LAB 2

Jack Sullivan and Ari Kobren

April 4, 2014

1 Main Components

As per the assignment, our architecture is built around three central components: one *DBServer* and two *FrontEndServer*s. These components are responsible for handling most of the requests from all of clients.

To make it easy to interact with these three components, we have also included a *RequestRouter*. This is a single router node that accepts requests from all clients and evenly distributes them. The *RequestRouter* supervises 5 *RequestWorkers* each of whom is arbitrarily assigned a *FrontEndServer*. When the *RequestRouter* receives a standard message, it distributes the message to one of its workers who forwards it to the appropriately (effectively load balancing). The load balancing follows a round-robin task assignment schedule. The *RequestRouter* can also accept *DBWrite* messages from the *CacafonixClient* (see previous submission). When it receives a *DBWrite* message, the router directly forwards the messages to one of the *FrontEndServer*s.

As before, we used an actor model to managed asynchronous concurrency across multiple processes. To keep the various different concerns of the two front-end servers and the database separate from one another, we extended the basic actor model with a *SubclassableActor* that allowed us to implement Leader Election, Clock Synchronization, Vector Clocks, and primary duties of each class separately. The structure allows us to define partial functions on each trait that dispatch only for certain messages. A final class that has all of the desired functionality can then be achieved by mixing all of the traits together. This design also allows us to separately test each component.

Within each of the leader election, clock synchronization, and vector clock subcomponents care has been taken to make their communication as asynchronous as possible to ensure that each component can continue to process reads and writes from clients and cacofonix while elections or synchronizations are in progress.

1.1 Leader Election

We implemented the bully algorithm for leader election. Leader selection is always implicit, and elections are initiated by members. When an actor that implements *Elector* receives a *GetLeader* message, it will either return its leader (if one is exists and is accessible) or initiate an election to determine a new one. In either case when the election is complete the actor that originally sent *GetLeader* (along with the rest of the electorate) will receive a *TheLeader* message. This message carries the id of the newly elected leader. Within the election process, the message passing occurs simply almost exactly according to the pseudocode of the algorithm. In order to resolve requests for all higher and lower ids, all of the electors have a reference to a *Franchise* which store all of the *Elector*s associated with it.

1.2 Clock Synchronization

We used Berkley clock synchronization, using the above election process to select a leader. To participate in clock synchronization a class must implement *SynchedClock*. When a *SynchedClock* method is asked to synch itself it first sends itself a message to determine its leader (*GetLeader*). Once the election is complete, all of the electorate will receive a *TheLeader* message. This message serves as a queue for the *SynchedClock* that is the leader to attempt to sync the clocks. Once the sync is in progress the leader will only accept *TimeSubmission*s from slaves with a matching sync start timestamp. To ensure that the systems do not move out of phase the backend server process attempts to re-sync the clocks every minute.

1.3 Vector Clocks

We implement causally ordered messages using vector clocks. To do this, both front-end servers and the back-end server are able to accept *SendInOrder* messages (each of which wraps a single *payload* message). When a server receives a *SendInOrder* message, the server updates its clock, extracts the payload and wraps it, along with it's own vector clock, in a new *TimedMessage* message. The *TimedMessage* is then broadcast to all of the other servers.

When a server receives a *TimedMessage* it adds that message to its internal message queue. After that, the server begins trying to process all of the messages in its queue. To maintain a causal ordering, the server, s_l, will only process a *TimedMessage*, $\mathcal{M}$, from server, s_r, if

$$\begin{aligned} \forall i \neq r \quad \mathcal{C}[i] &\geq \mathcal{M}[i] \\ \mathcal{C}[r] &= \mathcal{M}[r] - 1 \end{aligned}$$

This means that a server will only process a message if it has received all other messages from that sender and at least all messages that sender has received from everyone else. If the server cannot find a valid message to process, it waits for the next message. In this way, we make the assumption that the servers will all receive all messages (however, the messages may be out of causal order).

To process a message, a server simply unwraps the payload of a *TimedMessage* and processes it normally.

1.4 Previously Built Components

With our modular design, we were able to easily incorporate components (messages and classes) that we built for Lab 1. Specifically, we used (and slightly modified): *EventRoster*, *TeamRoster*, *Event*, *EventMessage*, *CacofonixListener*, *EventSubscription*, *MedalTally*, *TabletClients* and *EventScore*.

2 System Runtime Workflow

We start our system by running two *FrontEndServer*s and the *DBServer* all on different processes. We also start a process for *CacafonixClient* and n *TabletClients*. When the system fully comes up, we run leader election (Section 1.1) and clock synchronization (Section 1.2).

The clients continually send *EventScore* and *MedalTally* requests as in the previous assignment. As before, clients receive responses to their requests and print them out. The difference is now all of these requests are sent to the *RequestRouter* which forwards the requests to the *FrontEndServer*s

who timestamp the messages (i.e. wrap in *TimedMessage* objects). As described above (Section 1.3) the messages are causally ordered using vector clocks. When the *DBServer* receives these requests, it orders them and keeps every hundredth for the raffle. *CacafonixClient* sends updates periodically as in the previous assignment.

3 Design, Bottlenecks and Potential Improvements

Like before, we modularized our design to make debugging easier. This also makes our system more resistant to failures.

That said, with more time, we'd make our system more fault tolerant. Although leader election occurs when the nodes initially come up, we could improve our system by setting other nodes to watch in case some of the nodes go down. Then new nodes could be respawned and leader election/clock synchronization run again.

Additionally, we could improve our system by making new clients join the system and register themselves with the routers. This would complicate things significantly: vector clock causal order would be more complex (to introduce a new node midway through); additionally, there would be more logic (i.e. handshaking to make sure clients were connected properly to the rest of the system. Because there are more independent components in our implementation of Lab 2 (as opposed to Lab 1) this would be a bit more complicate.

Lastly, we'd like to connect more of our components through routers with special routing logic (i.e. round-robin) than with pointers. We noticed that passing messages through routers is much more efficient and failure tolerant than when we simply pass them independently from node to node (as we do in some of our transactions).

One bottleneck in our overall design is that the *FrontEndServer*s are single threaded. This means that they have to reason about their vector clocks and respond to requests on a single thread. We could improve our implementation by multithreading our servers.

4 Results

We ran our code using 5, 10 and 15 clients each with a rate of one request per .01 seconds and measured the min, max and average response times.

NUM CLIENTS	MIN	MAX	AVG
5	0.06s	0.33s	0.19743s
10	0.05s	0.8s	0.46895s
15	0.02s	0.97s	0.54663s

In our second experiment we ran our code using 5 clients with .01, .1 and 0.5 seconds between requests and measured the min, max and average response times.

REQUEST FREQ.	MIN	MAX	AVG
0.01s	0.06s	0.33s	0.19743s
0.1s	0.01s	0.22s	0.03533s
0.5s	0.01s	0.04s	0.01343s

As we observe, adding more clients seems to increase average latency. There is a significant difference in average latency (2.4x) when we increase from 5 to 10 clients. there is less of a relative difference when we increase to 10 clients. Surprisingly, the minimum latency decreases in these runs.

We hypothesize that this could be due to other things occurring on the network while running our tests or some noise related to our leader elections.

Also, we observe that having the clients make requests at higher frequencies severely affects the latency. As we flood the network with more requests, things drastically slow down. When we cut our between request wait time by 5 (from .5s to .1s) we experience a 3x average latency increase (3x slower to get a response). When we again reduce our wait time by a factor of 10 (from .1s to .01s) we see a 5.5x slow down.

5 Software

Like Lab 1, we've built our system on top of the Akka *actors* library. This library provides a hierarchical message passing architecture for the Scala programming language.

6 How to Run

Instructions on how to run are found in the README file in the project root.

7 Testing

We unit tested the system by writing code to demonstrate that each actor correctly responded to each of the possible messages it could receive and by checking to ensure delivery between actors. In particular, we tested each of the leader election, clock synchronization and vector clocks separately and have included example test run outputs in the zip file.