SOFTWARE ENABLED LOAD BALANCER BY INTRODUCING THE CONCEPT OF SUB SERVERS

Dhanda Mamta*
Assistant professor, JMIT, Radaur, Haryana
Gupta Parul**
Scholar, JMIT, Radaur, Haryana
parulgarg05@gmail.com

ABSTRACT
In computer networking, load balancing is a technique to spread work between two or more computers, network links, CPUs, hard drives, or other resources, in order to get optimal resource utilization, maximize throughput, and minimize response time. Using multiple components with load balancing, instead of a single component, may increase reliability through redundancy. The balancing service is usually provided by a dedicated program or hardware device (such as a multilayer switch). One of the most common applications of load balancing is to provide a single Internet service from multiple servers, sometimes known as a server farm. Commonly load-balanced systems include popular web sites, large Internet Relay Chat networks, high-bandwidth File Transfer Protocol sites, NNTP servers and DNS servers. In this paper we propose a software enabled load balancing model by introducing the concept of sub servers for regional services to overcome the overhead of the main server.

Keywords: FTP, NNTP Servers, DNS Servers, SLB

1. INTRODUCTION
Apache Traffic Server is an Open Source project, originally developed as a commercial product by Inktomi, and later donated to the Apache Software Foundation (ASF) by Yahoo! Inc. Apache Traffic Server was accepted as a Top-Level Project in April of 2010, after 6 months of incubation. Graduating as a TLP is a milestone not only for the community, but also shows ASF’s commitment to Traffic Server, as well as by all the contributors. Yahoo! has actively used the original Traffic Server software for many years, serving HTTP requests for many types of applications:

- As a Content Delivery Network (CDN), serving static content for all of Yahoo’s web sites
- For connection management across long distances, and providing low-latency connectivity to the users
- As an alternative to Hardware Server Load Balancers (SLBs)
As such, TS already is (and has been for several years) a critical component of Yahoo!’s Network. By releasing Traffic Server to the Open Source Community, a new tool is now readily available for anyone to use.

1.1 WHY APACHE SOFTWARE FOUNDATION

This presentation does not focus on Yahoo!’s decision to open-source Traffic Server, and the choices that were made during the process. However, it’s useful to understand why Yahoo! chose ASF, and what benefits we derive from being an ASF Top-Level Project.

Being part of an already established and well-functioning Open Source community brings immediate benefits to the project:

- We benefit from the many years of experience of ASF leadership in Open Source technology.
- We immediately gained new contributors to the project.
- There is plenty of existing source code, skills and experiences in the ASF community, into which we can tap.
- We are part of a reputable and well-maintained Open Source community.

1.2 HTTP PROXY AND CACHING

HTTP proxy servers, with or without caching, are implementations of an HTTP server with support to act as an intermediary between a client (User-Agent), and another HTTP server (typically referred to as an Origin Server). It’s quite possible, and in many cases desirable, to have multiple intermediaries in a hierarchy, and many ISPs will proxy all HTTP requests through a mandatory intermediary.

There are three primary configurations for a proxy server:

- Forward Proxy – This is the traditional proxy setup, typically used in corporate firewalls or by ISPs. It requires the User-Agents (e.g. browsers) to be configured and aware of the proxy server.
- Reverse Proxy – In a reverse proxy setup, the intermediary acts as any normal HTTP server would, but will proxy requests based on (typically) a specific mapping rule.
- Intercepting Proxy – This is similar to Forward Proxy, except the intermediary intercepts the HTTP requests from the User-Agent. This is also typically done by ISPs or corporate firewalls, but has the advantage that it is transparent to the user. This usually is also referred to as Transparent Proxy.

Any HTTP intermediary must of course function as a basic HTTP web server. There is definite overlap in functionality between a proxy server and a regular HTTP server. Both typically provide support for access control (ACLs), SSL termination and IPv6. In addition, many HTTP intermediaries also provide features such as:

- Based on the incoming request, finding the most appropriate Origin Server (or another intermediary) from which to fetch the document;
- Providing infrastructure to build
redundant and resilient HTTP services;  
✓ Cache documents locally, for faster access and less load on Origin Servers;  
✓ Server Load Balancing (SLB), by providing features such as sticky sessions, URL-based routing, etc.  
✓ Implementing various Edge services, such as Edge Side Includes (ESI);  
✓ Acting as a firewall for access to HTTP content: providing content filtering, anti-spam filtering, audit logs, etc.

Traffic Server can perform many of these tasks, but obviously not all of them. Some tasks would require changes to the internals of the code; and some would require development of plugins. Fortunately, Traffic Server, similar to Apache HTTPD, has a feature-rich plugin API to develop extensions. Efforts are being made to not only release a number of useful plugins to the Open Source community, but we also aim to improve and extend the plugin APIs to allow for even more complex development. We are also starting to see the community contribute new Traffic Server plugins.

1.3 TRAFFIC SERVER UNDER THE HOOD

Apache Traffic Server differs from most existing Open Source proxy servers. It combines two technologies commonly used for writing applications that deal with high concurrency:
1) Asynchronous event processing  
2) Multi-threading  
By combining these two technologies, TS can draw the benefits from each. However, it also makes the technology and code complex, and sometimes difficult to understand. This is a serious drawback, but we feel the positives outweigh the negatives. Before we discuss the pros and the cons of this decision, we’ll give a brief introduction to these two concepts.

1.3.1. Asynchronous Event Processing

This is actually a combination of two concepts:
1) An event loop  
2) Asynchronous I/O

Together, this gives us what we call Asynchronous Event processing. The event loop will schedule event handlers to be executed as the events trigger. The asynchronous requirement means that such handlers are not allowed to block execution waiting for I/O (or block for any other reason). Instead of blocking, the event handler must yield execution, and inform the event loop that it should continue execution when the task would not block. Events are also automatically generated, and dispatched appropriately, as sockets and other file descriptors change state and become ready for reading or writing (or possibly both).

It is important to understand that an event loop model does not necessarily require all I/O to be asynchronous. However, in the Traffic Server case, this is a fundamental design requirement, and it impacts not only how the core code is written, but also how you implement plugins. A plugin cannot block on any I/O calls, as doing
so would prevent the asynchronous event processor (scheduler) from functioning properly.

1.3.2. Multi-Threading

Different Operating Systems implement multi-threading in different ways, but they are generally a mechanism to allow a process to split itself into two or more concurrently running tasks. These tasks (threads) all exist within the same context of a single process. A fundamental difference between creating a thread and creating a new process is that threads are allowed to share resources (commonly) shared between separate processes. As a side note, it is typically much less expensive for an OS to switch execution between threads than between processes.

Threading is a simpler abstraction of concurrency than the asynchronous event processing, but every OS has limitations on how many threads it can handle. Even though switching threads is lightweight, it still has overhead and consumes CPU. Threads also consume some additional memory, of course, although typically not as much as individual processes will.

1.3.3. Why make it twice as complicated?

Now that we have a basic understanding of what these concurrency mechanisms provide, let’s discuss why Traffic Server decided to use both. This is an important discussion because it will help you decide which HTTP intermediary solutions you should choose.

Multi-threading is a popular paradigm for solving concurrency issues because it is a well-understood and proven technology. It is also well-supported on most modern Operating Systems. It solves the concurrency problem well, but it does have a few problems and concerns, such as:

- Writing multi-threaded applications is difficult, particularly if the application is to take advantage of shared memory. Lock contention, deadlocks, priority inversion and race conditions are some of the difficulties with which developers will need to confront.
- Even though threads are lightweight, they still incur context switches in the Operating System. Each thread also requires its own “private” data, particularly on the stack. As such, the more threads you have, the more context switches you will see, and memory consumption will increase linearly as the number of threads increases.
- It generally is easier to program for asynchronous event loops, and there are many abstractions and libraries available that provide good APIs. Some examples include libevent[2] and libev[3] for C and C++ developers. (There are also bindings for many higher-level languages for both these libraries, and others.) Of course, there are a few limitations with event loops:
  - The event loop (and handlers) typically only supports running on a single CPU.
  - If the event loop needs to deal with a large number of events, increased latency can occur before an event is processed (by the nature of the events being queued).
  - To avoid blocking the event loop, all I/O needs to be asynchronous. This
makes it slightly more difficult for programmers, particularly when integrating existing libraries (which may be synchronous by nature).

Traffic Server decided to combine both of these techniques, thus eliminating many of the issues and limitations associated with each of them. In Traffic Server, there are a small number of “worker threads”; each such worker thread is running its own asynchronous event processor. In a typical setup, this means Traffic Server will run with around 20-40 threads only. This is configurable, but increasing the number of threads above the default (which is 3 threads per CPU core) will yield worse performance due to the overhead caused by more threads.

1.4 CONTENT DELIVERY NETWORKS

A Content Delivery Network, or CDN, is a service or infrastructure used to deliver certain types of HTTP content. This content is usually static by nature, where Edge caches can effectively store the objects locally for some time. Examples of CDN-type content are JavaScript, CSS, and all types of images and other static media content. Serving such content out of a caching HTTP intermediary makes deployment and management significantly easier, since the content distribution is automatic.

A CDN automates content distribution to many collocations, simplifying the operational tasks and costs. To improve end-user experience, a CDN is commonly deployed on the Edge networks, assuring that the content is as close as possible to the users.

There are several reasons this is beneficial:

✓ Cost reductions, and more effective utilization of resources
✓ Faster page load times
✓ Redundancy and resilience to network outages.

The biggest question you face when deciding on a CDN is whether to build it yourself or to buy it as a service from one of the many commercial CDN vendors. In most cases, you are probably better off buying CDN services initially. There are initial costs associated with setting up your own private CDN on the Edge, and this should be considered when doing these evaluations.

Notwithstanding the above limitations, I am a strong proponent of building your own CDN, particularly if your traffic is large enough that the costs of buying the services from a CDN vendor are considerable. Further, to be blunt, building a CDN is not rocket science. Any organization with a good infrastructure and operations team can easily do it. All you need is to configure and deploy a (small) number of servers running as reverse proxy servers for HTTP (and sometimes HTTPS).

1.4.1. Building a CDN with Apache TS

Apache Traffic Server is an excellent choice for building your own CDN. Why? First of all, it scales incredibly well on a large number of CPUs, and well beyond Gigabit network cards. Additionally, the technology behind Traffic Server is well-geared toward a CDN:
The Traffic Server cache is fast and scales very well. It is also very resilient to corruptions and crashes. In over 4 years of use of the Yahoo! CDN, there has not been a single (known) data corruption in the cache.

The server is easy to deploy and manage as a reverse proxy server. The most common configuration tasks and changes can be done on live systems, and never require server restarts.

It scales well for a large number of concurrent connections, and supports all necessary HTTP/1.1 protocol features (such as SSL and Keep-Alive).

As a proven technology, Traffic Server delivers over 350,000 requests/second, and over 30Gbps in the Yahoo! CDN alone. This is an unusually large private CDN, with over 100 servers deployed worldwide. Most setups will be much smaller.

Of course, many of the other existing HTTP caches can be used to build a CDN. We believe Traffic Server is a serious contender in this area, but there is healthy competition.

1.4.2. Configuration

We are not going to go into great details about how to configure Apache Traffic Server for building your CDN. There are primarily two configuration files relevant for setting up Traffic Server as a caching intermediary:

- records.config – This file holds a number of key-value pair, and in most situations the defaults are good enough (but we will tweak this for a CDN).
- remap.config – This configuration file, which is empty by default, holds the mapping rules so that TS can function as a reverse proxy.

Of course, there can be much more complex configurations, particularly in the remap configuration, but the examples demonstrate how little configuration would be required to get a functional CDN with almost zero configuration using Apache Traffic Server.

1.5. CONNECTION MANAGEMENT WITH ATS

Connection management is very similar to a CDN; in fact, many CDN vendors also provide such services as well. The purpose of such a service is primarily to reduce latency for the end-user. Living on the Edge, the connection management service can effectively fight two enemies of web performance:

- TCP 3-way handshake. Being on the Edge, the latency introduced by the handshake is reduced. Allowing for long-lived Keep-Alive connections can eliminate such latency entirely.
- TCP congestion control (e.g. “Slow Start”). The farther away a user is from the server, the more visible the congestion control mechanisms become. Being on the Edge, users will always connect to an HTTP server (an Origin Server or another intermediary) that is close.

The following picture shows how users in various areas of the world connect to different servers. Some users might connect directly to the HTTP web server (the “service”), while others might connect to an intermediary server that is close to the
Figure 1. Connection management

Connections between the intermediaries (the connection managers) and Origin Servers (“web site”) are long-lived, thanks to HTTP Keep-Alive. Reducing the distance between a user and the server, as well as eliminating many new TCP connections, will reduce page-load times significantly. In some cases, we’ve measured up to 1 second or more reduction in first page-load time, only by introducing the connection manager intermediaries.

1.6. HTTP SERVER LOAD BALANCER

Load Balancing is a basic technique for routing traffic, such as HTTP requests, to a server in a way that achieves optimal performance, high availability, or easier service implementation. HTTP Server Load Balancer can handle any (or most) TCP and UDP protocols, while an HTTP specific SLB would obviously only do HTTP and perhaps HTTPS. With an HTTP Server Load Balancer, you can:

✓ Assure a particular user always hits the same backend (real-server) to serve a particular URL.
✓ Assure there is always at least one real-server available to serve any type of request.

Getting users, or requests, associated with a smaller number of servers can significantly improve the performance of your applications. You can achieve better cache affinity, smaller active data sets, and easier (and faster) code to evaluate.

The following picture depicts a typical HTTP Server Load Balancer setup.
omewhat lacking in Apache Traffic Server, and our hope is that discussing this openly will attract attention and interest from other developers who wish to work on these features. In all fairness, some of the other TP intermediaries do a great job; we have much to learn from them.

Section 2, we discusses the proposed architecture work. In section 3, we discuss the proposed methodology (work model and assumption). Finally, the discussion of future work is discussed in section 4.

PROPOSED ARCHITECTURE

EXISTING SYSTEM

Existing system they are not using sub servers for centralized server so there is a chance to occur time delay in message delivering and packet discarding and complexity in receiving more messages are the main draw backs available in existing system.

"backend" servers, which usually represent the load balancer. This allows the load balancer to reply to the client without the client ever knowing about the internal separation of functions. It also protects clients from contacting backend servers directly, which may have security benefits by hiding the structure of the internal network and preventing attacks on kernel's network stack or unrelated services running on other ports.
3. PROPOSED METHODOLOGY

This paper focuses on the concept of software enabled load balancer using java platform. We will simulate the java code using simulator to get better results considering following parameters mentioned below:

✓ Load balancing
✓ Message distribution

3.1 Load Balancing

In this module we have checking the availability of the server space and its sub server’s space before receiving the message. If there is no space to receive message then intimate to the source no more space in server to receive message.

3.2 Message Distribution

Centralized server receives message from source it checks its own availability if it is exceed then send message to the sub servers before send the message it checks sub server’s availability too if it is exceed then inform to the source no more space to receive message.

4. REFERENCES


