PROJECT PROPOSAL

SCALABLE WEB SERVER

Raman Bansal and Ruchir Garg

Dr. Surya Prakash

Background

A **web server** is a system that serves content to clients via HTTP. Like any other server, a web server must wait for an incoming request, parse the request, find the desired resource, then return that resource out to the requesting client. These resources can be anything, of any size, from small HTML files that are simply text, to massive video files, or the results of expensive computations. A generic web server must be able to robustly handle any of these.
In general, the pseudocode for a server looks like this:

```
init_server(); // Set up sockets, initialize cache, etc.

while(1)
{
        connection = accept_connection(socket);
        request = read_request(connection);
        resource = parse_request(request);
        serve_resource(resource, connection);
        close(connection);
}
```

It is also important to note that web servers suffer heavy traffic--particularly as a web site becomes more popular. Facebook notes that it experiences millions of daily active users that is too many HTTP requests. This is a clear motivation to parallelize--it is infeasible to have a server fulfill these requests serially, forcing users to wait on others before their feed can load. Requests come in parallel, so they should be serviced in parallel.

Ideas

Parallelism

Luckily, web servers are conceptually very amenable to parallelism. Individual requests are generally completely data-parallel, in that they have no dependencies on other requests.

Caching

Additionally, there can be a bonus in terms of locality based on the structure of the web site a user visits. A very simple example can be seen on Facebook. At the top-left corner of every page on FB is the FB logo. Every request to a page must include a request to this image, so it can be cached for better performance. This motivates the use of caching. Further, most queried data like top posts, trending on Facebook can be cached. Since fetching from database is an expensive task, if we can cache the data (key-value pair), we would save a lot of time. Facebook uses memcache for that.

Elastic

Another challenge to designing a parallel web server is that server traffic, while high, is also variable. Imagine an Flipkart. Traffic on this site is going to be highest sometime where it is evening across the country, once people are back from their jobs. On the other hand, when it is late night or early morning, traffic will be low as most people are asleep.Since a web server must provision for the highest load, each day will result in wasted computational resources (*e.g.*, electricity). Traffic will be enormous on Big Billion days and very low on the very next day. This problem can be solved via **elasticity**, a concept in which resources are added or removed from the server system on-demand, based on the total load on the system. To make good use of elasticity, the server system should be distributed across multiple physical machines, so that an individual machine can be considered as a single computational resource to be added or removed.

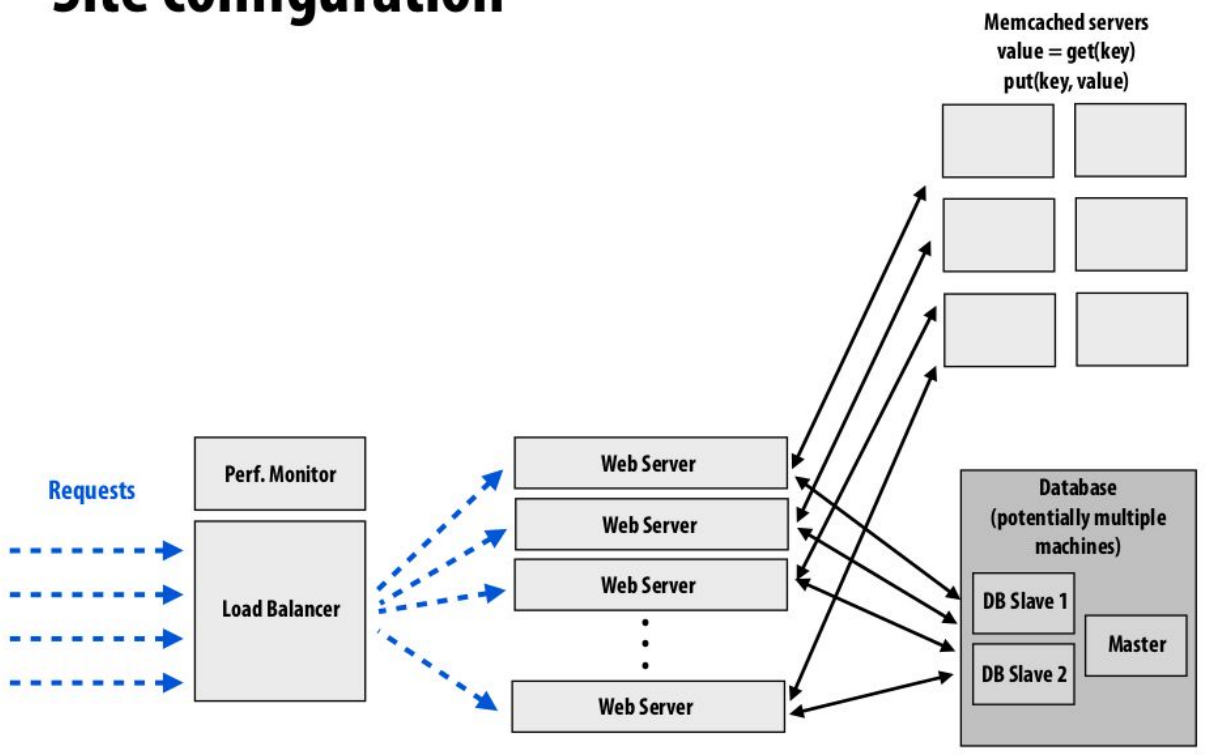
Approach

We will use MPI, which abstracts away the communication into messages. Our system would use different processes, called `Load Balancer`, and `Web Server`, `cache(possibly memcache), Database(Master/Slave)` respectively. The Load Balancer is a single process that accepts all incoming requests to the server. It then redirects the client to a chosen `Web Server` via an HTTP 307 redirect response. The client's redirected request is then serviced by the `Web Server`.

1. A request comes in from a client to the Load Balancer.
2. The `Load Balancer` then redirects the request to least busy worker.An HTTP 307 redirect response is sent to the client, directing to the chosen `Web Server`
3. The redirected request comes in to the Web Server.
4. The `Web Server` services the request, from Memcache if possible else get the data from one of the slave/master database.

Meanwhile, the balancer is in its own loop, receiving the messages being sent to it.

Site configuration



Goals/Deliverables

Plan to Achieve

- Implement a parallel web server that implements all basic parts of the HTTP protocol in a high-performance fashion.
- It should be capable of supporting many concurrent requests.
- Investigate what kind of speedup would be expected on a large number of concurrent requests and attempt to achieve this. We will look into what kind of performance we should be expecting and then incrementally improve our code until we have an acceptable speedup.
- Prepare a demonstration of our work. We will host some page or pages, possibly this page, and allow people to make requests to it on their computer or ours.

Hope to Achieve

- Caching functionality.
- A fully functional website (dynamic/database), where anyone can post and top posts are displayed.

Platform

We would use MPI to implement this server in a high-performance parallel fashion. We would simply run it on one of the four core lab machines. We would use C++, MySQL and a caching library (possibly memcache).