CS6650 - FINAL PROJECT REPORT

Chat-Bot Application for stock investment

Contributors : Darshi Shah, Cole Crescas, Siyang Zhang, Pu Liu

USE CASE

Understanding the stock market is extremely difficult for most people and even experts. Experts often debate about what market metrics they may need to analyze in order to achieve maximum returns. There are many fundamentals in metrics that analysts use to predict stock price movement, one of those being, sentiment analysis. Market sentiment refers to the overall attitude or feeling of individual investors about the macroeconomic climate and stock market. Many analysts and traders factor in market sentiment into their decision-making and a market can be described as bearish or bullish. For some investors they would like to track specific stock prices as well as see but other investors are interested in. Our chatbot application would allow investors to get stock prices while also seeing what other clients using this service are interested in.

Investopedias key takeaways on market sentiment [1]:

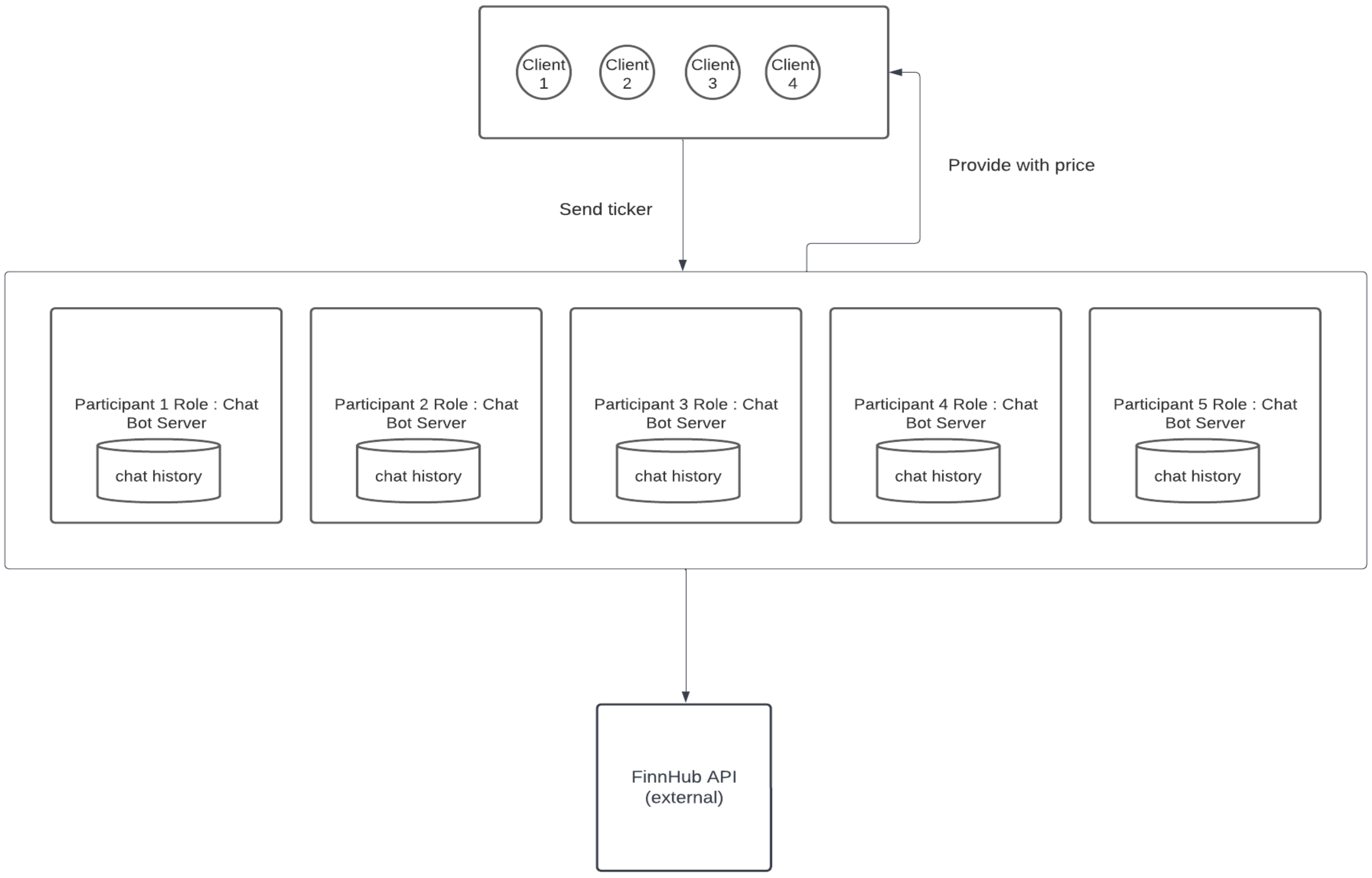
- Market sentiment refers to the overall consensus about a stock or the stock market as a whole.
- Market sentiment is bullish when prices are rising.
- Market sentiment is bearish when prices are falling.
- Technical indicators can help investors measure market sentiment.

Investors can be in multiple & moving locations while still looking for information about their stocks, tracking chat history, and giving other users chat history with regards to time. We wanted to create an application that would serve these needs. Our application can handle multiple clients from multiple different servers while being fault tolerant, mutually exclusive in committing new data, and robust. We achieve this by using what we've learned in this class; specifically, the paxos algorithm for achieving consensus algorithm, leader election, group communication via multicast, total ordering for handling proposal ids within paxos, replication & fault tolerance.

SYSTEM ARCHITECTURE

The PAXOS algorithm works by sending “prepare” messages to all servers through a simple for loop and keeping a counter of a passed flag indicating a server is ready to accept. The selection process is done if the proposalId is equal or greater than another proposalId seen. We used timestamps created when the message was sent. Once a majority of acceptor nodes accept, the key and value are passed to the learner where the commit into the hashmap occurs. Once the commit occurs a message is sent back to the client terminal outlining what actions were taken by the server. The client can then commit to the hashmap a key with the timestamp and a string message of the ticker name and price.

System Overview


Client 1
Client 2
Client 3
Client 4
Send ticker
Provide with price
Participant 1 Role : Chat Bot Server
chat history
Participant 2 Role : Chat Bot Server
chat history
Participant 3 Role : Chat Bot Server
chat history
Participant 4 Role : Chat Bot Server
chat history
Participant 5 Role : Chat Bot Server
chat history
FinnHub API (external)

This diagram shows a series of clients interacting with the participants/BotServers. The participants have a chatbot server with access to the chat history stored in a hashmap. These participants interact with a FinnHub API which is an external API embedded into the client.

An interaction with the chatbot would look like this:

Client 1: Sends out multicast to see what other chatbot instances are online using group multicast communication (algorithm 6). What is the price of ticker AAPL?

External API: the API would then return with the participant and relay to the client: AAPL price is $170.63.

Paxos run decides to commit and the interaction is added to the chat history, ensuring mutual exclusion by using a synchronized method and concurrent hashmap described as algorithm 4

Participant: would then add <timestamp1, “AAPL price is $170.63”>

Client 2: Sends out multicast to see what other chatbot instances are online using group multicast communication (algorithm 6). What is the price of ticker MSFT?

External API: the API would then return with the participant which will then relay to the client: MSFT price is $340.76

Paxos detects a failed server when the user enters an exit command, the leader election algorithm described below as algorithm 5 then kicks off and a new initiator is proposed and the majority switches to exclude failed server. Since we utilized the principles of replication & availability shown in algorithm 1, the data store is not lost and we can still access the chat history.

Participant: would then add <timestamp2, “MSFT price is $340.76>

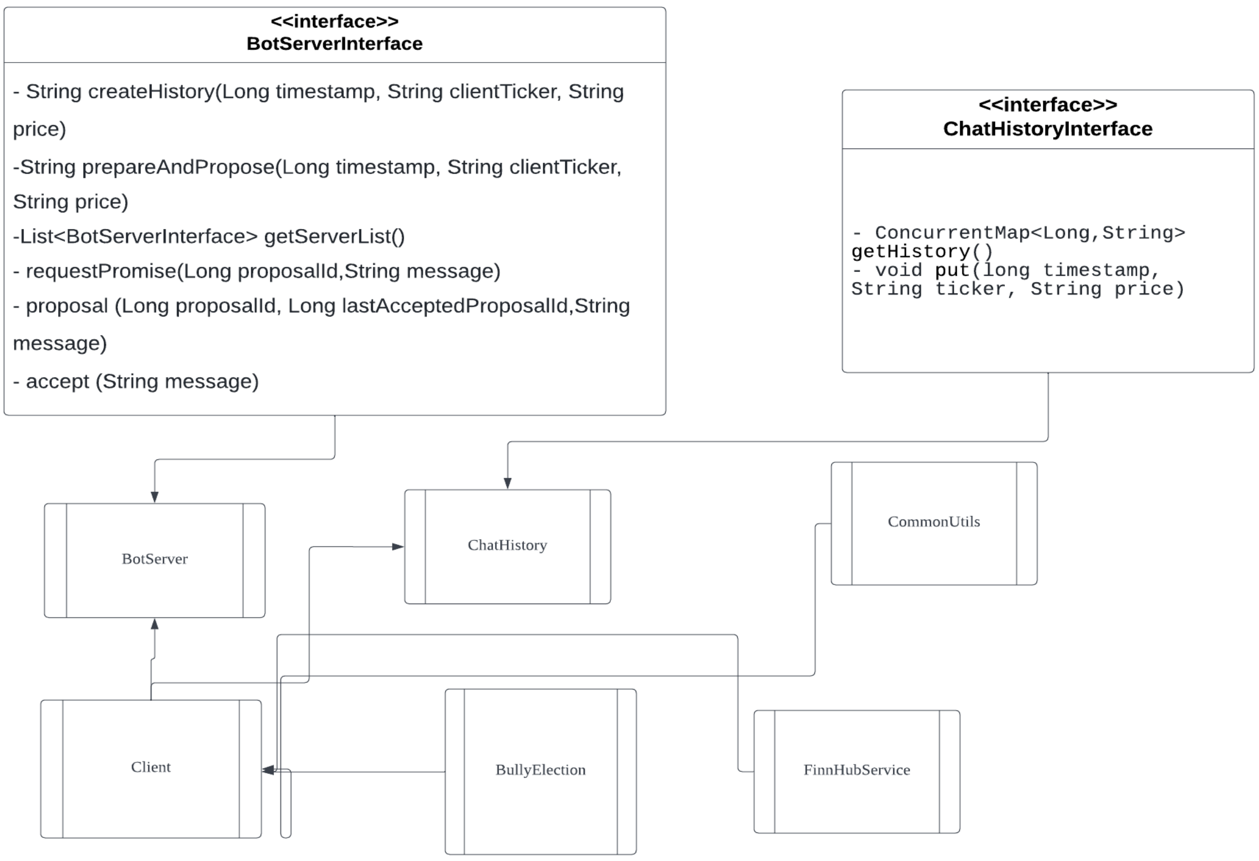
Client 3: Sends out multicast to see what other chatbot instances are online using group multicast communication (algorithm 6)

Client 3: What is the price of TSLA?

Paxos run decides to commit and the value is added to the hashmap, ensuring mutual exclusion by using a synchronized method and concurrent hashmap described as algorithm 4

External API: the API would then return with the participant will relay to the client: TSLA price is $810.23

UML diagram



<<interface>>
BotServerInterface
- String createHistory(Long timestamp, String clientTicker, String price)
-String prepareAndPropose(Long timestamp, String clientTicker, String price)
-List<BotServerInterface> getServerList()
- requestPromise(Long proposalId,String message)
- proposal (Long proposalId, Long lastAcceptedProposalId,String message)
- accept (String message)
<<interface>>
ChatHistoryInterface
- ConcurrentMap<Long,String> getHistory()
- void put(long timestamp, String ticker, String price)
BotServer
ChatHistory
CommonUtils
Client
BullyElection
FinnHubService

EXTERNAL APIS USED

FinnHub Stock API

We are using FinnHub Stock API for stock price quote, which has predictable resource-oriented URLs, accepts form-encoded request bodies, and returns JSON-encoded responses. In our application we use HttpUrlConnection to send the query, then decode the response and return the stock price quote as a String object. The API requires a GET method, a stock symbol query, and a FinnHub Api token in the request query or header. In our design we choose to use authentication via X-Finnhub-Token header.

ALGORITHMS AND TECHNIQUES USED

Following are the algorithms and key concepts we have used in our project from our learnings in this course.

1. Replication & Availability:

We have used the concept of replication to provide a fault tolerant application. In a distributed system data is stored over different computers in a network.

Therefore, we need to make sure that data is readily available for the users. Availability of the data is an important factor often accomplished by data replication. Replication is the practice of keeping several copies of data in different places [2].

2. PAXOS:

As mentioned above, we have provided replication of the single bot server to achieve fault tolerance and whenever replication is done in such a way, there is a need for consensus. It is important to make sure that the chat history at each replica node is consistent even if some replica server fails or becomes faulty. There needs to be an agreement on the common ordering of events across all the replicas of the bot. Since there can be many clients chatting with the bot, it is not practical to have all the replicas agree first and then carry on updating the chat history. Having this immediate consistency would have been too expensive and would cause delays. Therefore, we needed an algorithm that can give us eventual consistency along with satisfying the all-or-nothing property that distributed databases should have to remain consistent. PAXOS is the perfect choice to achieve all these characteristics – atomicity, eventual consistency and fault tolerance.

3. Total ordering concept:

We used the idea behind total ordering to create proposal IDs of the proposals in the PAXOS algorithm. To generate the proposal IDs we thought of using the system time itself. However, we realized that it would only result in partial ordering as it is possible that two proposers propose at the same time. To achieve total ordering, we use the process identifiers (in this case, the port number that the process is hosted on) and concatenate it to the system time(System time in milliseconds + port number = proposal ID) so that even if two proposers propose at the same time they will be ordered.

4. Mutual Exclusion:

We need mutual exclusion to handle situations where a learner is updating the data object for a PAXOS run but then before it finishes, the learner has to make another update as a result of another PAXOS run. If we had used some other programming language than JAVA, we would have to use a mutual exclusion mechanism such as a mutex but for us, to achieve mutual exclusion was a simple task as JAVA provides a keyword "synchronized" and a concurrent version of a hash map (the data structure we use to store the chat history).

5. Leader Election Algorithm: Bully algorithm

Leader election algorithm helps when one proposer from Paxos fails, the application server will run the bully algorithm to select a leader for the client to re-connect. If a proposer server dies, the application will first exclude the failed proposer then randomly select an alive proposer as the initiator to run the bully algorithm; during the bully algorithm, each proposer is assigned with a unique identifier, the proposer with the highest ID will be selected as the new leader for the client to re-connect. Then client could try to reconnect with the new port

6. Group communication: IP Multicast

All clients will have a separate thread functioning as a multicast receiver, which will join a multicast group of 224.0.0.1.At the start of a client, it will join the same multicast group of 224.0.0.1, then send out a multicast packet to that group to announce that it has come online. The receiver thread in other clients will receive the message and subsequently know there is another client going online at which time. This function can be extended in the future for client to client direct communication.

7. Failure Detection:

We detect failures of a server when the user enters an exit command. This is known as a fail-stop since we know that the server has gone down. Using this failure detection as a SocketTimeoutException, we can conclude that a server failed and then initiate the bully leader election algorithm to select a new initiator to run Paxos.

HOW TO RUN

Below are instructions to execute various scenarios in our application.

Scenario 1 – Normal execution

To set up the system, do the following:

- Open console 1 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 1
- Open console 2 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 2
- Open console 3 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 3
- Open console 4 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 4
- Open console 5 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 5
- Open console 6 and type java -jar Client.jar

You will be prompted with a message asking to enter the port number of the server you want to connect to in the Client console. Let's say you enter 32002.

Now, you will be prompted to add a ticker symbol. For the scope of our project, the allowed ticker symbols are AAPL, MSFT and TSLA. If you enter anything else, you will be prompted with an error message and be asked to enter the ticker symbol again. Let's say you enter AAPL.

When you do that, you will be prompted with the current price of the stock immediately and a PAXOS run will start on the servers to maintain consistency of the chat history on each replica. To monitor it you can switch on to console 2 and you will be able to see when phase 1 completes, how many votes were received in phase 1 and phase 2 and finally, you will be able to see the interaction recorded in the chat history on all of the 5 server consoles. (Delays were added at various points so that the PAXOS algorithm executed slowly and you can monitor its execution and introduce messages from another client at any point during the execution of the the first client's PAXOS run)

Simultaneously, you can open another console and start another client and enter another ticker symbol. You will notice that the two PAXOS runs can run together without colliding with each other.

Two client requests will join the same PAXOS run only if the timestamp of the client request is the same (Since the chat history needs to be totally ordered, there should only be one chatbot-interaction data at a single timestamp. So, when two proposers will propose chat data at the same time, one of them will have to propose again depending on when it had entered the ongoing PAXOS run - during phase 1, during phase 2 or if it had entered with a lower proposal ID). It is difficult to demo such a case, but you can check our code to see that it can be handled. You can see that we are using a map data structure to keep the PAXOS runs different on the basis of the interaction's timestamps.

Scenario 2 – Acceptors crashing in phase 1

Now let's try a scenario where the acceptors in a PAXOS algorithm crashes during phase 1 of the PAXOS algorithm. To test this out, you will have to shut down each console by pressing Ctrl+C and then start the system in the following way:

- Open console 1 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 1 crashP1 32001 32002
- Open console 2 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 2 crashP1 32001 32002
- Open console 3 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 3 crashP1 32001 32002
- Open console 4 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 4 crashP1 32001 32002
- Open console 5 and type java -jar BotServer.jar 32001 32003 32003 32004 32005 5 crashP1 32001 32002

- Open console 6 and type java -jar Client.jar

The part “crashP1 32001 32002” means that the acceptors at ports 32001 and 32002 will crash during phase 1 of the PAXOS run. You can enter any two distinct port numbers instead of 32001 and 32002 but make sure they are the same across all replicas.

Follow the same steps as above to interact with the Client. Enter the port number of the propose/server you want to connect to. Suppose you connect to 32001 this time. Enter the ticker symbol AAPL, you will be displayed the ticker’s current price and then go to the console 1 to see how the PAXOS run plays out. You will notice that the acceptor votes in Phase 1 will only be 3 instead of the usual 5 and on the consoles of servers 32001 and 32002 , you will see the message “Acceptor Failed in Phase 1”. In spite of the acceptors failing, you will see that entry was made in the chat history. This is because the majority of the acceptors were still running.

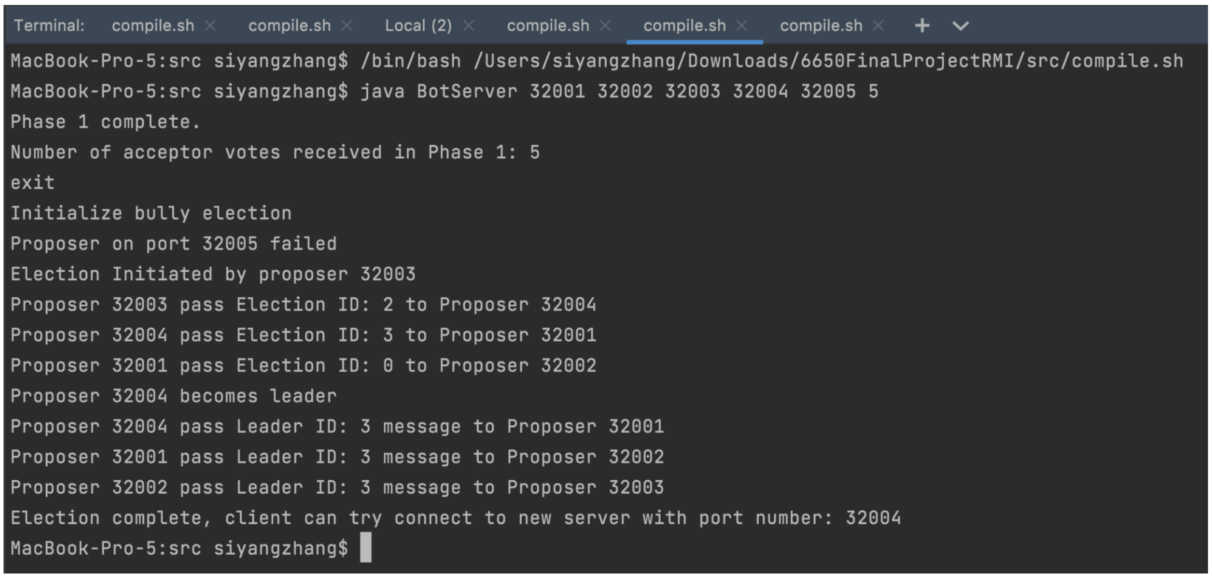
Scenario 3 – Acceptors crashing in phase 2

You can try testing a scenario where acceptors crash in phase 2 by simply shutting down all the consoles by pressing Ctrl+C and then starting the system with the same steps above but this time instead of “crashP1” you should pass “crashP2” in the command. You will notice that the acceptor votes in Phase 2 will only be 3 instead of the usual 5 and on the consoles of servers 32001 and 32002 , you will see the message “Acceptor Failed in Phase 2”.

Scenario 4 – Leader election algorithm

Now, let's test a scenario where the leader election algorithm kicks in. This scenario can be tested along with any of the 3 scenarios above. The leader election algorithm will kick in when a server will crash. You can make a server crash by pressing “exit” in that server’s console. Suppose the client is connected to the server at 32001 and you have entered the ticker symbol AAPL and the PAXOS algorithm is running. Now, enter “exit” on the console of server 32001. You will notice that the server stops and the client is prompted with “Remote call failed”. But before the server stops, it will run the leader election algorithm and suggest the new leader that the client should be connected to. Ideally, we would have wanted the client to connect with the leader automatically however, due to the scope of our project we could not do that part. So, we decided to print all the detailed steps of the leader election algorithm and the elected leader’s port number. You can now make the client (which will be waiting for you to enter the port number) connect to the new leader and carry out the rest of the steps as explained above.

Below is a screenshot that shows that when server 32005 fails, the server with the next largest ID is elected as the leader.


Terminal: compile.sh compile.sh Local (2) compile.sh compile.sh compile.sh

```
MacBook-Pro-5:src siyangzhang$ /bin/bash /Users/siyangzhang/Downloads/6650FinalProjectRMI/src/compile.sh
MacBook-Pro-5:src siyangzhang$ java BotServer 32001 32002 32003 32004 32005 5
Phase 1 complete.
Number of acceptor votes received in Phase 1: 5
exit
Initialize bully election
Proposer on port 32005 failed
Election Initiated by proposer 32003
Proposer 32003 pass Election ID: 2 to Proposer 32004
Proposer 32004 pass Election ID: 3 to Proposer 32001
Proposer 32001 pass Election ID: 0 to Proposer 32002
Proposer 32004 becomes leader
Proposer 32004 pass Leader ID: 3 message to Proposer 32001
Proposer 32001 pass Leader ID: 3 message to Proposer 32002
Proposer 32002 pass Leader ID: 3 message to Proposer 32003
Election complete, client can try connect to new server with port number: 32004
MacBook-Pro-5:src siyangzhang$
```

LIMITATIONS

The following is less of a limitation and more of a design choice that we made. Our system will support only 5 replicas and you can configure failure of two acceptor crashes. We thought this is a good choice as it allowed us to focus on implementing other algorithms.

The bully election algorithm cannot know the status update of other proposers after the initialization, e.g. when another proposer fails after the initialization of failed proposer, other nodes will assume the new failed proposer is still working. A fix should be adding a service tracking the status of alive proposers.

CITATIONS

[1]https://www.investopedia.com/terms/m/marketsentiment.asp#:~:text=What%20is%20Market%20Sentiment%3F,securities%20traded%20in%20that%20market

[2]https://www.geeksforgeeks.org/what-is-replication-in-distributed-system/