PROJECT REPORT

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 (Scalability)

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.

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.

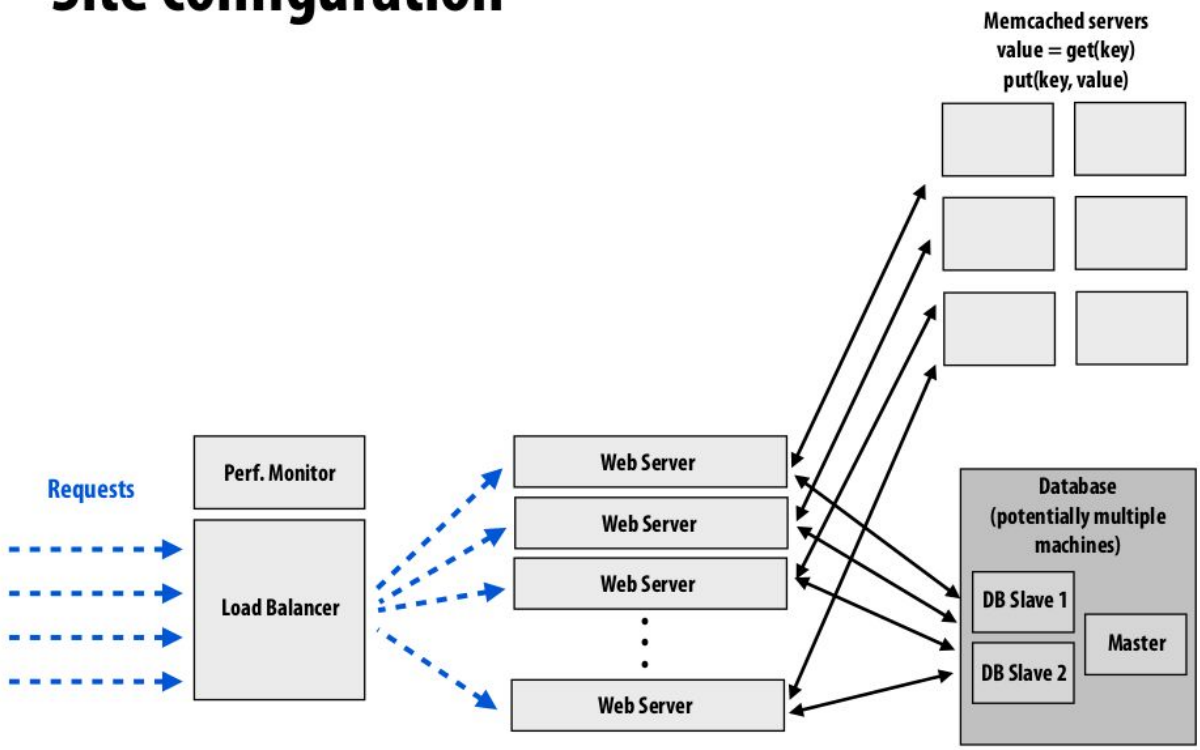
Approach

MPI was overhead, Since our requests will be random distributing the requests in round robin scheduling was best option, and each server is able to handle requests parallell. Our system use different processes, called `Load Balancer`, and `Web Server`. 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 parallely.

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

Site configuration



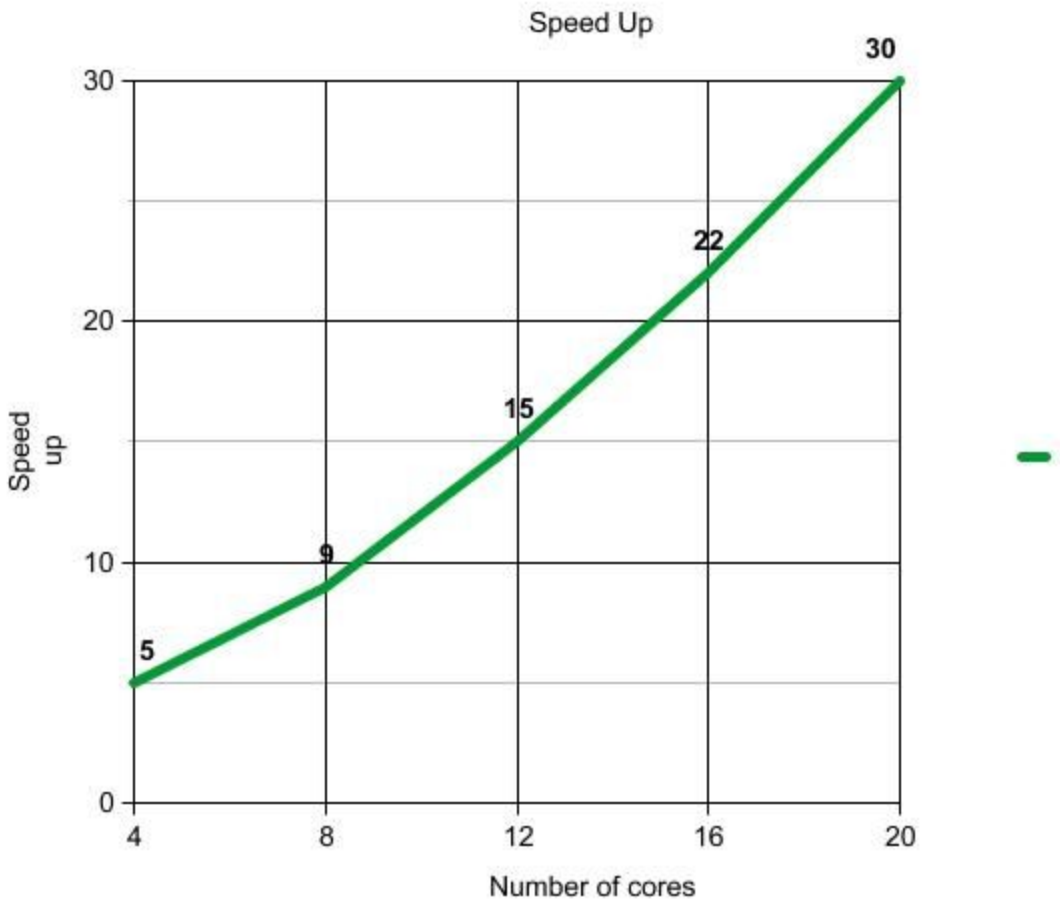
Goals Achieved/Delivered

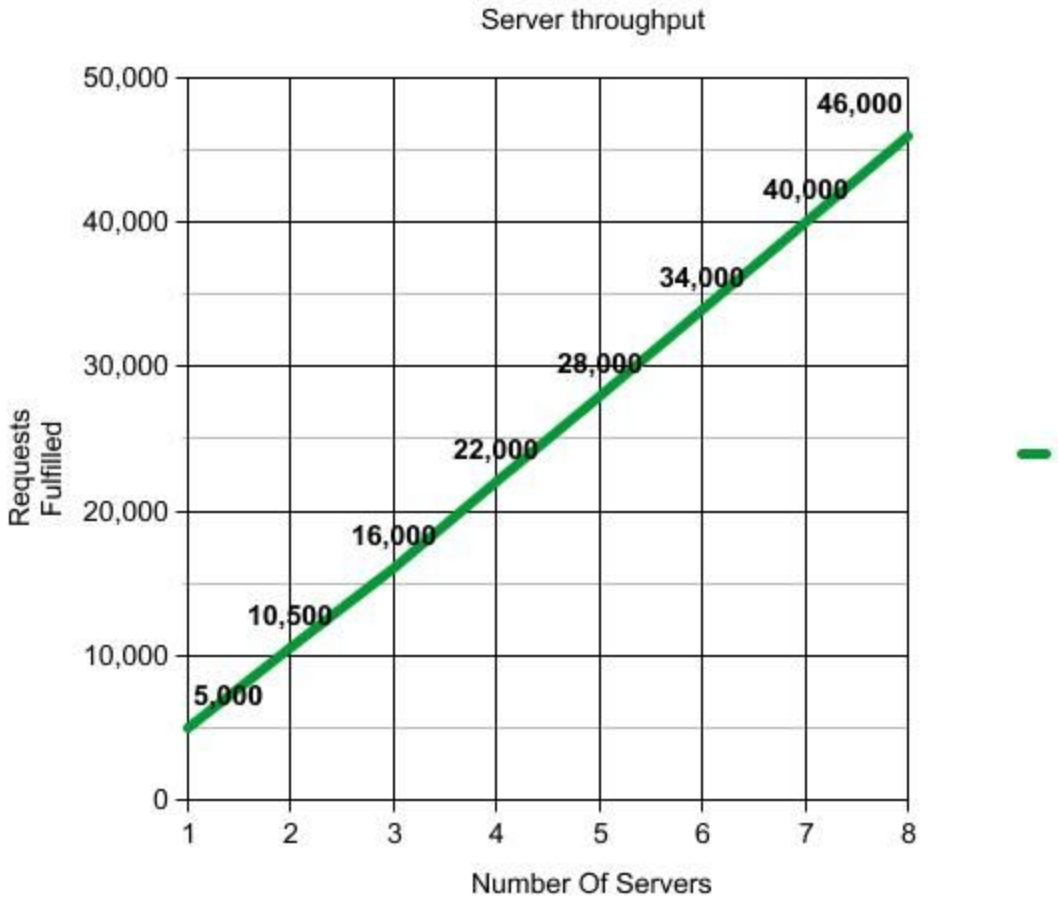
Achieved

- Implemented a parallel web server that implements all basic parts of the HTTP protocol.
- It is capable of supporting many concurrent requests.
- Investigated what kind of speedup would be expected on a large number of concurrent requests and attempt to achieve this.
- Prepared a demonstration of our work. We hosted pages, and allowed people to make requests to it on their computer or ours.

Results

To handle testing, we wrote an automatic HTTP request generator. It requests file from different servers parallelly and sequentially. Since we want to measure how our server deals with load, it makes sense to stress the server over a given amount of time, using number of requests fulfilled as the performance metric. Each computer was having 4 cores.





Immediately important things to note:

- We achieved linear speedup.
- No elasticity configuration, workload achieved much better performance at 8 workers than with elasticity on.

We were quite happy to achieve linear speedup in the standard configuration. Looking carefully at the graph, we can see that the total speedup is actually superlinear. Since requests are fully data-parallel, this makes sense, since the only inherently sequential part of our algorithm is in the dispatcher, which has a relatively unintensive job.

One possible reason for super-linear speedup in low-level computations is the cache effect resulting from the different memory hierarchies of a modern computer: in parallel computing, not only do the numbers of processors change, but so does the size of accumulated caches from different processors. With the larger accumulated cache size,

more or even all of the working set can fit into caches and the memory access time reduces dramatically, which causes the extra speedup in addition to that from the actual computation.[4]

- WIKIPEDIA

Even though the system scales linearly, performance can surely be improved. Better scheduling than Round Robin is one option. Changing elasticity thresholds could also have improvements. An additional idea that we wanted to implement was caching.

Distributing over multiple servers is an excellent way to scale a web server, due to the data-parallel nature of HTTP requests. Bottlenecking occurs only at the dispatcher, but for particularly large websites, using multiple dispatchers in different geographic areas is always an option(the way that content distribution network works). Elasticity is also great in order to save costs.

Github-
https://github.com/ramanb25/Server