool of active connections to other web servers in the cluster. With an active pool of connections, the overhead of establishing a TCP/IP connection to a web server to proxy a request is avoided by reusing active connections.

Both of these components coexist in a single daemon process that is separate from the main Apache process. This process is started at module initialization time when the web server started. The resource utilization matrix is stored in shared memory in the same fashion that the Apache scoreboard is stored. This allows for seamless read-only access to this information to the child Apache processes during the decision making phase. The connection pool is maintained in this separate process so that the majority of available file descriptors can be used for inter-server connections. The connections "loaned" to Apache children processes via IPC and then returned to the RMA once single HTTP transaction is handled.

The diagnostic tool is an Apache content handler. It acts in a similar fashion to the *server-status* content handler that is provided as a module in the Apache distribution. When activated as a URL within a `<Location>` directive, it will yield an HTMLized matrix detailing the current resource utilization matrix being maintained by the RMA. This URL is extremely useful for monitoring complete systems resource availability for your entire cluster -- at a glance.

When the Apache web server processes an HTTP request, it cycles through all of the modules at various stages. In the first possible stage, *mod_backhand* determines whether or not load-balancing is desired, administratively, for this URL. After that it passes the request information through a chain of decision making functions. We call these functions candidacy functions. These functions are responsible for weaning the set of available servers into an ordered set of the "best" servers. The candidacy functions have access to the `request_rec` structure for the current request as well as the shared memory segment that holds the resource utilization information maintained by the RMA. The functions in this chain are defined in the Apache configuration file and can be different for different directory or extensions. The candidacy functions can also chose the method of request reallocation to be used.

Because the framework for decision making was designed to be extremely flexible, candidacy functions can be compiled outside of *mod_backhand* and loaded dynamically at run-time. This make for extremely simple development and deployment of customized decision making functions.

Once the chain of candidacy functions has executed on a given request, we are left with a list of servers. The *mod_backhand* redirection handler is called and executes the actual request reallocation to another server. There are currently two methods of request reallocation supported: HTTP redirection and HTTP proxying.

When using HTTP redirection, the appropriate hostname is composed and the client browser is issued an HTTP 302 temporarily moved response, which directs them to contact the other server for their information. The built in function that provides HTTP redirection will, for example, redirect and incoming request to `www.example.com`, that arrives at `www-3.example.com` to `www-5.example.com`, but will not redirect requests to `www-3.example.com` as they have been delivered to this specific server.

When using HTTP proxying, *mod_backhand* does much more work. First, it requests an active connection to the first machine in the candidacy list that was returned from our chain candidacy functions. The request goes to the RMA and the RMA responds by handing the Apache child an open file descriptor to that machine (establishing one first if no active, unused connection exist in the pool.) The child then upgrades the request incoming request to use HTTP keepalives. If the request is already requesting keepalives, then no augmentation is made. The request is then proxying to the new server and the result is passed back to the client. Once complete, the client connection is closed if the original request did not request keepalives. The connection between the servers is still active, as the HTTP protocol used requested

keepalives, and is thus returned to the RMA to be held in the connection pool.

Keepalives are commonly disabled on large Internet web sites and the Apache 1.3.x architecture requires one child per active connection. Assuming keepalives were disabled; if 10,000 requests were made over a 5 second period and each request took approximately 30ms, then 60 Apache children would be required. On the other hand, if keepalives were enabled the same transaction set could require up to 10,000 Apache children. This is a worst case scenario, but then again, 10,000 concurrent processes are not feasible on most modern operating systems. For this reason, keepalives are often disabled in Apache for large sites expecting extremely heavy traffic arrival rates.

As mod_backhand greatly benefits from keepalives being enabled (at least between the servers,) there is a small patch available for the 1.3.x source tree that will override the disabling of keepalives for connections that are being proxying internally via mod_backhand. The can tremendously reduce the overhead incurred by reestablishing TCP/IP connections between the servers for each and every proxied request.

Syntax for Apache mod_backhand Configuration

Directive for broadcasting on the 10.0.5.0/24 on port 4445:

```
MulticastStats 10.0.5.255:4445
```

or the same but explicitly binding to the 10.0.5.10 interface

```
MulticastStats 10.0.5.10 10.0.5.255:4445
```

Directive for multicasting to 240.220.221.20 port 4445 with a time to live of 3 hops.

```
MulticastStats 240.220.221.20:4445,3
```

or the same but explicitly binding to the 10.0.5.10 interface

```
MulticastStats 10.0.5.10 240.220.221.20:4445,3
```

Directive for accepting statistics from 10.0.5.1:

```
AcceptStats 10.0.5.1
```

Directive for accepting statistics from all IPs from 10.0.5.128 through 10.0.5.255:

```
AcceptStats 10.0.5.128/25
```

Directive for specifying the directory for unix domain sockets:

```
UnixSocketDir /opt/apache/backhand
```

Directive for activating the diagnostic handler for the /backhand/ URL:

```
<Location /backhand/>  
  SetHandler backhand-handler  
</Location>
```

Directive for randomly assigning requests to all active servers for the /data/docs directory:

```
<Directory /data/docs>  
  [ ... normal Apache configuration here ... ]  
  Backhand byAge  
  Backhand byRandom  
</Directory>
```

Directive for assigned perl scripts (.pl extension) and PHP scripts (.php extension) the application servers (called app-{1,2,3,4,5,6,7,8,9}.example.com) with the loadest system load in a random log-sized window of machines (to reduce contention):

```
<Files ~ \.(pl|php)$>  
  Backhand byAge  
  BackhandFromSO libexec/byHostname.so byHostname app  
  Backhand byRandom  
  Backhand byLogWindow  
  Backhand byLoad  
</Files>
```

Built-in Candidacy Functions and Writing Your Own

They are as follows, in the order introduced above:

- `byAge [#seconds]` (removes all servers with resource information older than #seconds from the candidacy set. 5 is used if #seconds is not specified)
- `byRandom` (reorders the entire candidacy set not effecting its cardinality)
- `byLogWindow` (removes all candidates except for the first log base 2 of cardinality of the candidacy set.)
- `byCPU` (reorders the entire candidacy set, placing those with the lowest CPU utilization first)
- `byLoad` (reorders the entire candidacy set, placing those with the lowest one minutes average system load first)
- `addPrediction` (predicts the load that redirecting a request would incur on a given server and adds that to the load seen in the shard serverstats structure. This specified last, so as to only add the predictive load to the machine being selected.)
- `byCost` (reorders the entire candidacy set placing those with the lowest cost first. Cost is based on the cost-benefit framework as described in "A Cost-Benefit Framework for Online Management of a Metacomputing System", by Amir, Awerbuch and Borgstrom.)

If `mod_backhand` changes, every server in the cluster needs to be updated and restarted. This is not effective or even feasible in large clusters. The ability to create a candidacy function outside of `mod_backhand` and dynamically load it in at run time alleviates the need to restart servers. An attempt was made to make this as straight forward as possible by introducing an API with a single function prototype. It simply requires writing a function and compiling it as a dynamically shared object. A sample `byHostname` is provided with the `mod_backhand` distribution. The function prototype is as follows:

```
int function_name(request_rec *r, int *servers, int *n, char *arg);
```

`r` is the request structure for decision making purposes. `servers` is the array of candidates and `n` is a pointer the number of viable candidates. The new cardinality of the candidacy set (`servers`) should be placed in `*n` as well as returned. During the decision making, there is `serverstat *serverstats` variable (global) that contains all of the resource utilization information about each machine in the cluster. The definition of the `serverstats` structure is as follows:

Directive for choosing the least loaded server with fresh information:

```
Backhand byAge  
Backhand byLoad
```

Directive for choosing the least loaded server in a random log sized window of servers with names that match the regex `/alpha/` and have information recent to within 2 seconds:

```
Backhand byAge 2  
BackhandFromSO libexec/byHostname.so byHostname alpha  
Backhand byRandom  
Backhand byLogWindow  
Backhand byLoad
```

```
typedef struct {  
    /* General information concerning the server and this structure */  
    char hostname[40]; /* or truncated hostname as the case may be */  
    time_t mtime; /* last modification of this stat structure */  
    struct sockaddr_in contact; /* the associated inet addr + port */  
  
    /* Actual statistics for decision making */  
    int arriba; /* How fast is THIS machine */  
    int aservers; /* Number of available servers */  
    int nservers; /* Number of running servers */  
    int load; /* load times 1000 (keep floats off network) */  
    int load_hwm; /* The supremim integral power of 2 of the load seen thus far */  
    int cpu; /* cpu idle time 1000 */  
    int ncpu; /* number of CPUs (load doesn't mean too much otherwise) */  
    int tmem; /* total memory installed (in bytes) */  
    int amem; /* available memory */  
} serverstat;
```

Advantages/Disadvantages of mod_backhand

The advantages and disadvantages of using mod_backhand are difficult to quantify without directly comparing it to another scenario. We will attempt to discuss where it sits in the field of web clustering solutions by comparing it to several prominent load balancing set ups. The easiest way to begin a comparative analysis is to describe the advantages of other products and/or methods.

The most common method of balancing sites on the Internet today is the use of multiple A entries in a DNS record. This method is commonly called DNS round robin. This provides a naïve distribution of incoming clients over a set of servers. However, other than providing a simplistic method of distributing incoming requests, it only has drawbacks.

Due to the nature of DNS, caching name servers, and the variety of DNS servers that are not RFC compliant, the time to live attribute on a DNS record often is not obeyed. Though this removes a single point of failure found in proxied set ups, this makes updating the DNS to take a downed server out of rotation extremely ineffective. Also, due to the nature of caching and the uneven utilization of name servers across the Internet make for poor load balancing. Resources within the cluster are not utilized effectively.

There exist two types of proxied set ups, one is on layer 4/5 and the other is on layer 3 of the OSI network model. Both suffer from the single point of failure problem as well as network topology restrictions. The layer 4/5 designs functions very similarly to mod_proxy combined with internal round-robin request assignment. It has the single point of failure problem that all proxy based solutions have. It does have one main advantage; as it is processing requests on layer four, it is privy to the actual URL and headers that are being transmitted and could feasibly make more intelligent decisions on where to allocate requests.

The layer 3 model is implemented in products like BIG/ip, Alteon, Arrowpoint, and ServerIron. It doesn't process requests, but rather it processes TCP/IP sessions. It redirects the TCP/IP session, most commonly using IP masquerading, to a backend server. Most of the processing is done inside the kernel or is embedded in hardware and is thus very fast. The algorithms that it uses for assigning requests to servers are limited to the information to which it is privy. This includes average connect times, average turn around times on a session and other layer 3 statistics. Some of these products also offer "layer 5 switch" which allows it to look 100 to 1000 bytes into the HTTP request and make decisions based on information there. But, as HTTP requests are serialized, it only looks into the first one. This poses problems if all of the content doesn't exist on all of the servers. The concept of IP masquerading for multiple machines and attempting to allocate TCP/IP sessions intelligently across them is an excellent one. However, the more complicated it becomes (by attempting to operate outside its layer), the more difficult it become to adapt to new services. Another disadvantage is the new capital investment on these blackbox hardware solutions. As they are single point proxies, you will need two for failover!

The proxied approach does provide the ability to easily down a production server without the worry of a service interruption. The point of failure is now in the load balancing proxy instead of the web servers.

The `mod_backhand` approach is a mixture of the two and can be combined with any of the above solutions to solve specific problems. One of the major advantages of `mod_backhand` is that no dedicated proxying hardware is required to balance your cluster. Now, if you are currently using a server to balance a cluster in a proxy configuration, you can now use its resources to actually service HTTP requests.

The main advantage over a proxied approach is that all of your hosts can be accessible from the Internet. There is no single point of failure (other than your connectivity itself) sitting between your cluster and your clients. If one or more host is accessible from the Internet, then the cluster is accessible. The common method of placing more than one host in rotation is by using DNS round robin. To compensate for the slowness of DNS updates/propagation an operation system level fault tolerance solution can be implemented. During the event of a downed server, another accessible server will make itself available from the IP address of the downed server.

Using operation system level fault tolerance with simple DNS round robin is a good mechanism for maintaining availability, but it can make a poor load balancing scheme even worse. When three servers are present and one goes down, another will assume its IP and its load.

`mod_backhand` redirects requests to the best server in the cluster, so even under poor mechanisms of balancing incoming connections (like DNS round robin) `mod_backhand` will compensate by reallocating individual requests.

We have not yet touched on the two main advantage that `mod_backhand` has over all other available solutions:

- It requires no change in the current cluster configuration. It will drop in an existing standalone web server as well as in both DNS round robin and proxy configurations.
- It is smarter! It has available a wealth of information about the request and the resources available within the cluster and it can redirect individual requests in a single HTTP session to different servers.

The combination of the those two advantages with the fact that it incurs no overhead when it does not reallocate requests, means that it works well as a full-on implementation and as a correctional facility for existing load balanced clusters.

As an additional perk, it provides an excellent tool (the DT) for monitoring resource utilization within a large cluster. And with resource utilization information being multicasted on your network, other applications that could benefit from such knowledge could be augmented to do so.

Concerns have been raised about the performance of an application level proxy with `mod_backhand`'s architecture. The proxying implementation respects HTTP/1.0 and 1.1 keep alive semantics, so it is much more efficient than Apache 1.3.x `mod_proxy`. And it has been used under relatively heavy real-world load handling roughly 15 million requests per day.

mod_log_spread as a new approach to distributed logging and monitoring

Handling access logs on a larger cluster can be a painful process. You either have to go through the effort of recombining web server logs or find an alternative method of log collection. There are several black box products that sniff network traffic and reconstruct web logs from the transactions it witnesses. However, if your revenue stream is based on the validity of logs, these will not do as they lose messages.

A robust, reliable, ordered and efficient transport for access logs is needed. That is a hefty requirement for such a seemingly simple task. After careful investigation of the requirements, we find that group communication systems provide those exact semantics (and more.) Spread (www.spread.org) is an example of such a group communications system.

`mod_log_spread` is a set of patches to the core Apache `mod_log_config` that add the ability to log access logs to Spread a group as an alternative to files and pipes. The Spread system provides a flexible messaging facility, with open group semantics, to process groups that can guarantee reliability and ordering. It uses cutting-edge network protocols that are both robust and efficient. This allows multiple clients to publish information to groups and multiple clients to receive messages from groups. The protocols within Spread implement efficient reliable multicasting of information. When compared (grossly

simplified) to TCP/IP:

Bandwidth B available to data

	1 publisher -> 1 subscriber	1 publisher -> m subscribers	n publishers -> 1 subscriber	n publishers -> m subscribers
TCP/IP	B	B/m	B/n	B/(nm)
Spread	B	B	B/n	B/n

As the entire purpose of "distributed" logging is to handle a cluster of machines, we see that the first two columns do not apply. The third column requires a single subscriber to the log stream, which prohibits any additional, separate analysis tools from attaching to the stream. Rather, they must all feed of the incremental additions to the log file being generated by the single subscriber.

So, under normal circumstances, the system will be operating as described by column four. This clearly shows that a distributed logging system must be built atop a message bus that uses efficient, reliable multicast protocols.

The other outstanding benefit of using a group communication system over an $O(n^2)$ TCP/IP connection mesh, other than simplicity and efficiency, is the ability to rely on the ordering of messages. Spread can ensure that all messages that enter the system will be delivered to each subscriber in the same ordering. In a cluster set up, load balancing may deliver one request to one server and the next to another server. In order, to preserve cause and effect, ordering of messages is a necessity.

Though `mod_backhand`'s DT provides a nice interface for monitoring a cluster's resources in real time, when things go wrong this doesn't give you fine grained information you need. You would like to know how many requests per second each server is satisfying, the number Mbits/sec each server is serving, or perhaps the rate of serving various response codes from each server. All of this is possible by analyzing the access logs in real time. With the simplistic C, Java, and Perl API's provided with Spread, it is trivial to write custom monitoring tools that can be used in conjunction with `mod_log_spread`.

`mod_log_spread` has been tested rigorously under heavy load with an Apache cluster serving approximately 40 million hits (log lines) per day.

Syntax for Apache `mod_log_spread` Configuration

We will assume that our spread daemon runs on the default port of 4803.

Directive to set the global spread daemon to spread running locally (via UNIX domain socket):

```
SpreadDaemon 4803
```

Directive to set the global spread daemon to a remote spread instance (via TCP):

```
SpreadDaemon 4802@remotemachine
```

Directive to log to a spread group named "site4" in common log format instead of a file or pipe:

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common  

Custom_Log $site4 common
```

Future Directions

Several technical design and implementation issues that must be tackled before any major augmentations are done. The first is to design more elegant support for SSL enabled connections. The second is to eliminate the need for an entire Apache child process to be blocked while proxying a request. One or more RMA type processes should be created that normal child processes can hand requests to for proxying. Each of these dedicated proxying processes can handle a multiplicity of active requests using `select()/poll()` semantics and asynchronous I/O. A mechanism for handing a client back to a normal child process must be investigated.

There are a few places we would like to see the `mod_backhand` project go in the bigger picture. The first is back into the research stage from whence it came. Analysis on the effectiveness of the decision

making and reallocation of requests is necessary for any major improvement.

Our vision for `mod_log_spread` is mass adoption and a comprehensive suite of community developed open-sourced monitoring tools.

Acknowledgements

The Backhand project was started at the Center for Networks and Distributed Systems at The Johns Hopkins University. This project would not have been possible without the excellent collaborative environment for research and development found at CNDS. Many thanks to my advisor, Yair Amir (yairamir@cnds.jhu.edu), for constantly nudging me in the right direction. Also, many of the design and implementation details inside `mod_backhand`'s architecture are the product of collaborative work with Alec Peterson (ahp@hilander.com) and Jonathan Stanton (jonathan@cnds.jhu.edu). The implementation and integration of the cost-benefit framework were done by Ryan Borgstrom (rsean@cs.jhu.edu); the specific tunings and adaptations to web server were a collaborative effort from Amir, Borgstrom, and myself.

`mod_log_spread` was written and is maintained by George Schlossnagle (george@omniti.com). It is an externally developed and maintained component of the Backhand project. `mod_log_spread` relies on the Spread (www.spread.org) group communication system and would not be what it is without it. Thank you to the creators and developers of Spread.

Many thanks go out to all of those who have written in with bugs, congratulations and success stories.