ESE 605 Homework 3

Stephen Phillips

April 27, 2015

Problem 1

Part a

How would a recursive DNS name resolution of the URL "http://www.cis.upenn.edu/ cis505/" travel, from the servers root, edu, upenn, cis and such?

I am assuming no caching. First you would go to your local DNS server sending a request for the URL. The local DNS server would ask the root server for the location of the URL. The root would in turn send a request to the "edu" server, which would make a request to the "upenn" server, which would make a request to the "cis" server, which would return the URL, and it would go back up the chain to the "upenn" server, "edu" server, the root, back to the local DNS server, and finally back to you. Then you could request for the " cis505" resource.

Part b

Recusive DNS name resolution makes it simpler for the user - they only have to worry about sending a request to one server. This also allows greater caching, since if lots of people are querying the same URL (e.g. Google, Amazon, Facebook), they can connect them in one step, saving lots of time. Iterative is not so simple for the user since they have to send requests to each of the heirarchical name servers.

The disadvantage is that if that server crashes they the user will lose all the work that server has done. Also the servers have to maintain states for every request they get, which could be huge. Iterative only has to respond once and the user takes care of the rest. Essentially it flips the burden of the work from the server to the user. With this is also greater security since the user doesn't have to exclusively trust one server.

Problem 2

Part a

A file is replicated on 10 servers. List all the combinations of read/write quorum that are permitted by the voting algorithm. We have N nodes. Let the number of nodes voting for a read quorum be R and for a write quorum W. The only conditions we need to satisfy are:

1. $W + R > N$, for read-write consistency
2. $W > N/2$, for write-write consistency

That gives us these options (when N is 10): ($W = 10$, $R = 1$), ($W = 9$, $R = 2$), ($W = 8$, $R = 3$), ($W = 7$, $R = 4$), ($W = 6$, $R = 5$). If you just want read-write consistency, we disregard the second condition, and can get these options as well: ($W = 5$, $R = 6$), ($W = 4$, $R = 7$), ($W = 3$, $R = 8$), ($W = 2$, $R = 9$), ($W = 1$, $R = 10$).

And there are other technically valid but less efficient options when $W + R > 11$, so we won't list them here.

Part b

In the two-phase commit protocol, why can blocking never be completely eliminated, even when the participants elect a new coordinator.

Well, if the newly elected leader crashes, and then they elect a new leader, which crashes again, and so forth, then we still will have a blocking issue. This is unlikely but still theoretically possible.

Part c

Receiver-based message logging is generally considered better than sender-based logging. Why?

If you use reciever-based logging, if a node crashes, it can recover from the logs of the messages it has recieved to a more temporally recent place in the system. With sender-based logging, this would not be possible. We also have more certainty of recieving a message than of sending one, since we have no guaratees if the message made it to the destination.

Part d

Taking into account cache coherence, which kind of cache-coherence does NFS implement?

The cache of NFS uses two levels, disk of the client and main memory of the client. The cache is flushed when the file is closed. It can keep the file in cache after the file is closed, but it needs to validate the file if it opens it again.

Part e

NFS implements the remote access model to file handling. It can be argued that it also supports the upload/download model. Explain why.

This could be argued since each client caches the file into their own disk (download) and then sends the file back to the NFS server to update it (upload) - this is done due to cache, though it was not officially in the standard until version 4.

Problem 3

We want to compare the trade-offs made by the Coda File System (Coda) and the Google File System (GFS).

GFS is designed for massive amounts of throughput - the whole internet is being stored through it. Coda was designed for large scalability and availability - for high use in disconnected operation like mobile clients. Therefore their designs are different. GFS is optimized for large data (e.g. append only writes, 64MB chunks, etc.), and does not concern itself with latency. It assumes failures are the norm, but expects very reliable data transfer (since it expects everything to be on a local network).

Coda, on the other hand, wants low latency and high availability so it uses weak transactions so that multiplie users can have access to a file, and uses conflict merging to resolve the differences. This also makes it more robust to network partitions and users going offline, something GFS doesn't have to worry about. It uses caching in this way as well to increase availability - invalidations are pushed by the server to all the users using that file to invalidate the cache. This also increases availability since if the server goes down, the users can still use the file. Again, Coda focuses heavily on low latency and high availability.