A Survey of Fault Tolerant Web Service Solutions

Abstract

The purpose of this paper is to provide the reader with an overview of fault tolerant web services. It will begin with an overview of the various challenges that present themselves when designing such a system. Then it will follow a view of the progression of different solutions, from early solutions to more recent. Finally, we will arrive at some approximation of the state of the art for the design of these types of systems.

1. Introduction

As the web grew from its infancy into a nearly ubiquitous platform, so did the need for fault tolerant and scalable web servers. Reliability and uptime have become the central features around which commercial web solutions are sold. Delivering reliability and scalability presents a set of challenges that are difficult to overcome in practice. Furthermore, as the web evolved from merely static content to dynamic and interactive content, an even larger set of challenges presented themselves. Not only must web servers replicate static content, but they must now preserve session information between all web server nodes in real time. If the given web content is database driven, the web nodes must share a database as well. If not properly designed, this database can act as a single point of failure. An entirely different set of challenges presents themselves if the system requires that web servers be distributed geographically.

2. General Concepts

There are certain general goals common to most all fault tolerant web services. The first and most fundamental concept to fault tolerant web services is redundancy. At the very least, all fault tolerant server systems must use redundant web servers. In order for a system to tolerate $p$ failures, there must be $p+n$ redundant servers, where $n$ is the minimum number of servers necessary to handle all of the incoming web traffic. This detail is often easy to overlook, but is critical to maintaining services in the presence of a fault. If a redundant server system has fewer than $p+n$ servers and experiences $p$ failures, a cascade failure can occur as the remaining servers each become overloaded. In this scenario, the performance of the system can spiral downward and cause a complete system failure. This concept applies not only to web servers, but also to database servers, load balancers, file servers, network links, and any other subsystem on which the web servers depend in order to serve content.

The second general concept common to fault tolerant web services is transparency. A fault tolerant system must be able to map incoming requests to a single internet domain name or address to multiple web servers without any interaction from the client. Furthermore, the HTTP protocol itself contains very few mechanisms for fault tolerance. Thus most HTTP clients do not have any established procedure, other than
simply retrying a request, for fault recovery. Thus any fault tolerant system will need to be handled by the server and be completely transparent to the client application.

The final general concept related to fault tolerant web services is the concept of tiered architectures. Each of the fault tolerant web service solutions discussed here will fall into two categories: tiered and flat. The flat approach contains only one type of server node--the web server. In this simplistic design, web servers are simply replicated. Each server contains its own copy of all data required to serve the website. This approach has the advantage of being easy to implement and easy to scale. Unfortunately, the flat architecture cannot easily handle dynamic content or stateful user interactions. To handle these situations, a tiered architecture must be used. The tiered approach separates servers into frontend and backend servers. The frontend servers are responsible for serving client HTTP requests, but actual web content (HTML files, databases, video content, etc.) is stored on the backend servers. The two tiers are generally connected via a private network. Having a tier architecture allows dynamic content to be served, but it also increases the complexity of the system. Each tier must provide \( p+n \) redundancy in order to have a total system redundancy of \( p+n \).

### 3. Incoming Request Distribution

When differentiating between various fault tolerant and scalable web solutions, one of the key design elements that can be used to categorize techniques is how incoming requests are distributed to various web nodes. This is an important design choice as it affects how the workload is distributed and the system recovers from a fault of a sub component. It also addresses the mapping of a single internet domain name to multiple servers, thus allowing transparency to the client. In the following, several different techniques for distributing incoming requests will be discussed.

#### 3.1 Round Robin DNS

One of the earliest fault tolerant web solutions was the National Center for Supercomputing Applications (NCSA) scalable web server [5]. This was a fully implemented tiered solution designed to handle the rapidly growing web traffic seen by the NCSA's web service beginning in February of 1994. This system used a technique known as round robin DNS to distribute incoming requests. When using round robin DNS, each server is given its own public IP address. The domain name server then attempts to distribute incoming requests by evenly distributing domain name requests to multiple IP addresses. In order to provide fault tolerance, the domain name server will return a list of multiple IP addresses, starting with the IP address that it would like the client to try first. Thus, the first arriving client will receive server A's IP address at the top of the list and the second arriving client will receive server B's IP address at the top and so on. If the first IP address on the client's list does not respond, the client will simply retry the request to the next IP address on the list. If a server goes down, it will be removed from the list of IP addresses that the DNS servers return.

There are several drawbacks to this approach. The first, and arguably most severe drawback, is that this technique can cause load imbalance. DNS is a caching protocol. When a client attempts to resolve a domain, it will query its local nameserver which will
in turn query the public name servers for said domain. The local nameserver will cache the results for some time period (anywhere from 5 minutes to 3 days). If any other clients make a request during this time period, they will receive the exact same IP list as the first client did. Thus if a large number of clients are using the same local nameserver, all of the clients will make their HTTP request to the same server which could become overloaded. This could end up causing a fault, rather than preventing one. The second major drawback to the round robin DNS request distribution technique is that there is no standard client behavior for what to do if a request fails. Generally the client will wait 30 or 40 seconds for a request to return and then move down its list of IPs. There are, however, no guarantees that the clients will follow this behavior and thus this is an unreliable way to mask a server failure.

3.2 Client Based Request Distribution

Another early technique was to push request distribution to the client. This technique was used by Netscape in early 1996. Since Netscape distributed its own HTTP client at the time, it could embed some information into the client to allow it to distribute requests to its home page. When a user entered the URL http://home.netscape.com/ into the Netscape Navigator HTTP client, it would intercept the request and translate it into something of the form http://homeX.netscape.com where \( X < 33 \) and \( X > 0 \) [6]. Each homeX.netscape.com address resolved to a different public IP address which resolved to a different server. Fault tolerance was largely the same as round robin DNS. If a request times out, simply pick a new address and random and resend the request. Pushing the host selection to the clients did, however, allow Netscape to get around the load imbalance problem with round robin DNS that is caused by DNS caching. In this scheme it doesn't matter if the domain names are cached in at a local DNS server, because clients will choose a different hostname at random.

The primary drawback of this technique is the reliance on the HTTP client. This technique is the least transparent of all the techniques discussed. It was never standardized and other web browsers that attempted to access home.netscape.com would not behave correctly. Furthermore, there was no generic mechanism which allowed other sites to use this technique. For this reason, this technique never really saw widespread adoption.

3.3 Active Load Balancing

In order to address the issues with round robin DNS and the client based approach, another technique was developed to perform request distribution. This technique is known as active load balancing. This technique makes use of a special type of router called a load balancer which is used to distribute incoming requests. The load balancer will listen on a single external IP address and each of the web servers will be given a separate private internal address. The load balancer maintains a table of active web servers and it will poll each one of them on a short interval to ensure that it is notified when a fault is detected. As a client request comes in, the load balancer will select a web server from its list and forward the request via Network Address Translation
(NAT). It may select a web server based on a number of different criteria, such as server load, number of active connections, or simply in a round robin fashion.

The active load balancing technique presents several key advantages over round robin DNS. The first, is of course, that the load balancer intelligently picks which server will accept each incoming request. This avoids the load imbalance issues caused by DNS caching when using round robin DNS. The second advantage is additional fault transparency from the client's perspective. Since the load balancer has a single public IP address, multiple retry/timeout cycles are not necessary to find a responsive server in the case of a fault.

A keen observer will note, that as is often the case, in solving the issues related to round robin DNS we have also introduced greater complexity to the system as a whole. This load balancer is a new single point of failure. A fault in the load balancer will result in a total failure of the system as there will no longer a link from the private network to the internet. To further complicate matters, we cannot simply replicate load balancers as easily as we can HTTP servers. Unlike HTTP, NAT requires state information that must be maintained to prevent service interruption. Furthermore, we must have a method of transferring the public IP address between load balancers.

Fortunately, these problems can be addressed and we can provide proper \( p+n \) redundancy for the load balancer subsystem. In order to hand off the public IP address between load balancers, they can use a special protocol to share a public IP address between the load balancers. One such protocol is known as the Common Address Redundancy Protocol (CARP) [9]. When using CARP, each load balancers' external interface listens on a common virtual MAC address. In this configuration, one of the load balancers will act as a master and the others will act as backup servers. The master server will perform all of the routing of traffic. The backup servers will each poll the master server continuously to ensure that it remains up. In the event of a fault in the master balancer, the backup balancers will select a new master according to a pre-determined order. This new master will advertise the shared MAC address to the public switch which will begin forwarding traffic to the appropriate physical link. In this way the public IP address will be transparently transferred to the newly active balancer in a manner which is completely transparent to clients.

There remains only one additional step to implementing fault tolerance in the load balancer layer. The master load balancer must have a mechanism for replicating the NAT
table to all of backup load balancers. The master will accomplish this by broadcasting all changes to its NAT table to each of the backup load balancer. Thus, when a backup load balancer takes over, it will have all the information necessary to continue transferring packets from the clients to the web servers. Presuming that the polling interval and handover time are less than the TCP session timeout value, the master balancer can go down and the backup balancer can take over without even loosing a single TCP session.

4. Backend Fault Tolerance

Incoming request distribution and web server redundancy remain only half of the picture, however. There still remains the question of how the data that needs to be served to the clients is distributed throughout the system. The simplest case, as mentioned earlier, is the flat distribution approach. In this scheme all web servers maintain a local copy of all the data. Changes to the data can be pushed to all web servers using a version control system [6] at some regular interval. The flat approach, however is not suited to systems whose data is changing rapidly.

4.1 Block Level File Server Replication

In order to share file information in real time, the web nodes must use some time of shared file system over the network. One such file system which has attained widespread adoption is the Network File System (NFS) [8]. With NFS data is stored in a centralized file server. File locking, access control, and scheduling are all handled by the NFS server. While this eliminates the problem of data sharing between web nodes, we have once again increased the complexity of the system and reintroduced a single point of failure. If the file server fails, the whole system fails.

To address this new single point of failure, we must once again replicate servers. NFS does not natively support replication or distribution. To provide replication for NFS servers a block level replication system such as DRBD [7] can be used. DRBD is a system which supports synchronous and asynchronous mirroring of block devices over a network. With synchronous mirroring, servers will wait for acknowledgements from their replication partners before continuing. This provides the maximum level of data consistency, but also is most sensitive to latency. Asynchronous mirroring provides a more relaxed consistency model which allows filesystem mirrors to be distributed over greater distances [10].

Fault tolerance for these mirrored NFS server works in a manner quite similar to that of the redundancy of the load balancing layer. One filesystem is the master which handles incoming requests and the others act as passive backups that monitor the master server to ensure that it remains online. These servers can share an IP using CARP or another similar technique. When the primary server fails a backup server will take over the master's IP address and begin serving files to the web nodes. It is important to note that when a failed server is restored it must replicate all of the changes made to the file system during the time in which it was offline. A careful system designer must take in to account the increased load added to the system during this mirror resync process.
4.2 Database Replication

The final component of our fault tolerant web service is the database system. Many modern web applications are database driven. The database subsystem, just as with the rest of our subsystems, must be replicated to be fault tolerant. There are many different varieties of database replication available, but some of the more common are master-master and master-slave replication setups. In a master-slave replication schema, the slave acts as a read-only replica of the master server. This configuration is simpler than a master-master replication schema and may have better performance depending on how bad lock contention is between servers.

5. Conclusion

From these examples, the fundamental principles of fault tolerant web solution design become clear. Each subsystem must provide $p+n$ reliability and do so in a way that is transparent to each of the other subsystems. To build the most reliable system, errors must be detected and masked as quickly as possible to prevent service interruption for the clients. Proper implementation of these principles will allow us to continue to build fault tolerant web solutions to feed today's ever growing demand for web based content.

6. Resources


[9] "PF: Firewall Redundancy with CARP and pfsync." 
http://www.openbsd.org/faq/pf/carp.html

http://www.drbd.org/