

CMSC 621 Term Project : Brandon

Don Miner
Brandon Wilson

Abstract

Web services are web-based applications that use an XML-formatted communication protocol, such as SOAP, to communicate and service requests made by clients. The service providers face several weaknesses, including a limitation on the number of clients they can serve at once, and making their service available to a large set of clients. The goal of our system is to provide a distributed foundation of web service providers so as to provide fault tolerance, load balancing of client requests through request distribution and dynamic replication, efficient provider and service discovery, and high service availability.

We utilize various techniques to achieve all of the mentioned attributes of distributed web service architecture. An eventually consistent cache is utilized at every server to maintain a cache of other providers in the network and perform node and service discovery through random synchronization periods. Also, in any distributed system, messages are passed among nodes for communication purposes, which can become a very bandwidth expensive task. We use a multicast method that minimizes redundant messages while messaging a list of recipients. Also, to ensure confidential communication, all servers have an RSA key pair that is used to secure the communication channel during incoming connections using SSL.

1. Introduction

As the internet becomes an increasingly large set of interconnected systems, the idea of tying a subset of these systems together into a distributed system arises. A distributed system has many advantages to an ordinary uni-processor system by having the resources of all nodes in the system at their disposal to service the tasks that it is assigned. Also, since a distributed system consists of many smaller systems, the failure of the entire distributed system is less likely when a single node goes down. On the other hand, a distributed system that shares a workload introduces problems of its own. For example, load balancing must be performed so that one node does not handle a substantially larger number of requests than the other nodes in the system. Also, when a new node is introduced to the system, how are it and its resources discovered and integrated into the overall system without incurring a large overhead in message passing?

Web services are web-based applications that use an XML-formatted communication protocol, such as SOAP, to communicate and service requests made by clients. The clients and servers are typically applications

and web services provide them with a way to function together in a platform independent manner. Web services are a perfect application for a distributed architecture because as their popularity increases, the server hosting this service gets flooded with requests and either the system crashes or the server response time increases dramatically, making many client application users unhappy. With a distributed infrastructure instead of a single uni-processor server, the service provider could potentially tie many machines together and then have a unified set of resources with which to service requests. For example, if 10,000 requests are sent to uni-processor service provider, then the server will get heavily loaded and most likely crash due to an overload on the network card. A service provider with multiple machines that is running our system would be able to receive the 10,000 requests and as they arrive balance the load across all of the systems by replicating the more popular services and redirecting the incoming requests to lower loaded systems.

Our system also supports WS-Policy at all levels of our architecture. Policies are verified for intersections between clients and servers as well as between servers and serv-

ers. The use of WS-Policy has greatly affected our design mainly due to the fact that new restrictions are in place and a client may not be able to request a web service from a particular node in our system if it does not have an intersection. A background and introduction on WS-Policy is contained within section 3.3 of this document.

We propose that our designed system named BranDon, to be referred as BD later on in this document, achieves the necessary load balancing, service availability, node discovery, and security that is necessary in any distributed system while still appearing as a single provider of web services to the user of the final product. BD fulfills these needs by providing a middleware solution which takes a client's request and distributing the task as needed. BD handles all administrative tasks in the system while being as transparent to the user as possible. Also, BD strives to depend as little as possible on specific configurations so that no modifications to existing systems have to be made.

In this document we will discuss every aspect of this project and its proof-of-concept implementation. Topics covered include system architecture, system design, load distribution, WS-Policy, security measures, node discovery, system state caches, synchronization and others. The overall design, as well as details of an actual implementation is to be covered in this paper.

2. System Architecture

The system architecture is a client-server model, where the server application is running at each and every node in the system. Also, each node has a web service registry, such as Apache Axis, and an HTTP server. Any application that wishes to utilize our middleware must use the Client Module (see Section 5) locally to interface with the rest of the system. If one does not wish to use our client, as long as it adheres to the standard protocol used by the client module, it will function. The nodes in the system communicate with each other in order to keep an eventually consistent conception of system state. As an example, if a client wishes to make a request for a service, it

must use the client module to obtain the WSDL of the desired service. The client application can then proceed to make traditional SOAP requests to the service registry that is specified in the WSDL.

3. System Design

We had several general design goals in mind while deciding which route to take in creating BD. These goals coincide with the standard goals of distributed systems, however we lent most of our effort to making a system that is very scalable and services are as available to clients as much as possible. These classic goals in distributed systems we accomplish are described in detail in the rest of this section.

As well as striving for these goals, we used a couple rules of thumb that make our design more robust. One such goal is that no individual BD node communicates with other nodes while referring to itself in the third person. We feel that this is very important, as a single machine may have several IP addresses or hostnames or may differ from one node to another.

Another goal in designing the system was to use an implementation which required the least amount of modification possible. Possible implementations that would have increased the transparency of the middleware, such as hacking the Python SOAP code or hacking the internals of Tomcat were thrown out in favor of a complete standalone application which only requires a directory inside a web service registry, such as Axis. This makes the system easier to use in different configurations as well as makes it easier for one to update their software not involved in the actual BD code. For example, if one wanted to switch SOAP libraries, this should not be an issue as long as the SOAP library can take in an http path to a WSDL file.

3.1. Multicast Message Routing

All messages among nodes in the system are sent using a multicast over a path created by solving a problem much like the Steiner Tree problem [2]. Our algorithm finds a sub-

set of edges in the network topology and produces a path that includes all of the recipients in the recipient list. This approximation produces a solution to the Steiner Tree-like problem, although it may not be optimal. The use of the multicast greatly reduces the network load on the message source in comparison to the alternatives, such as just resending messages.

Our algorithm plans the path of the multicast by trying to pick the edges that will require the least amount of hops possible. Jumps from one node to another may be necessary in the case of network disconnectedness or two nodes not being able to communicate because of WS-Policy conflicts. The algorithm works by flowing outwards from each node that needs a message and when two paths meet, two sets of nodes are now joined. This process continues until the message can be sent to every recipient that needs to receive it. A more formal description of this algorithm follows.

Our algorithm performs a simultaneous Breadth First Search (BFS) from each of the message recipients in the recipient list. Each BFS moves one level at a time and no node can move to the next level until all nodes have completed searching at that level. All of the initial nodes (message recipients) in the search are labeled with a path that contains only itself. These recipients are also added to a global set list, GS , as individual disjoint sets. With every iteration of the search, each node checks the nodes one edge away and if any of these nodes have not been labeled yet, it adds the new node to the proper set in the GS and labels this node with the resulting set. If a node is encountered that has already been labeled, the unification of the paths is performed. Then, the two sets these nodes belong to in GS are combined. On the next iteration of the BFS, the search continues from these newly found nodes and repeats the process until only one set remains and this is the set that consists of the nodes that are needed in the multicast.

3.2. Load Distribution

The load distribution in BD is achieved through a combination of forwarding re-

quests and replicating service to other nodes. The goal of the load distribution is to keep the load as evenly distributed among the system nodes as possible. Every server maintains a receiver threshold, T_R , and as long as system load is below this threshold, the server is willing to accept forward requests for the services it currently hosts, or replicate a service in order to handle an outstanding request that cannot be handled by anyone else.

3.2.1. Request Forwarding

In order to query a service to perform a SOAP/WSDL job request, a client must first know where the WSDL file for the service is located. Our middleware handles this request and provides the client with the information it needs. To get this information, the client queries a known server S_0 asking for a service location and provides the policy that the client supports. S_0 then looks through its cache and compiles a list of servers that it believes to host the specified service and compiles a list of these servers that have a policy that is compatible with the client policy. If the list containing the nodes that have the service is empty. This is because the server must tell the client the search failed because no such service is known. If this list is non-empty, S_0 then sends a multicast message to the all of the possible service providers asking them if any of them is willing to execute the request. If any of the servers respond positively, meaning they are under the receiver threshold, then S_0 returns the WSDL information for this service to the client. Finally, if the server does not find a provider willing to execute the service, but there are providers that have the service and refuse to service it because they are outside of the receiver threshold, then we resort to replication, described in Section 3.2.2. If nobody is willing replicate, then as a last resort S_0 randomly chooses a node that has the service and forces it to service the request.

3.2.2. Adapting Via Replication

Replication among nodes is necessary whenever the current set of nodes that are capable of servicing a request are overloaded. Repli-

cation can be performed when there are nodes without the service that are below the receiver threshold and thus willing to replicate a service. In the case that replication must be performed, as described in Section 3.2.1., a list of servers is compiled where each element in the list intersects the client policy with their own and does not already have the service. S_0 sends a multicast to the eligible servers and asks them to replicate the service. Upon receiving this multicast, if the receiving server is below the receiver threshold, it will send out a multicast to all of the nodes in its cache that have this service to package it up and send it back. If no one in the cache has the service then the replication at this node fails, and the search for someone to replicate continues. Finally, if no one can replicate, S_0 is forced to give the request to an already overloaded node with the service as mentioned in Section 3.2.1.

Each node has a pre-defined maximum number of replicated services that it can hold. These replicated services are called *dynamic* and may be deleted if the node is full and needs to make room for a new service. Services manually placed into the web service registry directories are often placed as *static*, which are guaranteed not to be deleted, at any time. This was put in place because a single node may become overloaded very quickly if it happens to have all the services the system needs at the time. By cycling out the less heavy services this possibility is reduced.

3.3. WS-Policy

Our system uses WS-Policy in order to give more power to configure which node is able to talk to which node or client and what guarantees are needed in order to establish communication.

WS-Policy allows a service provider to specify their policy for providing services to a client in an easily managed XML format. WS-policy allows two types of statements, *ExactlyOne* and *All*. *ExactlyOne* indicates that one and only one assertion from within the *ExactlyOne* tags should be chosen. *ExactlyOne* provides a series of alternatives, similar to an XOR. On the other hand, *All*

statements are used to require a series of assertions to be present, similar to a *and*. A policy is said to match another given policy, if the policies have an intersection. A policy intersection is a new policy can satisfy both original policies.

Example WS-Policy File:

```
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp>All>
      <sp:Faculty />
      <sp:PhD>
    </wsp>All/>
  </wsp:ExactlyOne/>
```

The sample policy requires that a client supply credentials proving to be a faculty member with a PhD in order to request a service.

Every server and client in our system has a policy file and upon requesting a service, the client presents its policy and the server finds the intersection. If the intersection exists, the server notifies the client of the policy intersection with which it must conform, otherwise the server refuses to comply with the request and tells the client there was a policy mismatch. Inter-server communication is also managed using policies. Servers will not replicate or synchronize with any server that has a mismatching policy.

The most common way to use WS-Policy is to just have it ensure that no client invokes a service without being able to intersect with the server's WS-Policy. The WS-Policy is checked first thing and the message may be rejected if no intersection exists.

Also, our system allows for WS-Policy checks to be honored between system nodes. This could be useful for several things, such as when some node does not wish to talk over unencrypted channels, for example. The issue that arises with this addition is nodes which are no longer one hop away

from each other. Nodes can now be several hops away and our software is programmed to handle this. This problem is easily solved with our multi-casting algorithm described in section 3.2.1.

3.4. Secure Communications

All communication among the servers is secured using SSL. All servers have a signed X.509 certificate containing a public key and a corresponding private key stored on disk. Every time a connection is made to a server, the connection is secured with SSL. With this, all information that crosses the channel is confidential and the client that is connecting to the server can verify that the information it is receiving is unaltered and was sent by the server.

3.5. BDCerts

BDCerts are certificates that are associated with each and every service that is introduced into the system. The BDCert establishes a type for the service that can be either *Static* or *Dynamic*. A service that is specified as *Static* will never be deleted from the node that has a BDCert specifying it as static. *Dynamic* on the other hand indicates that the service can be deleted if it is not being very productive and not receiving many hits. The BDCert also specifies a user-defined weight for the service. The weight is a judgment of how much system resources are required to service a single request. When calculating the load of a node, the weight of a service is taken into account as well as the number of hits the service has received. Hits are also worth less as time goes on to more accurately depict recent loads.

4. Server Design

The BD server is very modular in design, having a separate module to handle different aspects of the distributed environment. When the server is first launched, it branches into two different processes. One process, deemed the Discoverer, is in charge of service and node discovery, while the other thread listens on a pre-designated port

for incoming connections from servers and clients.

4.1. Node Discovery

The Discoverer is responsible for maintaining a cache of all other nodes in the system and keeping it as up to date as possible with the information related to that server.

4.1.1. The Server Cache

The server cache contains information about all of the nodes that the Discoverer has come in contact with to this point. For each server the cache maintains the cache WS-Policy file for the server, the IP address, port, last time of synchronization, and the services that are located at the site.

4.1.2. The Synchronization Queue

The Discoverer maintains a queue of nodes that it knows exist and may or may not have previously synchronized with. The head of this queue is the node the discoverer synchronized with the longest time ago. Whenever the Discoverer synchronizes with a site in the queue it places the site at the end of the queue. When selecting a node from the queue to synchronize with, it randomly selects a node from the queue, with the nodes closer to the front of the queue having a higher possibility to be dequeued. This way the Discoverer is to synchronize more often with nodes which it has not heard from in a long time while introducing a little bit of a random element to mix up the order.

4.1.2. Pair-wise Synchronization

The cache is updated and maintained using the idea that some inconsistency among caches at different servers is allowable as long as they eventually obtain complete synchronization. In order to maintain *strict* consistency, there would be a huge overhead in the message passing sub-system. The eventually consistent cache is an idea that originated from the Bayou File System [1], which uses this concept. The nodes in our system hold pair wise synchronization ses-

sions at a time interval that is directly proportional to their system load. This limits the amount of time spent synchronizing while operating at high load. When the time comes to synchronize, the Discoverer chooses a node from the synchronization queue by the method specified in Section 4.1.2. and initiates the synchronization process by sending the specified node its cache.

When the node receives this synchronization request, it replies with its cache. Then it will look through its entire cache and for each node that it has in common, it will save the element of the cache that has the most recent timestamp. Logical synchronized clocks are not needed here because the timestamp for a given node will only be given by that node. For each of the sites that are not common, the node will ping the other node and if it receives a response, it will either leave the node in its cache if it was there already, or add it if it was not. If the ping does not come back, the node will ensure that the node is no longer in the cache.

Using this method of discovery, nodes can be added the system easily by alerting a single server of the entry and the discovery will be propagated to other systems via the synchronization periods. The same theory of knowledge propagation applies to newly discovered services.

4.2. The Listener

As previously mentioned, the listener is a process that binds and listens on a port for incoming connections and upon receiving a connection, it launches a new process that will handle the incoming message, as specified in Section 4.3.. The thread then goes back to waiting for more incoming connections.

4.3. Protocol

The BD middleware node servers use a very specific protocol in order to communicate. All messages are marshaled Python lists that are compressed with Python's zlib module. All messages are of the form send-receive so that a minimal amount of state and open connections have to be kept. All messages

are two-phase: the first phase is a WS-Policy check and the second phase is the actual request.

We describe the usual functionality of the protocol methods. We omit the error checking details and error codes returned by these.

4.3.1. Will You Execute

This message is utilized during request forwarding and it asks a node if it will execute a given request.

Inputs:

- i) Service Name

Outputs:

If not overloaded:

- i) Node to host's Port Number
- ii) Node to host's Web Services Directory

If overloaded:

- i) NO

4.3.2. Give Your Services

This message assists in avoiding the cost of a full synchronize when a node tries to send a request to a server that does not have a service, but it is noted in the cache as being a provider. This simply updates this nodes entry in the cache.

Inputs:

- i) NONE

Outputs:

- i) The node's services and a timestamp

4.3.3. Synchronize

Issued by the Discoverer when it needs to synchronize caches with another node

Inputs:

- i) The initiating party's cache

Outputs:

- i) The receiving party's cache

4.3.4. Where is?

This message is used in the client module to query a node for a service and obtain the WSDL to know how to use it.

Inputs:

- i) Service Name
- ii) Client Policy

Outputs:

- i) The location of the WSDL for the service

4.3.5. Give Service

Packages up the specified service and sends it across the network

Inputs:

- i) Service Name

Output:

- i) The marshaled and zipped directory.

4.3.6. You Will Execute

This message is used during request forwarding to tell a server that it must execute a request since no other nodes could be found to service the request.

Inputs:

- i) Service Name

Outputs:

- i) Port Number
- ii) Web Services Directory

4.3.7. Hint

This message notifies each of the recipients in the input list in a circular pattern of their neighbor to the right. This results in a synchronization between the two nodes.

Inputs:

- i) A single node to be hinted into the system

4.3.8. Ping

Pings the specified site to see if it is reachable. This is used to settle conflicts during synchronization to decide if a node is available or not.

Inputs:

- i) The IP address
- ii) The Port

Outputs:

- i) Any response if the system is reachable

4.3.9. Status

The status message is use by the audit tool to query nodes for their current status. It returns information on its services, its connections with other nodes and its WS-Policy, among other things needed to perform useful statistics.

4.3.10. Will You Replicate

This message is used to query nodes and have them replicate a given service if they are under the receiver threshold.

Inputs:

- i) Service Name

Outputs:

- i) Port Number
- ii) Web Services Directory

4.3.11. Find

This message performs a depth first search on the network to find the location of a service.

Inputs:

- i) Service Name

Outputs:

- i) A packaged directory containing the requested service

4.3.12. Mass Message

The mass message is used to multicast a message to a list of recipients. This is the only message protocol we use and it is ca-

pable of sending to any size list of recipients in the network.

Inputs:

- i) List of recipients
- ii) The message

Outputs:

- i) “YES”, “NO”, or “DON’T HAVE”

5. Client Design

The client software is intended to be a simple interface to the middleware. We also designed it so that the least amount of changes would have to be made to existing code. What our middleware does is provide the WSDL location as an http path to the client so it can then create a SOAP object. For example, a normal SOAP request made with the SOAPy package in Python may look like:

```
wSDL='http://www.foo.com:8080/Foo.jws?w  
sdl'
```

```
x = WSDL.SoapProxy(wSDL)
```

```
print x.FooMethod('bar')
```

In order to use the BD middleware all that is done is the hard coded http path is now replaced with a method that queries a given node for a service. For example, to perform the previous example with our client, the only line that would change would be:

```
wSDL=get_wSDL('foo.com:9001', 'Foo.jws')
```

This method will query the BD node at foo.com listening on port 9001 for the http address of some wSDL file and is then returned as a string, replacing the hard coded one that was there before. The returned string may or may not be from the node being queried, it is important to note that the first parameter of get_wSDL is simply the entry point into the system and the actual location returned could be anything that is contained within.

The method itself is currently implemented in Python so that we may use it in our Python/CGI scripts. If the functionality were to be expanded so that it could be used with Perl/CGI or PHP, the client written in that language would simply have to follow the protocol. To issue a request to the entry point to find the needed service, the client sends a WHEREIS request (See 4.3 Protocol), which then returns the host name, the port and the directory from the root. This information is then compiled into a string and returned to the method invoker.

It is better if the user of the client re-invokes the client for each request. For example, if a request to Foo.jws is made five times, it is better for the system to reissue the WHEREIS query so that the system can properly load balance. This is a downside to using our system, however we felt that having the burden of doing this on the client was simpler to implement than dynamically re-routing the SOAP object to a different WSDL file as the system load changes.

6. Testing

The tools used to analyze the tests that we performed are described below, however to see the actual results of the tests, see the attached Testing Results Document.

6.1. Audit Tool

We provide a system audit tool called BDAudit which is a simple Python/CGI script that displays a snapshot of important information about the currently running system. The script contacts a single node first and queries STATUS (see 4.3. Protocol). Then, the tool contacts all the nodes the origination node knew about and queries STATUS. The tool continues to query these nodes' adjacent nodes and so on until no more nodes are left to query. All the information gathered from the individual nodes are compiled into a data set. Once this information is available, BDAudit displays information separated into different categories: Node Topology, Connectivity Statistics, WS-Policy Intersections, Weight

Distribution Graph, Web Service Statistics and Individual Server Statistics. Since BDAudit is a standard CGI script, it can be accessed just by providing the path and location of a BD middleware node. An example access to the BDAudit is as easy as follows: <http://www.csee.umbc.edu/~bwilson1/cgi-bin/BDaudit.cgi?site=130.85.94.186:9000>

6.1.1. Node Topology

The node topology is a simple graphical representation of the overview of the system. It is designed to quickly allow the BDAudit invoker to view which nodes are connected, which nodes have WS-Policy conflicts, which services each node has and whether the service is dynamic or static (see 3.2.2). Also, nodes that appear to be down which state has not been discovered by the other nodes are displayed in the graph.

Sites are displayed as ovals with an IP address and a Port, representing the location of the middle-ware. This oval can be green or red, depending on if the node is perceived to be accepting connections or not, respectively. Any two system nodes are then connected by two directed arrows, representing what a given node believes the state of the other node is. A green edge represents an edge which can be used and the other node appears to be present. A red edge represents that the node the edge is coming from does not believe that the other node is in the system. A yellow edge represents the two nodes' WS-Policies do not have an intersection, thus cannot talk, however the node knows the other exists. Also, there are black ovals which represent web services in the system. There is then a directed edge from a site which has the service to the oval representing the service. This edge is black if the service is static for that node or blue if the service is dynamic for that node.

The node topology graph is perhaps the most useful tool while testing and observing the system as it conveys information in a quick manner. Replication can be observed by watching more blue edges attach from nodes to services. When a node goes down, it turns red and slowly the green and yellow edges turn to red. When a new node is intro-

duced to the system, one can observe how it builds connections with other nodes while it is coming up.

The node topology graph is created by the graphviz dot program [5] which dynamically generates node/edge graphs. The output is then saved as an image and displayed in the BDAudit output.

6.2.2. Connectivity Statistics

The connectivity statistics section of the BDAudit tool conveys information that tells the timestamp of the most up to date information a node has about another node. The rows of the graph are each node's connections to others. When the row and the column of a node intersect, the data in the cell states the last update received from that node. Note that this update can either be from the node itself or through a pair-wise synchronization session with a 3rd party node. Also, an "X" appears in the spot if the row node does not contain any information about the column node.

The table is useful for demonstrating how updates propagate. Usually a node that is several hops from another one will have a less-updated version of that node, which can be viewed from the timestamps. The cells with an X also represent the same thing as a red edge in the Node Topology Graph.

6.2.3. WS-Policy Intersections

The WS-Policy Intersection table shows the WS-Policy intersections between each node with every other node. The columns of the table are two Host/WS-Policy pairs and a column with the intersection of the two policies. If two policies have no intersection "NULL" appears in the intersection column and the line is italicized. These lines with "NULL" should correspond to green lines in the Node Topology graph.

The general case of knowing if two nodes have WS-Policies that intersect is shown in the Node Topology graph. This table displays in more detail the actual policies and the actual intersections of these policies.

6.2.4. Weight Distribution Graph

The Weight Distribution Graph is another visual tool which demonstrates the load of the system by looking at each node. It is designed to be able to view the system load at a glance, view which nodes are overloaded, view which nodes are dynamic or static and view which specific services are being used on the system.

The graph is set up with the system nodes in the left column. If the text is colored black, the site's load is below its threshold. If it is colored red, the site's load is above its threshold. To the right of each of this is a bar that extends horizontally that shows the weight of each individual service. The bar colors correspond to the color key at the bottom of the graph and will be shaded with diagonal black lines if the service is dynamic. Each row has a vertical line which represents the threshold for each site.

The Weight Distribution Graph is the best way to view the load at once. Although the specific weight numbers are specifically stated in detail in the last two sections of the audit tool, it is easier to view how the loads relate to each other with a bar graph. This graph is useful for viewing which services are currently being used on the system and whether the use of these services are about to cause a site to go over its threshold. At this point some replication should be seen amongst the other nodes.

6.2.5. Web Service Statistics

The web service statistics portion of BDAudit provides specific load information about each service in the system. This is the same information the Weight Distribution Graph provides, however it gives numbers instead of a visual representation. Also, interesting statistical percentages are shown as well as statistics about static and dynamic services.

First, some general information is presented, such as the number of different web services in the system, the total number of service

replicas that exist in the system, the number of static replicas, the number of dynamic replicas, the total weight of the system and the average weight of each service. The second half of this section is a graph which shows the weight, the percentage of the total weight this service has of the system, number of replicas, number of static replicas and number of dynamic replicas for each service.

This table is good if you want to know what the popularity of a specific service is and how many times it has been replicated in the system. The weight percentage is also important in determining how heavy the service is in comparison to others.

6.2.6. Individual Server Statistics

The Individual Server Statistics table is designed to demonstrate statistics for each node in the system. It basically provides the same information as the previous section, the Web Service Statistics, however it provides the information centered on the nodes, not the services.

For each node in the system, the table shows the weight, the receiver threshold, list of static services, the weight of each static service on that node, a list of dynamic services and the a weight of each dynamic service on that node.

This table is extremely similar to the Weight Distribution Graph and is provided in a very similar format; the only difference is that it provides exact numbers for the weights, which can be useful for taking a closer look at each node.

7. Conclusions and Future Work

After extensive testing and analysis of our design, we can conclude that our system does succeed at load balancing, service replication, and fault tolerance. The eventual consistency cache model maintains a stable view of the system even in the presence of multiple node failures and discoveries.

7.1 Security Levels

One foreseeable extension to this project would patch a vulnerability that we have found. Our implementation of WS-Policy between servers allows the possibility for a client requesting a service from node A and the only way to get it is by replicating off a node which does not match the client policy. This is a problem if the node did not want to service the client for a reason. Implementing a model similar to Bell Lapadula security levels on a node by node basis would solve this problem. Every node will be assigned a security level and nodes that are higher level cannot be replicated off of by lower level nodes. Also, lower nodes would not be able to execute services off of higher nodes. Higher nodes can, however, assimilate the services on the lower levels and bring those services up to a higher security level to accommodate the balancing needs of the higher level clients. This could be imple-

mented by higher nodes simply telling lower nodes that they have no services.

7.2 Versioning

Versioning is a feature we wanted to implement. We would use our eventually consistent caches to detect newer versions of services. Once a newer version is detected our node would then go and fetch it, effectively updating his service. This would require modifying the cache data structure, keeping a version number in the BDCert and checking for updates during synchronizes.

8. References

- [1] Terry, Douglas B., et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", Proc. 15th Symposium on Operating Systems Principles, (SOSP), Dec 1995
- [2] Sahasrabuddhe et. al., "Multicast Routing Algorithms and Protocols: A Tutorial". (IEEE). January 2000
- [3] Anne H. Anderson, "An Introduction to the Web Services Policy Language (WSPL)". 5th IEEE International Workshop on Policies for Distributed Systems and Networks. (IEEE). June 2004
- [4] Anne H. Anderson, "Predicates for Boolean web services policy languages". (Sun Microsystems). May 2005
- [5] "Graphviz – Graph Visualization Software". <http://www.graphviz.org>