Cpsc 4660 Project Report

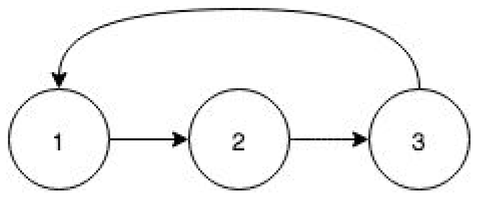
Lukas Grasse

001172543

November 29, 2015

For this 4660 project, I implemented a similar system to the one outlined in "*EDB: a multi-master database for liquid multi-device software"* published in *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (MOBILESoft '15). In this paper, the system described is a distributed, peer-to-peer database system that enables software on multiple devices to sync with one another whenever available, but to still operate for long periods of time while disconnected from the network. This network also functions without any central server controlling everything, and each node in the network contains a copy of all the other node's data from the last time they were in sync. The system works by having each node maintain a log of all the changes it has had happen to its database. Then other nodes use the log of changes to perform the changes to themselves as well. The second component of the project is a web app that makes use of the replication component to be able to create and sync notes from node to node, similar to notepad or document apps that can sync from a smartphone to tablet to computer and other devices.

Although the implementation of this project is similar to the EDB system described before, there are a few differences I made in implementation compared to the original system. In the original system there was a *Platform Adaptation Layer* that both the replicator and api were built upon, however in my implementation the api is also the adaptation layer. This means that whenever a note is added, removed, or updated using the api, the api also logs the change. Then when a sync is initiated, the replication component uses this log to perform the replication. Another difference between the original paper and my implementation is that my implementation only supports a node having one direct peer. This means that in order to add as many nodes as desired, they must be chained together. For example, consider three nodes as shown below:



In this case, node 1's peer is node 2, and node 2 syncs with node 3, and node 3 back to node 1. In this way an unlimited number of nodes can be added to the network, and yet every node only needs to sync with its closest peer, which keeps the nodes from performing lots of checks to make sure they are updated with all the other nodes. Although this provides the advantage of less tracking required it has a few disadvantages. The first disadvantage is that the network propagates lots of log changes throughout the network every time changes happen. For example, if 2 updates an element, 1 sees this as a change. It then also performs the update, and this change is picked up by 3. Then node 2 picks up this change and it continues to propagate forever. This is a really bad problem because it means the database ends up syncing every record in the database every time it syncs with another node. There are two ways to fix this problem. The EDB system from the paper solves this by having every node keep track of the logs of every other node. I decided to solve this a different way however. My system uses the timestamp of the original action, such as insert, update, or delete, instead of the timestamp of the sync action. This means that for any action, it will propagate through the network once, but the sync action will only propagate through the network where it has not been seen yet, and then if the timestamp has not increased, the log entry for the sync will be ignored. This prevents the sync from propagating through the network indefinitely.

I implemented the replication component described above using node.js. Each node.js server also has its own local MongoDB database. This database is responsible for storing a copy of the application's data locally, and all indexes and queries the application makes on the data are queried from the local database. Inside the database, there are three separate collections. The first one is Note. A Note object has three properties: and Id, a Title, and a Note. The id is generated by the local database, and the title and note are both strings, with the note being the body of the note. A possible improvement of this setup would be to have each note id created as a UUID, or universally unique identifier, to prevent collisions with note ids on other servers.

Currently the id’s are not generated this way so collisions are more likely, although I did not experience a collision in my testing. The next collection is called Log. A log item contains 6 different properties. The first is a log id which is a sequential id, so that the Log collection forms a list of changes. The second property is the id of the object being logged. The next property is the eventType which can be either ‘Insert’, ‘Update’, or ‘Sync’. Another property is timestamp, which is the timestamp of when the event occurred. Also, a version number which indicates the version of element, for when an element is updated. Finally we have a random number which is used to break ties when the timestamp and version are both tied. The third collection in the database is called SyncLog, and it’s schema only has two properties: serverId and maxLogId. The serverId is the id of the server that has been synced with, and maxLogId is the largest log that has been synced with the serverId server.

To start a new node on the network the node.js server takes 3 parameters: the address/port that the server runs on, the address/port of the server it syncs with, and the address/port of its local database. The notes application that makes use of the replication servers is a web app that is also served by the node.js servers. The application makes use of an api that is exposed by the replicator server. The main routes are:

- get '/': returns html of all the notes currently in the database, as well as a form to add new notes to the app.
- get '/docs': a route that returns all documents in the database in json format.
- get '/log/:id': a route that returns all logs greater than the id parameter in the url. Also in json format.
- get '/viewLog': returns entire log formatted in html.
- get '/log': returns entire log in json format.
- get '/doc/:id': returns the note specified by the id in url parameter as json document.
- post '/doc': inserts new note into database. Takes note in json format as post data.

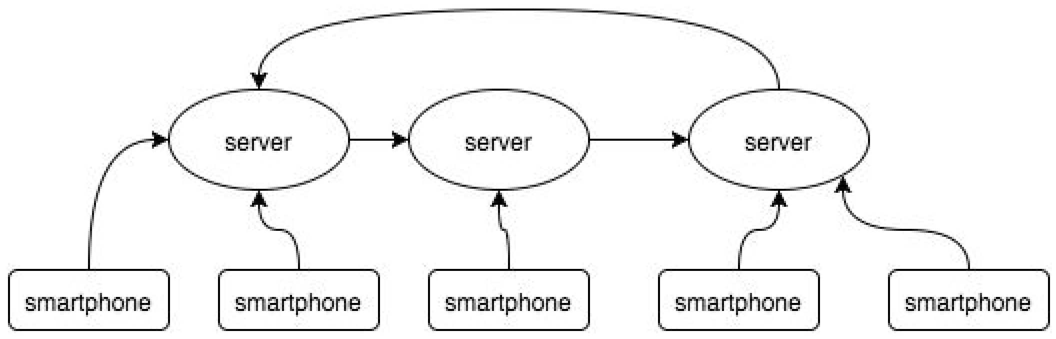
- post '/updateDoc': Updates note in database. Takes note in json format as post data.
- delete '/doc/:id': deletes the note specified by the id url parameter from the database.
- get '/sync': syncs the node with its peer node.

There are a few benefits from designing the server in this way. The first benefit is that other apps and services can make use of the api to interact with the replicator server from other devices and browsers. So for example, someone could write a mobile app that uses the replicator as a server to store its note data. Another benefit of this setup is that it is very simple to adapt the server to documents of different types. The routes that interact with the notes in the database use the generic name doc, and changing the note schema is all that would be required to create a replicator server that handles different types of documents.

The evaluation method I originally described in the proposal consisted of three components. An evaluation of the correct replication from database to database, the effect of a replication on the performance of mobile devices, and how the regularity of replication affects performance.

The first evaluation works correctly, with the records being replicated from server to server correctly. One of the issues I discovered when testing this was the previously described issue of sync logs being continuously propagated throughout the network. Also, it was interesting to observe that it is possible to delete all logs and notes from a server, and it can still recover them based on the sync logs of other servers. This means if a node on the network was accidentally destroyed, the log contains the information needed to restore the node.

The second evaluation component was not tested because the project pivoted from the original goal of running on mobile devices to running on servers. This means testing the performance of battery life and limited network connections were not able to be performed. However, having the replication happen server to server opens up some interesting opportunities for incorporating mobile devices. One of these is the ability to create a peer-to-peer network of servers, and have a smartphone connect and sync to the geographically closest node in the network. This network could look like below:



This is an architecture that makes more sense for implementation, because it is easy to create servers that have a static ip, but a smartphone's ip is changing all the time as the user moves around and connects to different wifi networks. I simulated a smartphone node by creating a leaf node that did not have a node syncing from it, and tried throttling the network connection. I did not implement a smartphone app in this project, but it would be an interesting next step to take.

The third aspect of evaluation was how the rate of syncing affected performance of the network, and this aspect is very important considering each node only has one direct peer in the final network. This means changes from one node to another are only propagated to another when all nodes in between them have also synced. This is a severe disadvantage caused by the choice to only sync with one other node per node. Since the main use case of this network from the paper the project is based on is a single user sharing documents across all their devices, this might make sense since the user could click the sync button whenever they desired. However, it is probably a good idea for the application using the network to call sync every so often to make sure the local database is up to date. With notes, there does not appear to be much overhead with performing a sync, but if an application stored lots of images or other large resources this infrastructure might need to be modified.

This project taught me some very interesting things about distributed systems. The first is that there is a balance that has to be decided between the density of the graph of connected nodes and the performance the network should have. This means the tradeoff is between performance and reliability of the network. In my implementation I chose a very sparse network with each node syncing with only one other neighbour, but this was shown to have some severe disadvantages. A more reliable solution would be more expensive and also harder to implement. The other thing I learned from this was project was that it while it is easy to sync servers with each other, it is tricky to do the same thing with mobile devices. Another observation is that when a system is distributed like this it would be very hard if not impossible to enforce any of the ACID properties of traditional databases. This project worked because the main use case was a single user on multiple devices, and issues such as race conditions are not that big of an issue with a notes application. However, this kind of network would have some serious issues if trying to sync with multiple users or usage for data that needs ACID properties. All in all, this project was an interesting investigation into some of the benefits and downsides to a distributed system.

References

Oskari Koskimies, Johan Wikman, Tapani Mikola, and Antero Taivalsaari. 2015. EDB: a multi-master database for liquid multi-device software. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (MOBILESoft '15). IEEE Press, Piscataway, NJ, USA, 125-128