Notes on Scaling Up Web Applications - CSE 408222 – Cloud Computing

Caution: These are high level notes that I use to organize my lecture. You may find them useful for reviewing main points, but they aren't a substitute for participating in class.

Scaling Up a Web Service to the Cloud

A traditional single-machine web application consists of these main components

WWW – A web server that processes HTTP requests.
DB – A relational database that provides transactional storage of structure data.
FS – A hierarchical filesystem that stores large static objects.
CODE – Web page code which generates dynamic content from FS and DB.

This setup will work fine for a modest amount of traffic, but as the number of clients and transactions grows, performance will suffer. It is possible to buy larger machines with more memory, cores, etc. but costs grow exponentially and a limit will be reached.

Question: What are the possible bottlenecks in the basic system?

This architecture can be incrementally converted into a scalable cloud application. The existing components can be kept with (minor) changes, and improvements can be made as load demands.

Scaling Up by Steps

Step 1: Put static files in an S3 bucket.

Code must be modified to direct all file accesses to S3 servers.
For web apps dominated by large file accesses, this may be all that is necessary!
(Question: What sort of web applications would benefit from this?)

Step 2: Use CloudFront to perform content distribution for static files.

Using S3 only still causes the client to make multiple requests to at least two servers that are geographically distributed. Register a CloudFront URL with your S3 bucket, then modify code to direct all accesses to CloudFront instead.

Step 3: Use Elastic Load Balancing to run multiple WWW servers.

If the bottleneck lies in processing many client requests, then use ELB to run a large number of WWW processes. ELB also does some health checks on the servers in order to avoid sending clients to bad servers.

Step 4: Use ElasticCache (memcached) to cache accesses between WWW and DB.

If the bottleneck lies in accessing data in the DB repeatedly, DB queries can be cached. Memcached deployed on each node (or in a separate ElastiCache cluster) can offload much of the traffic from a single DB instance.

Step 5: Scale up the database itself.

Approach 1: Run multiple relational databases, each one responsible for a separate "shard" of the data.

Approach 2: Use an inherently scalable OLTP database, like DynamoDB or Hbase. (Which effectively performs the sharding for you.)

More Detail on Memcached

Basic Idea: Memcached provides the infrastructure for distributed caching, but leaves the semantics and the consistency entirely up to the client (e.g. the web developer) and so it must be used with care.

Interface to a single standalone mcd server:

put(key,value): mcd stores the key and value, possibly evicting other items

get(key) -> (value): mcd returns the value if available, but not guaranteed

Web server code must be rewritten as follows:

Old code:

```
result = dbquery(sql);
```

Becomes:

```
value = get(sql)
if(value and value is up-to-date) return value;
result = dbquery(sql)
put(sql,result);
return result;
```

This is all very easy to do, but the client must be careful with two questions:

1 – How do I know that the result is up-to-date?

2 – Which mcd server should I query?

Possible mcd configurations:

1 mcd for the whole system (simple, but doesn't scale)

1 independent mcd for each www servers (swasteful and doesn't provide locality)

N cooperative mcd servers (high capacity, but must be used carefully)

Cooperative Caching Techniques

Suppose you have many clients, N cooperating mcd servers, and one shared database. How should we distribute puts/gets across the mcd servers most efficiently?

Simple idea:

h = hashfunc(key);

n = h % N;

do that put/get at server n

Effectively, all requests will be spread across the mcd servers evenly, which is great for load balancing. Data items will not be duplicated, so there is no horizontal consistency problem. This is one big **cooperative cache**.

Problem: What happens when the number of servers change? Functionally, it will still work, because mcd is just a cache, and a cache miss still results in a correct result. But, for a time, there will be a large number of cache misses until the system stabilizes.

Can we modify the strategy so that a change in the number of servers does not result in a large amount of cache churn? This is called **consistent hashing**.

Idea 1: Let n = N*h/MAX. This will result in each server storing a continuous stretch of the hash number line. Now, sketch what happens when you go from 2 to 3 to 4 servers. We see that about 1/2 of the total amount of data will churn between nodes. Good idea, but not a great improvement.

Idea 2: For one server, assign the entire hash space. For each newly added server, divide the largest chunk in two, and give the new server half of the space. In this way, every server addition/removal only involves churning the cache of two nodes.

Question: In the worst case, how much cache space will go unused for idea #2?