

A Queuing Simulation of a Modern Web System

George Mason University
Systems Engineering & Operations Research
Prof. John Shortle
Spring 2010

Ryan J. O'Neil
roneil1@gmu.edu
ryan.oneil@washingtonpost.com

Part 1: Literature Survey

Problem Statement

In the mid 1990s, the Internet and World Wide Web became available for use by the general public. While the Internet supports a variety of protocols and applications, the Web quickly rose to dominate all others in terms of information sharing and prevalence in common use. Since that time, web traffic has grown explosively as Internet connectivity has become generally available.

Due to the number of users on the Internet, as well as the commonplace use of bots for scanning content, it is not uncommon for web servers to experience high sustained volume or random traffic spikes. For instance, a site like Google can expect to experience high traffic all the time. News sites, on the other hand, can expect abnormally high volume on important news days, such as September 11, 2001 or the 2008 Presidential Election.

Despite the job of scaling a web site for high traffic being a frequently experienced growing pain, there seems to be a paucity of modeling and optimization applied to it in the web development community. The software that serve web content are essentially queuing systems, so there can be benefits from applying queuing theory to this problem.

Web Server Structure

A standard web server relies on three primary components, each of which is critical to its performance: the hardware on which it runs, the server software that processes incoming web requests, and the server Internet connection itself. The hardware is important because it determines the upper bounds of resources such as CPU, memory, and disk access speed available to the web server. The software is responsible for maintaining request queues and actually serving files. Finally, the Internet bandwidth determines how much data can be sent out to customers per unit of time.

When a web browser, or customer, requests a web page, that request first enters the accept queue of the web server. It stays in this queue until one of the worker processes of the web server becomes available to service it [1]. The worker process is then responsible for loading the files requested by this customer and passing those files to it over the network, at which point the connection is terminated. The number of workers and length of the accept queue are specified in the web server configuration [5]. Once the accept queue fills up, the server rejects incoming requests.

There are typically several files a customer must download in order to display an individual page. Once a customer finishes its wait in the accept queue, it downloads multiple files from its assigned worker synchronously, keeping the network connection open until all files are served to avoid having the client wait in the accept queue again [1]. Total response time for

a page from the server perspective is equal to the time spent in the accept queue plus the time spent downloading each file. Thus serving a page is equivalent to serving a small burst of files. Figure 1 shows the typical process a web server executes when handling a request.

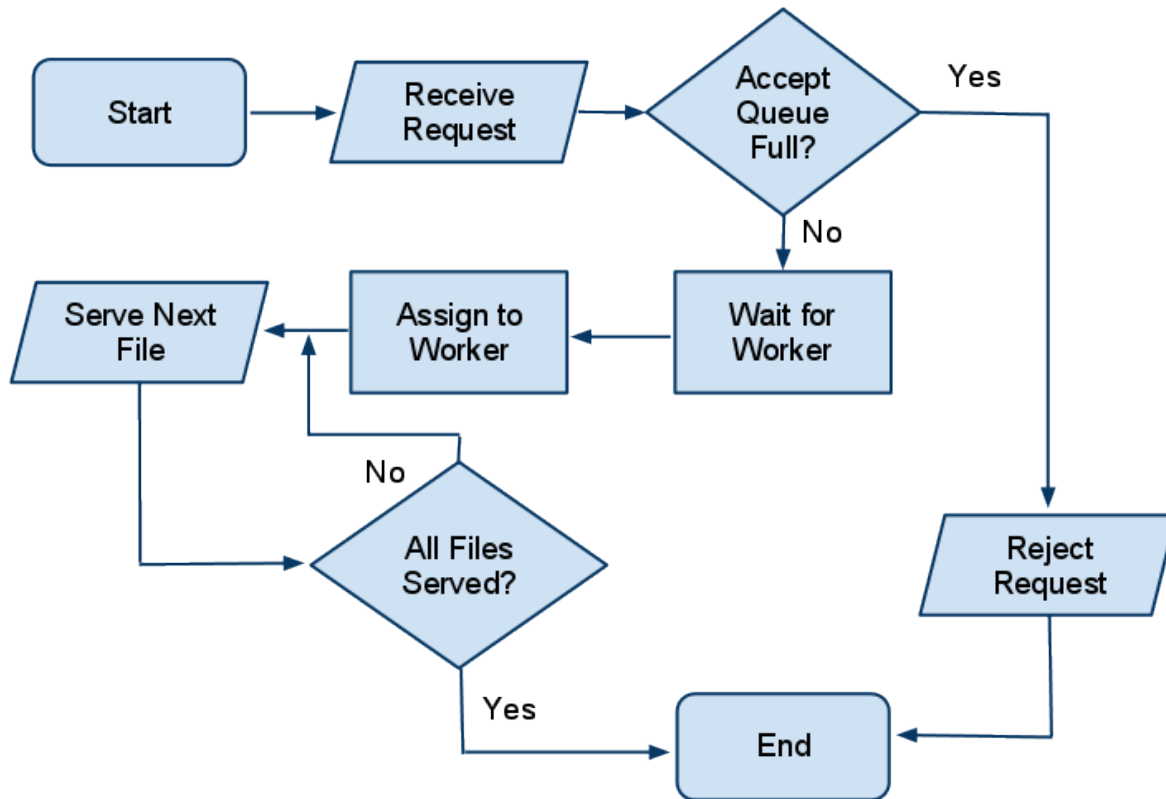


Figure 1: Process for a typical web request

There are multiple worker processes handling requests in parallel within a web server. These processes each require access to the same resources such as CPU, memory, or network bandwidth, which tend to be much more limited [3]. Meanwhile, there are a potentially unlimited number of customers which can make an unlimited number of requests to the server.

Modeling Web Servers

At its most basic functional level, a web server is a tool for allowing users to read remote files through a common queue. For simplicity, we ignore the effects of serving dynamic content, which requires running custom programs of unknown nature, such as Common Gateway Interface (CGI) script. Allowing these in the model would greatly increase its complexity, so we consider only the action of serving files to incoming requests. Elleithy and Komaralingam (2002) discuss how an exponential distribution provides a conservative approximation of file sizes, providing a reasonable upper bound on response time.

While these servers handle requests from multiple customers in parallel, the resources needed to service those requests, processor time, network bandwidth, memory, etc., are available in small or limited quantities [3]. Most web servers process requests with a fixed number of worker processes in a first-come-first-served discipline. As a result, queuing models can make good abstractions of the behavior of web servers and general results for queues are also applicable to web servers.

Admitting a queuing model as an abstraction of a web server implies one can collect and calculate standard queuing metrics. In this case, we are particularly interested in server response time, which is the same as expected wait in the system. Server utilization is an important related metric. Elleithy and Komaralingam (2002) show that, just as in a basic queuing model, when this quantity is at zero, no customers have to wait in the accept queue. As it approaches one, response time grows to infinity.

A major issue with using a queuing model to approximate a web server is that real web traffic does not tend to follow a well behaved probability distribution. Instead, traffic tends to arrive in bursts or spikes which are correlated with externalities such as times of day, content cycles, or news events. These traffic bursts are of particular importance because they can easily exceed the maximum capacity of a web server, filling its accept queue to the point that it denies incoming requests [1].

This behavior implies that it is essential for any model of web server performance be verified with testing against a live server. Liu et al. (1993) find that queuing models "consistently overestimate response times," possibly due to the assumption of exponential inter-arrival and service times. However, they also see that queuing models track performance characteristics of web servers fairly well. That is, they make good indicators of whether changes to a server configuration will improve or worsen its performance.

The queues used to model web servers differ depending on the type of service studied and the goals of a given author's analysis. An obvious model for web servers that have an accept queue of length K and c worker processes is an $M/M/c/K$ queue. This model's assumption of exponential inter-arrival times is likely valid for normal traffic, but will probably not apply during traffic bursts, and we have already seen that service time may be overestimated by an exponential distribution. The number of servers, c , and the maximum number of requests in the accept queue, K , are set in the web server configuration. Elleithy and Komaralingam (2002) use results from $M/M/c$ and $M/M/1$ queues to explain general results about web server behavior, while Sha et al. (2002) use $M/M/1$ and $G/G/c$ queue results. Liu et al. (1993) model their server on $M/M/c/K/N$ queues where N represents the number of sessions active in the web server.

Web Server Performance Characteristics

Web server response time has been studied under the assumption that sizes of files served are exponentially distributed. Elleithy and Komaralingam (2002) observe that server response time changes very little as load, or arrival rate, increases. This continues up to an easily identifiable point where the response time suddenly grew asymptotically. They describe this as a "clear upper bound" on the server load taking on a "hockey stick" appearance [2].

The same authors tested a number of factors to determine what inputs response time was most sensitive to. Among these, the network buffer size for transferring chunks of files over the Internet and the client bandwidth did not appear to affect response time much at all. The critical components of the system appear to be:

1. Server network bandwidth. This was a common culprit of performance problems during the 1990s when high volume sites were operating on T1s. With the larger pipes of today, it's less of an issue but is still important to monitor. This is closely related to the size of files transferred, which is why web servers commonly spend the time to compress files before sending them to customers.
2. Web server capacity. This only becomes an issue under very heavy load assuming the server network bandwidth is high enough to handle the given volume of traffic. Two values that determine this are the number of worker processes and the maximum length of the accept queue. Limiting factors on these values are available CPU and available memory.
3. Average file size and the number of files required for a page. Response time increases linearly with file size, up to the point where it becomes unbounded [2]. As web sites have become larger and more complex, this has turned into a bigger issue, replacing network bandwidth as a common culprit of performance problems. Setting proper values for the number of workers and length of the accept queue can have a major impact on server performance.

With regards to the unpredictable nature of web traffic, it is important to note that, due to the hockey stick nature of response time, a web server operating near capacity is very easy to push over capacity. This is particularly likely to happen if requests arrive in bursts [1]. Banga et al. (1997) demonstrate that a short burst of six times the average rate of requests can degrade overall server capacity by 12-20%.

Empirical Testing of Web Servers

Since queuing models track web server performance well, but may provide poor predictors of actual performance metrics, it is of vital importance that one run empirical tests to determine

performance of given configurations [1]. Unfortunately, this is a difficult problem in itself.

According to Banga et al. (1997), Web traffic has a potentially unlimited number of clients and tends to arrive in bursts of eight or ten times the average arrival rate. TCP limits the amount of traffic from each machine, so to properly simulate this, one would require a vast amount of hardware and network resources. This is prohibitively expensive. The best most organizations can hope for is a small testing lab of several machines, meaning their testing doesn't properly evaluate the effects of traffic bursts.

Thus, in stress testing a web server, it is possible to reach a state where the bottleneck is not the server, but the network limitations on the client hardware. In cases where the server is handling static content only, this will typically occur before the web server reaches capacity [1]. The ability to recognize this behavior and increase hardware resources until the web server exceeds capacity is useful in evaluating server behavior under traffic spikes.

Optimization of Web Servers

Possibly the most common solution for server administrators to increase web server performance is to scale horizontally by adding server machines and putting them behind a load balancer. One can add horsepower to the web site at the cost of additional administrative overhead and redundancy of resources. This is a very easy solution to performance problems and can work quite well. However, it ignores optimization that could take place within a single server itself by more effective configuration or changing the way it behaves. Worse, multi-server systems are extremely sensitive to mismatched loads. Adding lower power hardware to a load-balanced system may actually slow the entire system down [2].

Discovering a performance problem in a web server is of particular concern because it usually means that the problem is already evident to the world. It is advisable to be proactive and work to improve server performance before one is overtaken with problems. We examine some options from the literature for optimization of a single web server:

Option A: Asynchronous Serving

Web servers handle multiple requests concurrently, and typically do so in a synchronous manner. That means that, once a customer has a connection to its worker process in the server, it remains connected to that worker and that worker remains busy until the customer has received the files it needs.

It is possible to remove the requirement that communications between customer and server be synchronous. This would result in asynchronous handling of requests and leave behind some of the complexity incurred from multi-threading in the server.

Praphamontripong et al. (2006) examine a web server model based on the Proactor pattern in which each operation in the web server is executed independently. Worker processes exist to move requests between queues and to hand off requests to request handlers. After that, they go back into service. Individual asynchronous operations are processed from a common queue in first-come, first-served order by a pool of request handlers.

This method adds an extra level of complexity to the web server model in the form of an extra queue or queues. However, performance is not limited by the number of worker processes as much anymore, and our server will tend to fill up its queues under times of high traffic instead of experiencing runaway response times and deadlock conditions [4]. Unfortunately, this is not functionality that is supported by most mainstream web servers.

Option B: Control Theory Feed-Forward Loop

A second, and similarly invasive, approach uses control theory to regulate the response time of a web server. Given resources, a web server's response time is highly nonlinear in response to a workload that is stochastic and undergoes abrupt changes in magnitude. If it is possible to linearize the response time of the web server, we can achieve more predictable performance [5].

The basic idea in this optimization technique is to add a feed-forward loop into the web server itself. This loop is responsible for monitoring the response time of the web server, and artificially adding a delay to each request to achieve some desired response level. The monitor system attempts to minimize the difference between actual response time and desired response time.

Sha et al. (2002) use this technique as a form of congestion control. However, there are new problems introduced with regards to sampling. How often should one sample the server to get a realistic estimate of response times? By what quantities should the monitor increase or decrease the artificial delay? These are potentially as difficult to answer as the configuration of the web server itself.

Option C: Online Regulation of the Number of Servers

Another method examined in the literature involves doing live updates to the MaxClients value of the Apache web server. This technique has the advantage that it is not as invasive as the others and it can be implemented against a running system.

In this case, Apache is a commonly used web server and MaxClients is its configuration parameter that specifies the number of worker processes. One can think of this value as the number of servers, or c in an $M/M/c$ queue. This represents the number of requests a web server can perform in parallel. Server performance is extremely sensitive to the number of worker processes [3].

Having either too few or too many servers reduces performance dramatically, possibly by an order of magnitude. If there are too few servers, requests wait in the accept queue or get rejected by the server. Too many servers cause resource overuse, resulting in deadlocks. Since traffic intensity changes over time, we would like to find the optimal number of servers for whatever our current level of traffic is [3].

There is an open issue here of how to set the number of servers. Liu et al. (1993) examine using Newton's method versus a fuzzy method and a heuristic. It appears the heuristic method works well in practice, and consists of increasing the number of servers until the computer's processor is saturated and no further. At this point, further parallelism is not helpful to the system. This method has the advantage of being simple and achieving fast convergence to the response time minimum.

Conclusion

Web servers represent an important method for transmission of information in our society. We have seen an explosive growth in traffic during that last decade and a half and we can expect to see that growth continue for the foreseeable future.

Web servers operate as queues with multiple servers that require access to shared resources. Performance of a web server, measured in response time, tends to become worse very gradually up to a point where there server reaches capacity and response times grow to infinity.

Most optimization of web server systems is done on an ad-hoc basis by server administrators. Considering web servers as queuing models is a useful method for predicting performance improvements, but requires empirical validation. Several methods do exist for on-line optimization of web servers, but it does not appear such techniques have been implemented in major web servers at this point.

Part 2: Modeling a Modern Web System

Problem Statement

The models discussed in Part 1 all assume that a web server under investigation serves static files which reside on its file system. While this may be the case for a fair amount of the content on the Internet, it is an unacceptable assumption when it comes to sites that serve dynamic content. Such sites exhibit a much higher degree of randomness as they contain more moving parts which can interact in unpredictable ways. For instance, they usually require a database with a number of tables of differing complexity and size. Dynamic pages may require the web server to make a number of different queries to its database, selecting, updating, or creating new data. It is even possible for the size of the content served to grow over time as it is exposed to traffic. Similarly, dynamic pages typically require software code to power them, which can introduce unknown layers of complexity and bottlenecks to the system, such as deadlocks.

Of particular interest to this study is a framework that recently emerged for rapid development of database-driven web applications called Django [6]. Django is a model-view-controller framework similar to Ruby on Rails, which targets the Python language. The Washington Post is one of notable users of this system and serves many of its databases of editorial content to the public through a Django powered site. Due to the unpredictable nature of traffic for a high-volume news site, it is extremely important to the newsroom at the Post that this system be reliable and scale quickly to large volumes of traffic.

The Django installations at the Post currently host approximately 600 tables of data distributed among nearly 90 distinct applications. Adoption of the technology is growing internally as the newsroom seeks to differentiate itself from its competitors through innovative use of data and technology. The developers who use this platform and the administrators who are responsible for it seek to understand its performance profile better, including how it will react to changes in operating conditions. One of the questions they would like to answer is: once their existing system reaches its capacity, is it possible to simply add more hardware to achieve better performance or will that require intervention and improvements on the part of the programmers? In other words, who is responsible for the continued health of the installation?

Web System Structure

Most Django installations have much in common with other database-driven web application servers. At the Post, there are two identical application servers which run Apache and Django. These sit behind a load balancer which assigns incoming requests to one of them in a round-robin fashion. They share a single database, from which they request data via SQL queries over an internal network. Requests that come to an Apache server through the load balancer are handed off to Django for processing, which is responsible for communicating

with the database and constructing the page content to return.

Thus far this resembles a fairly standard web system. The one thing that may stand out as unusual is the way in which it uses caching. Each web server has an associated cache of pages which resides in memory. When a page is requested, Django first checks its cache to see if the page has already been generated. If it has, no further processing is necessary. Django retrieves the page from memory and hands it back to the user. If not, Django must retrieve data from the database and construct the page to return to the user. This content is then put in cache for use the next time that page is requested. The percentage of time a requested page is found in cache is known as the cache-hit ratio.

Pages can stay in cache for a maximum of 24 hours before they are regenerated. Pages are only put into the cache when requested by users. Further, when journalists in the newsroom change data in the database, pages associated with those records may be purposely forced out of cache, to be regenerated the next time they are requested. In this manner Django seeks to achieve high performance without sacrificing the dynamic nature of content. Pages are considered static if the content they serve has not changed recently. Pages are stored in memory, which is faster than storing them on the hard drive. Frequently requested pages are more likely to be in the cache, rendering the system more equipped to deal with traffic spikes that occur in the news cycle. Figure 2 shows a high-level structure of this web system. There is a queue in this system for each web server, as well as one for the database.

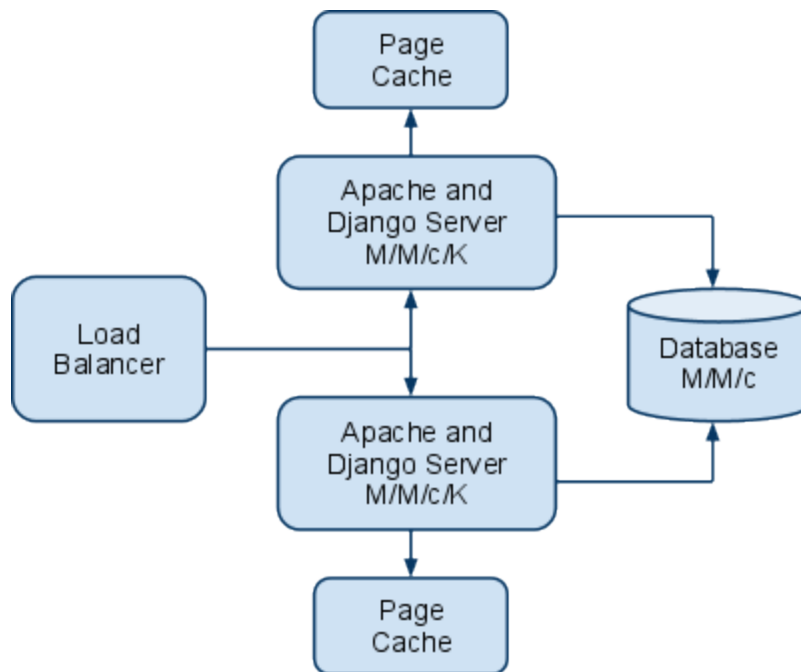


Figure 2: High-level structure of a modern web system

Queuing Model

This system can be reasonably approximated using an open queuing network. There are two web servers and incoming requests alternate between them in a round-robin fashion. As in the models in Part 1, these web servers exhibit exponential inter-arrival times, have a set number of worker processes c , and have a maximum length for their accept queues of K . The values are set in the Apache configuration. Each incoming request has a probability that the content it needs is found in cache, in which case the service time for that request is exponential with a fairly quick mean service time. If the content for a request is not found in cache, then that request enters another queue for processing by the database. The service time in this case is also exponential, with a much larger mean.

The process flow for an incoming request can be expressed as follows:

1. Assign the request to the next web server.
2. If that web server's accept queue is full, reject the request and stop. Otherwise keep the request in the server's accept queue until it is picked up by a worker process.
3. Test the assigned web server's cache for the requested page. If it is found, return that page and stop.
4. Handle the request through the database queue and return the resulting page. Store this page in cache for the next time it is requested.

Due to the limits on the length of the web server accept queues and the fact that web server assignment by the load balancer is not simply probabilistic, metrics for this system cannot be modeled using an open Jackson network. This paper uses a discrete event simulation to study the system's performance characteristics.

Model Assumptions

A number of simplifying assumptions must be made to bring the model to a manageable level. First, it is assumed that the two web servers are indeed identical and that any additional servers added in the future will also be identical. The load balancer is assumed to lack a queue of any sort, and to immediately hand off requests to the web servers without any measurable delay. Similarly, communication between the web servers and their caches are local to the web server and without overhead or queues. The caches themselves do not run out of memory and are fully described by the exponential service times and cache-hit ratio. Further, the cache-hit ratio is stationary, implying that large sets of pages do not fall out of cache all at once as would happen if the caching system were restarted. This last assumption is realistic for general news cycles, but may not apply to events such as national elections where data for many pages arrives simultaneously.

It is assumed that incoming requests are fairly constant and do not arrive in bursts. This is an acceptable assumption since traffic spikes could be simulated by simply increasing the inter-arrival rate of the system. The system modeled here will have the following initial configuration and environment:

- 2 identical web servers
- 25 workers per web server
- 25 workers in the database
- accept queue length of 100 per server
- cache-hit ratio of 75%
- rate serving pages from cache of 10 requests/second for each server
- rate serving pages from the database of 1/2 requests/second

Model Output & Analysis

In order to determine throughput for the system, it was simulated in its initial configuration using inter-arrival rates of 20 through 70 in increments of 5. Each configuration was simulated 50 times for 100 seconds, constituting several thousand arrivals.

The programmers and administrators of this system would like to know whether it is more important to improve their cache-hit ratio to keep performance acceptable, or if they can just ask the administrators to add hardware. In other words, how responsible are the programmers for the continued performance of this system? There are two conditions of interest here which were simulated in both positive and negative changes: the addition and removal of a single identical web server to and from the system by administrators, and the increasing and decreasing of the cache-hit ratio by 5% by clever programming.

Figure 3 shows the system throughput for each configuration as it achieves a steady state by increasing the arrival rate from 20 to 70 arrivals per second. There is a marked decrease in throughput by removing a web server, but adding an additional server does not provide any performance gain over the initial configuration. This is likely due to the bottleneck already being on the database side when there are two servers. On the other hand, a 5% drop in cache hits achieves roughly the same result as the removal of one server, while a 5% increase in cache hits yields a marked increase in system throughput. This makes sense, given that a higher cache-hit ratio removes traffic from the database server.

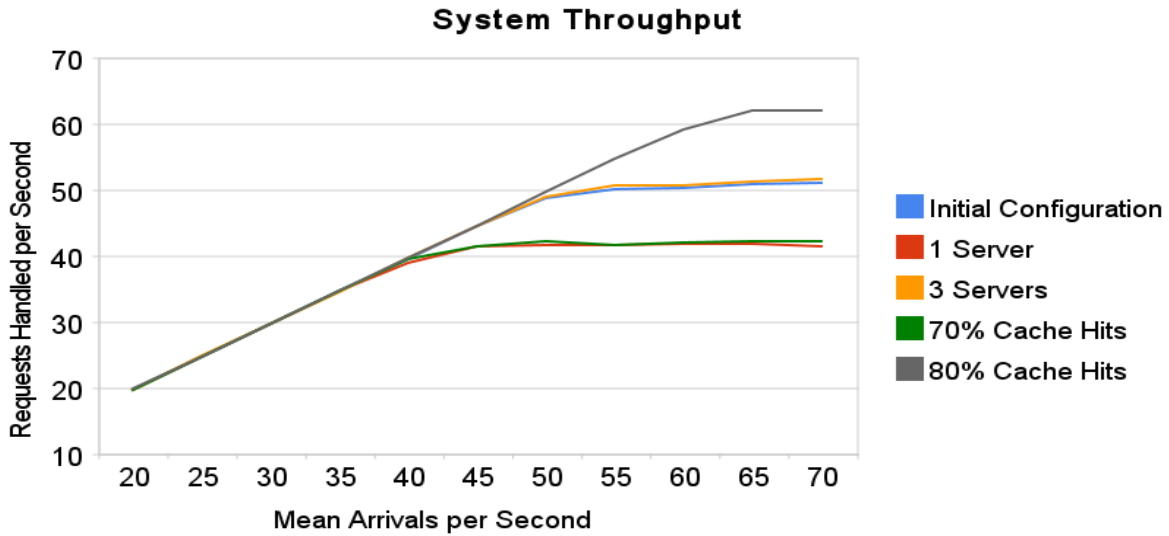


Figure 3: Mean throughput for each configuration

Figure 4 shows mean wait time in the queue for requests using the same configurations. Wait times for changes in cache-hit ratio show a similar trend to throughput: lower cache-hit ratio translates into higher wait times and higher cache-hit ratio decreases wait time significantly compared to the initial configuration. Changes to the number of servers are not so straightforward. It is interesting to observe that one server actually appears to give a better average wait time, while three servers make the result worse. This is deceptive, since it is likely due to more requests being rejected by the accept queue when there are fewer servers. Thus fewer requests are waiting on other requests in the database queue.

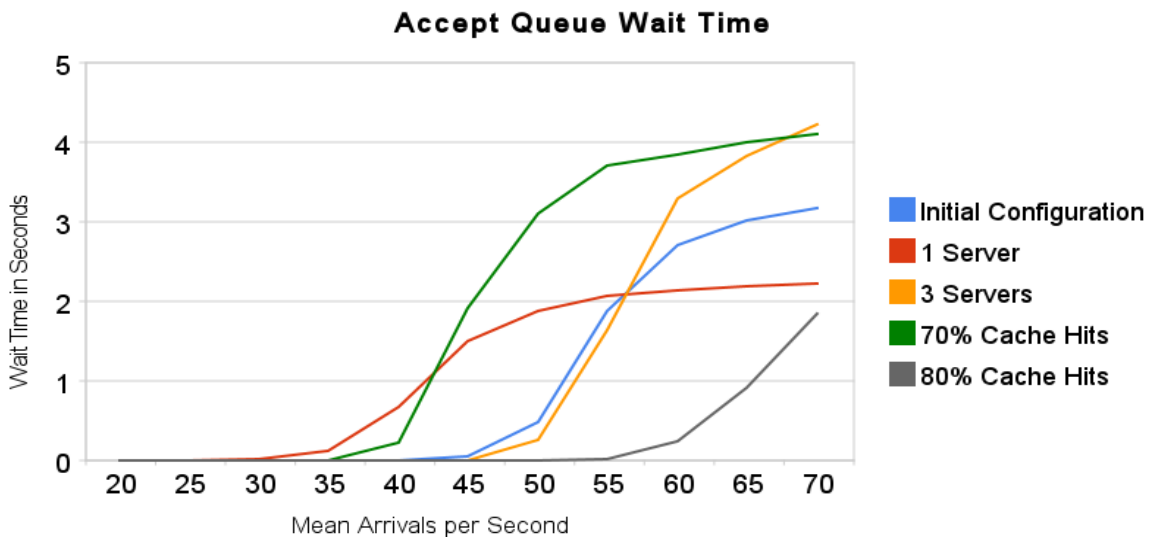


Figure 4: Mean wait time in the accept queue for each configuration

Figure 5 exhibits the time it takes to serve a request once it has been through the accept queue and been assigned to a worker process. Again, cache hit ratio causes exactly the changes one would expect, improving mean service time as it increases. Conversely, the number of servers affects service time in the reverse: adding servers actually increases time to serve a request. This should be for the same reason as above.

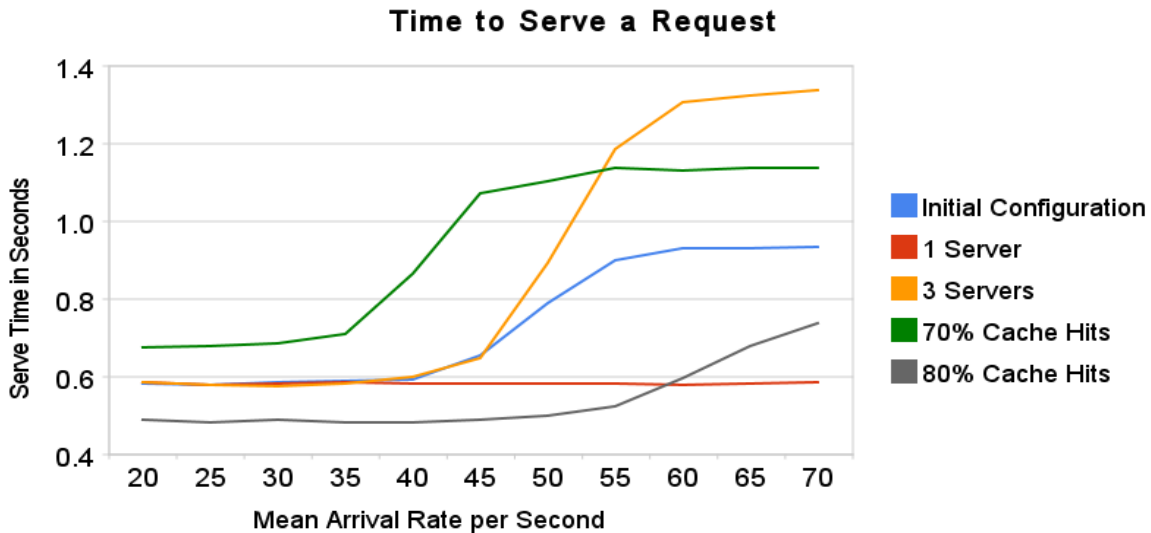


Figure 5: Mean time to serve a request for each configuration

Conclusion

There is well established conventional wisdom that performance problems can be solved by adding more hardware to a system. Even the Django documentation hints at the ability to scale horizontally by addition of web servers and distributed databases. This study provides some evidence that the situation is more complicated and suffers from exposure to differences in the queuing capabilities of combined web and database servers.

Of particular note are results for wait time and service time where adding hardware makes the system perform worse and removing hardware makes it perform better. These are counterintuitive and illustrates the unpredictable effects of queuing networks on server systems.

On the other hand, changes in the cache-hit ratio tend to have outcomes that exactly mirror one's expectations. These appear to be more predictable and, at times, have more impact. This implies that, at least for the system modeled here, performance is better served by improvements of the programmers who build the software systems. Administrators should add new hardware only once the capacity of the most constrained queue in the system is able to handle additional load, otherwise they risk making the system perform worse.

Bibliography

1. Banga, Gaurav, Druschel, Peter. "Measuring the Capacity of a Web Server." 1997.
2. Elleithy, Khaled M., Komaralingam, Anantha. "Using a Queuing Model to Analyze the Performance of Web Servers." 2002.
3. Liu, Xue, et al. "Online Response Time Optimization of Apache Web Server." *Lecture Notes in Computer Science*. 1993; 153+.
4. Praphamontripong, U., et al. "Performance Analysis of an Asynchronous Web Server." *Proceedings of the 30th Annual International Conference on Computer Software and Applications*. 2006.
5. Sha, Liu, et al. "Queueing Model Based Network Server Performance Control." *IEEE Real-Time Systems Symposium*. 2002.
6. Django. <http://www.djangoproject.com/>.

Appendix: Simulation Code

The simulation built for this paper uses SimPy 2.0.1 and was run on Python 2.6.2. It performs all logic that a request must go through between entering and leaving the system in the *Request.handle* method of the *Request* process. If a request is assigned to a server with a full accept queue, it exits the system. Servers and worker processes exist as resources which a request instance may have to wait for, as does the database resource. The *RequestGenerator* process is responsible for putting requests into the system with exponentially distributed inter-arrival times. The *LoadBalancer* class creates server resources and provides an iterator which the *RequestGenerator* uses to assign requests to servers in a round-robin fashion.

```
#!/usr/bin/env python
from SimPy.Simulation import *
from itertools import cycle
import csv, random, sys

SERVERS = 2 # number of web servers
WORKERS_PER_SERVER = 25 # worker processes per web server
ACCEPT_QUEUE_LENGTH = 100 # maximum number of waiting requests
ARRIVAL_RATE_MIN = 20 # min arrival rate of web requests
ARRIVAL_RATE_MAX = 70 # max arrival rate of web requests
ARRIVAL_RATE_INC = 5 # arrival rate increment
CACHE_HIT_RATIO = 0.75 # fraction of requests served from cache
CACHE_SERVE_RATE = 10 # rate for serving requests from cache
DB_SERVE_RATE = 0.5 # rate for serving data to the web server
DB_CONNECTIONS = 25 # number of allowed connections to the database
SIMULATION_STOP = 100 # time to stop simulation
SIMULATION_RUNS = 50 # number of simulation runs for point estimator

class LoadBalancer(object):
    """
    This class represents a load balancer with a round-robin policy.
    Calling assign() on an instance returns the next server resource
    that a request will be assigned to.
    """
    def __init__(self):
        servers = [Resource(
            capacity = WORKERS_PER_SERVER,
            name = 'web server %d' % i,
```

```

        unitName = 'worker'
    ) for i in xrange(SERVERS)]
    self.servers = cycle(servers)

def assign(self):
    return self.servers.next()

class Request(Process):
    """
    Represents a web request. The handle method manages the process
    execution model for an arbitrary web request.
    """
    def __init__(self, name, server, database, wait_monitor, serve_monitor):
        super(Request, self).__init__(name=name)
        self.server = server
        self.database = database
        self.wait_monitor = wait_monitor
        self.serve_monitor = serve_monitor

    def handle(self):
        # If the accept queue is full, this request is rejected
        if len(self.server.waitQ) < ACCEPT_QUEUE_LENGTH:
            arrival_time = now()

            # Get assigned to a worker process
            yield request, self, self.server
            start_time = now()
            self.wait_monitor.observe(start_time - arrival_time)

            # If the requested page isn't cached, incur a database hit.
            if random.random() >= CACHE_HIT_RATIO:
                yield request, self, self.database
                yield hold, self, random.expovariate(DB_SERVE_RATE)
                yield release, self, self.database

            # Always incur the cache serve rate. This is the time used
            # to simply load and serve requested content.
            yield hold, self, random.expovariate(CACHE_SERVE_RATE)

            # Release the worker process
            self.serve_monitor.observe(now() - start_time)

```

```

        yield release, self, self.server

class RequestGenerator(Process):
    '''Generates incoming web requests with exponential inter-arrival times'''
    def __init__(self, name, arrival_rate, load_balancer):
        super(RequestGenerator, self).__init__(name=name)
        self.arrival_rate = arrival_rate
        self.load_balancer = load_balancer
        self.database = Resource(
            capacity = DB_CONNECTIONS,
            name = 'database',
            unitName = 'connection'
        )

        # Monitors for data collection
        self.wait_monitor = Monitor()
        self.serve_monitor = Monitor()

    def generate(self):
        i = 0
        while True:
            yield hold, self, random.expovariate(self.arrival_rate)
            r = Request(
                'request %d' % i,
                self.load_balancer.assign(),
                self.database,
                self.wait_monitor,
                self.serve_monitor
            )
            activate(r, r.handle())
            i += 1

if __name__ == '__main__':
    out = csv.writer(sys.stdout, delimiter='\t')
    out.writerow(['arrival rate', 'mean throughput', 'mean wait time', 'mean serve time'])

    # Run our simulation for a variety of arrival rates
    for arrival_rate in xrange(
        ARRIVAL_RATE_MIN,
        ARRIVAL_RATE_MAX + ARRIVAL_RATE_INC,

```

```
ARRIVAL_RATE_INC
):
throughput = []
wait_times = []
serve_time = []

for i in xrange(SIMULATION_RUNS):
    initialize()

    gen = RequestGenerator('req generator', arrival_rate, LoadBalancer())
    activate(gen, gen.generate())

    simulate(until=SIMULATION_STOP)
    through = gen.serve_monitor.count() / float(SIMULATION_STOP)
    throughput.append(through)

    wait_times.append(gen.wait_monitor.mean())
    serve_time.append(gen.serve_monitor.mean())

# Take final means as the means of means
mean_through = sum(throughput) / len(throughput)
mean_wait = sum(wait_times) / len(wait_times)
mean_serve = sum(serve_time) / len(serve_time)

out.writerow([arrival_rate, mean_through, mean_wait, mean_serve])
```