Theo Schlossnagle
jesus@cnds.jhu.edu
http://www.cnds.jhu.edu/~jesus/

mod_backhand
use your resources

A load balancing module for the Apache web server.

Introduction

mod_backhand is a dynamically loadable module for the Apache web server developed at The Center for Networks and Distributed Systems (CNDS) at The Johns Hopkins University. This module aims to seamlessly allow for load balancing within a cluster of machines. This paper will discuss the concepts, implementations, advantages/disadvantages, practical applications and future directions.

mod_backhand is a part of the Backhand project initiated at CNDS. The purpose of the Backhand project is to develop tools for effective resource management and utilization. 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.

The Backhand project was designed to tackle all aspects of resource allocation and management. This was done for two reasons. First, the issues that must be tackled are far out of the scope of an Apache module. Second, it allows for intelligent separation of concepts and functionality. mod_backhand, in its current form, attempts to solve resource allocation and management issues within a cluster of machines on a relatively low latency, high−throughput network.

The problems involved in finding the best web server or cluster for a specific client are better tackled at a lower level. Specific approaches including optimizing algorithms based on DNS and IP routing are not in the scope of mod_backhand, but fit well in the more general Backhand project.

Problem

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

The first stepping stone of any load balancing system is its architecture. There are several to choose from, but mot contain common components. These components are external to web serving itself as they lie on the networking level.

How is a client (web browser) delivered to the correct web server in the first place? The answer, of course, is DNS. It has a record that looks something like "www 3600 IN A 10.0.1.5", meaning that host www has the IP address 10.0.1.5, but this information can only be cached for 3600 seconds (1 hour.) After this time period, clients and caching servers alike are to request this information from an authoritative source in case it has changed.

DNS is a fairly flexible architecture. It allows for multiple hosts to point at the same IP address and it allows for a single host to point at multiple IP addresses. If the DNS administrator provides multiple "A" records for a host like www (like "3600 IN A 10.0.1.5" and "3600 IN A 10.0.1.6"), then all IP addresses will be responded to any DNS queries. Due to the implementation of DNS, this will cause some clients to resolve www to the IP 10.0.15 and all others 10.0.1.6. This scheme of assigning multiple IP addresses to a host name to naively distribute requests amongst a set of servers is known as DNS round robin.

Will each server recieve an even number of incoming requests? Over some large time period, perhaps. Consider a two machine web cluster, where one machine receives requests during the the first 30 minutes of every hour and the second during the second 30 minutes of every hour. The average resource utilization on the machines is equal when looking at an average day and likewise when looking at an average hour. Yet, the servers are unbalanced.

Another approach to distributing incoming requests across multiple physical machines is known as proxying. The proxing can either be done on layer 4 (as mod_proxy does) or on layer 3 (as BIG/ip and other like products do). Proxying involves putting a machine between the clients and the actual servers. The clients will attempt to contact the server via the proxy and the incoming connections will be dynamically assigned to the "best" server. On layer 3, the dynamic assignments must be for the entire TCP/IP session, and the allocation decisison is not based of the actual request (URL or HTTP headers). On layer 4, the procy can now reallocate requests on a per request basis and is able to incorporate the HTTP headers and URL requested into the decision. On layer 4, the request must be processed at the proxy and then again at the actual server increasing overhead dramatically.

Proxying does not suffer the same issues as the DNS round robin approach with respect to poor distribution of requests. However, it does introduce a single point of failure. If the proxy fails, there will be many inaccessible servers.

Neither DNS nor proxying will solve the problem of allocation of requests to a machine that is too busy to handle them. They do not use information about resources within the cluster to make their decisions. In contrast, mod_backhand is able to gather the necessary information and utilize it in an intelligent way when distributing the requests.

Solution

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

Both DNS round robin and the proxying architecture have very specifc drawbacks. However, their drawbacks are such that they can be avoided is combined correctly. If each machine in the cluster had the ability to proxy requests to any other machine in the cluster, then the load can be distributed naively across those machines with DNS round robin and the single point of failure disappears. Both DNS and proxying do not provide a good framework for intelligent decision making. As they both operate on TCP sessions, the session will be assigned to a server without knowing about the future requests that may be asked across it.

This approach sidesteps the two major drawbacks in DNS round robin and proxying, but it is not a perfect solution. In fact, due to the inequity of resource requirements amongst requests, this solution is merely the architectural framework for a correct solution. We must decide where the requests are to be reallocated more intelligently. We need to use information about the request, the headers and the available resources in the clutser.

The serious issue that mod_backhand attempts to tackle is a resource allocation problem. These problems have been worked by too many researchers to name names. The fact that remains is that no web load balancing infrastructure uses resource utilization information to make decisions.

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 resource 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 a open area of research, and we are still investigating various algorithms. We realized that mod_backhand would be the first web clutser 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 on online resource allocation of HTTP requests.

As discussed before, average utilization can be deceiving if the time period for average is too long. How long is too long? It is directly related to the length of time required to satisfy a request. If requests commonly take a day to satisfy, measuring the resource utilization every 12 hours should be sufficient. The larger the averaging period for this information is, the larger the error will be.

Is load what we want to be balancing? Let's, for argument's sake, assume a web request takes 30ms. This would mean that our resource utilization information (load in this case) would have to be a 15ms snapshot or 30 ms rolling average. This is shorter than the time slice for process scheduling on most machines. System load on UNIX machines currently has a 1 minute rolling average. We see the problem immediately. We need resource utilization information available on the same time scale as our request times. Resource allocation techniques for scheduling jobs on clustered servers has 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 migratable 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.

Implementation

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

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

Each of the components interacts closely with the others to provide a framework for informed resource allocation decisions. They will each be discussed in turn with an example configuration directive if applicable.

The resource information manager (RIM)

The resource information manager is responsible for acquiring resource utilization information from the local web server, multicasting that information to the cluster and collecting resource utilization information from other machines in the cluster.

The RIM must collect local resource utilization in a manner that is architecture and platform dependant, so mod_backhand has a specific section of code that is very system dependant; therefore, porting to new architectures is not always trivial. All resource information is held in an attached shared memory segment created during Apache's initialization phases. All one time (unchanging) information is calculated at this time and stored in the shared memory segment. Current one time information includes the hostname, service, number of CPUs, total physical memory and the speed of the machine (calculated by forked, tight floating point loops). The speed of the machine is called the arriba and is stored in a file in order to speed restarts. The web server is blocked completely during start up while the arriba is calculated.

The RIM uses either IP broadcast or IP multicast to share resource utilization information. Multiple multicast and/or broadcast addresses can be used for information multicasting. The default multicast interval is one second and can be changed at compile time. The RIM is also in charge of trivial access control during the acceptance of information. This is done with simple accept clauses which are checked against the sender IP of each packet.

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 beginning with 10.0:

Theo Schlossnagle
jesus@cnds.jhu.edu
http://www.cnds.jhu.edu/~jesus/

```
AcceptStats 10.0.0.0/16
```

The resource maintenance assistant (RMA)

The resource maintenance assistant provides mostly cleanup and pre−allocation services for expensive operations that should not be done on demand. Its purpose is similar to the apache pre−forking ability. The RMA maintains a pool of existing connections to other servers within the cluster and manages their wellbeing.

The time required to allocate a system socket and connect to another web server can be quite substantial and requires a multitude of resources (systems calls and TCP/IP negotiation on both the local machine and the remote machine). This is done in an asynchronous manner, to avoid unnecessary blocking. The RMA maintains a pool of already established connections, building them and tearing them down as necessary. An individual Apache process, after deciding to reallocate an incoming request to machine X, will ask the RMA for an already established connection to server X.

The RMA and the Apache processes (children) communicate via UNIX domain sockets (AF_UNIX). These files are created in a directory specified at run time by the UnixSocketDir directive. The RMA and children communicate over these sockets to request connections to other servers within the clutser and to hand open file descriptors (attached to open BSD sockets with established TCP/IP sessions) back and forth. The connections that the RMA manages to other servers and kept alive as long as possible (via header rewriting discussed later).

The decision maker (DM)

The decision maker is simply a flexible (configurable) set function: dm(SS, R) −> server; where SS is the resource utilization information for the cluster and R is specific information about the request being made. In order to provide maximum flexibility, mod_backhand can dynamically load decision making functions at run time.

The decision making framework is designed around the concept of candidacy functions. Multiple functions can be provided and the DM will execute each in order to arrive at a decision. The function can reorder, narrow, or even expand the set of candidates for reallocation. The function is called with an array of integers, a parameter and a request line. The array has a size equal to the maximum number (n) of servers in a cluster (compile time value) and contains an ordered set of candidates. The process starts with an array enumerated from 0 to n. Each candidacy function can then reorder, remove or add candidates to the array. This array is called the candidacy set. At the end of the candidacy function processing, we choose the first element in the candidacy set.

As little research has been done in the area of per−request resource allocation algorithms, we required simple building blocks that allow the easy construction of more complicated decision making algorithms. The first and most obvious building block is one that eliminates servers that are no longer participating in the cluster (downed or crashed). Another convenient building block performs a randomization of a set of servers. Two more blocks reorder a set of servers based on system load and CPU utilization. These blocks allow algorithms to equalize both load and CPU utilization.

As has been discussed, using system load has sampling problems. Requests will be continually assigned to the same machine until the load slowly rises above the other servers in the cluster. Unfortunately, system load slowly follows resource utilization (because it is a rolling minute average) and, on some systems, it can take up to five seconds to change at all. This means that an algorithm balancing strictly based on load, we will overload each server in a round robin fashion.

A simple way to compensate for this problem is to have each server consider only a window of the available servers. This window was proposed to contain only a logarithm of the number actually available. Thus, we have a building block that will remove all but a base 2 logarithm of the available server set. We don't wish to have a server consistently look at the same set of servers, so by combining this with the randomizer block, we can select a random window of servers from a given set.

The last building block we want is based of a more theoretically based allocation approach. This approached is based of the supply and demand principles of economics. It assigns an exponentially increasing cost to each resource on a machine. This block will reorder a set of servers based on the calculated cost of allocating the necessary resources to satisfy a request. Determination of those resources, their costs and adapting this approach to short−term jobs is still in the research stage.

These blocks are implemented as built−in candidacy functions and are provided to allow the creation of customized load−balancing algorithms. 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)
- 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 or 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:

```
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;
```

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
```

The diagnostic tool (DT)

The diagnostic tool provides a mechanism for convenient cluster resource utilization monitoring. Its basic purpose is to present the information available to you (the administrator) the same information that is available to the DM. As the diagnostic tool is a simple apache content handler, it can be used in the fashion as the server−status content handler provided by mod_status. It looks in the shared memory segment for the resource utilization information and outputs an easy−to−read table with the data. One must be careful not to enable load balancing on that content handler (Location) or you may get another machine's backhand−handler. It also provides links to the Backhand project and CNDS to make looking for related resources more convenient.

The internal proxy

The internal proxy handles the dirty work of redirecting requests. It performs in a very similar fashion to mod_proxy. If mod_backhand finds that another server is a better candidate than itself to serve the request, the proxy will contact the RMA and request and open channel. It will then make a request for the same URI over this channel feeding the response back to the client. It does however make optimization in stride.

One of the more expensive operations in HTTP is the constant building and tearing down of TCP/IP sessions. This requires quite a bit of effort from both parties involved. The RMA, as described above, maintains established connections to the server; however, that does little good if the HTTP protocol terminates and the session completed. Many clients make requests in a closed connection manner. As a web server you have to respect this as it is legal HTTP. However, as we are guaranteed that mod_backhand will be the only process talking over the RMA established sessions, certain optimizations can be made.

When the client requests a specific URL from server X and we find that we are going to proxy it to server Y, we can augment the headers to maintain different semantics on the two different sessions (client::X and X::Y). The client can either request a document from X in a manner that will maintain the sessions after the response (called keepalive) or in a manner that will terminate the connection after that response (called close). These session semantics are defined by the HTTP headers that are transmitted to and from the server. X will maintain a legal HTTP connection to the client, however when it proxies the request to Y it will modify the headers to as to force an HTTP session with keepalive semantics.

Other header modifications are made to provide hints to server Y that the request has already been proxied once. The header is the HTTP_BACKHAND_PROXIED header and is processed in the post_read_request phase of the request processing. This header is ignored if the requests remote IP is not in the AcceptStats access control list. This effectively eliminates outsiders from defeating mod_backhand.

A particular issue to be dealt with is IP based authentication that is commonly used by webmasters to control viewing portions of their sites. mod_backhand will respect this initially, but the proxied request will be from X not the client. This is not the desired behavior, so the HTTP_BACKHAND_PROXIED header serves another purpose. While the fact that it exists denotes that the request is a proxied one, the value of this header holds the "real" clients IP address. During the post_read_request phase, mod_backhand will switch remote_ip addresses in the headers, environment variables and internal Apache structures required for transparent functionality.

Advantages/Disadvantages

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 provide 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 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 design functions very similarly to mod_proxy combined with internal DNS round robin. 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 from F5 labs, Inc. 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 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. It is feasible that a machine like this could acquire and manage resources in the same fashion as mod_backhand and use that information in the decision making process. As it operates on layer 3, it is very difficult, though not impossible, to extract the layer 4 information necessary to make decisions based of the request itself. 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.

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 combine 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 you 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 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 both DNS round robin and proxy configurations.
- It is smarter! It is 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 and 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 augment to do so.

Practical Applications

This section will discuss two specific implementations of web clustered solutions for different web applications. The first scenario we will discuss is a simple web clustering solution with a few PC servers and the second will be a much more elaborate, high volume database driven portal.

Scenario 1 (The small cluster)

Let as assume four machines exist in the current set up, one is sometimes for bulk emailing and another is used for processing logs. The remaining two are dedicated web servers, the first is used as a production machine and the second is a fail−over replica of the first. In the case of a downed web server, the replica will assume the production server's identity.

The outgoing email server (or relay) sends email from approximately 6pm to 6am every day. During this time, the load on that machine increases dramatically and it could not feasibly serve as a responsive web server. The log−processing machine is used for processing email and web server logs. It does this automatically every 4 hours, but also performs many manually invoked log analysis queries. This machine could be feasible used as a web server in the current scheme; however, it would not service requests as quickly if it is always given an even distribution. Putting the logs server in a DNS rotation scheme could be very bad when heavy log analysis is happening. The backup machine could easily be put in to the rotation, and help the current set up. Steps must be taken to assure network level fault tolerance, but any simple machine IP masquerading will work well.

Let us now look at the implementation using mod_backhand as a tool for efficiently allocating resources. We wish to immediately bring all available resources into the cluster, as they are not available to serve requests if they are not given the chance. We need to set up network−level fault tolerance. mod_backhand doesn't handle this directly; it is instead handled by a high availability tool (like Linux−HA). Next, we want to add all of our machines as A entries to the DNS record for the host names in question.

Install the mod_backhand module into the existing Apache set up and you are ready to go. Now, the only thing remaining is to set up the load−balancing schema. For simplicity, we will just load balance the entire site sans the DT. The rules that we wish to use are simple. We want to go to a machine, so long as it is participating (fresh resource information), and it's load is the lowest. The appropriate directives are to balance byAge and the byLoad.

Now there is a fault−tolerant load balanced cluster servicing HTTP requests. Downing a production server for maintenance and upgrades is now feasible without causing a service outage. And any given machine can we used for any given purpose (log analysis and email relaying) as mod_backhand will compensate and allocate incoming requests to the server that is least loaded.

Scenario 2 (The database driven portal)

This set up is an enterprise solution to serve serious content (bandwidth) and a high hit rate.

Let us assume that content is kept in a database and is processed 97% of the time by read−only queries. The other 3% of the requests made to system perform database updates and other read−write operations. The database server is currently a larger server in comparison to the production web servers. There are 4 machines running as web servers and 2 running as dedicated email servers (outgoing content). Currently all database transactions are asked directly of the database server. The email servers send outgoing mail on during the night for approximately 12 hours and the web servers maintain a system load of 3, while the database server maintains a load of 80. (No, this is not ideal).

First, we wish to bring all resources to the front lines. The email servers should be added to the scenario, adding 12 hours, each, of available resources to the set up. We do this in a similar fashion to scenario 1. Next, we wish to move the load from the database server to our web cluster resources. We install the database server software on each web server. We replicate the existing database on to the web servers making them read−only. The method of database replication is an orthogonal problem of information consistency; though in this case continually applying update/redo logs should be sufficient. The web servers can now make read−only transactions to a locally managed database. Finally, we drop unnecessary indexes on the database server as it isn't performing 97% of the queries it was before.

The original database server is now far under−utilized and we would like to use its resources in the cluster. We start by insalling Apache+mod_backhand on that machine as well. This new web server will be a part of the cluster (though not in the DNS round robin set up) that handles database transactions that perform read−write operations. Each machine can now interface with its own local database; in most client/server databases, this vastly increases performance.

All components of the web site are analyzed for transaction type and classified as either containing or not containing a read−write database transaction. All incoming requests will be naively distributed across the 6 front–end web servers. Any request that requires a read–write transaction will be reallocated to the database server for service. All other requests will be reallocated to the machine on which it will cost the least (in resources) to satisfy. Of course, analysis of certain database queries (like full table scans) may show that overall performance will increase if those transactions were always made to the database server.

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. The other place is the wide area scenarios. Though much of the wide area load balancing problem is better tackle under a more general project like Backhand, mod_backhand could specifically benefit from the use of HTTP redirections for wide area balancing.

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 with out 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.

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