Design and Implementation of Distributed Stock Trading App

Group 12: Nandan Nayak, Ramnath Rao Bekal, and Veena Sridhar

University of Stuttgart

1 Introduction

The report explains the design and implementation of a distributed stock trading application based on a client-server architecture that lets users simulate buying and selling stocks across a distributed system. The system features Dynamic host discovery, fault tolerance, Reliable ordered multicast, Heart Beat and leader election mechanisms to ensure robust and efficient operation. Implemented in Python, the application provides a command-line interface that allows clients to place orders and track the status of their transactions.

2 Project Requirements Covered

2.1 Architecture

The distributed system is based on client-server architecture comprising multiple clients and servers that may be deployed across different machines. These nodes communicate with one another through message exchange to achieve the system's objectives. A designated leader among the servers acts as the central coordinator, managing all critical functions, while the remaining servers serve as backup nodes, ready to take over in the event of a leader failure. Figure 1 provides an overview of the system architecture, illustrating the components and their connections necessary for fulfilling the requirements. Client-server connections are established and managed via sockets, with both TCP and UDP protocols employed for message transmission, depending on the specific functional needs. UDP is used for Dynamic Host Discovery, Leader election, Reliable ordered multicast, and heartbeat mechanism. TCP is only used to communicate with clients during trading.

2.2 Dynamic Discovery of Hosts

In the stock exchange application, dynamic discovery of hosts enables new trading client or server terminals to seamlessly integrate into their respective server or client pools without manual intervention.

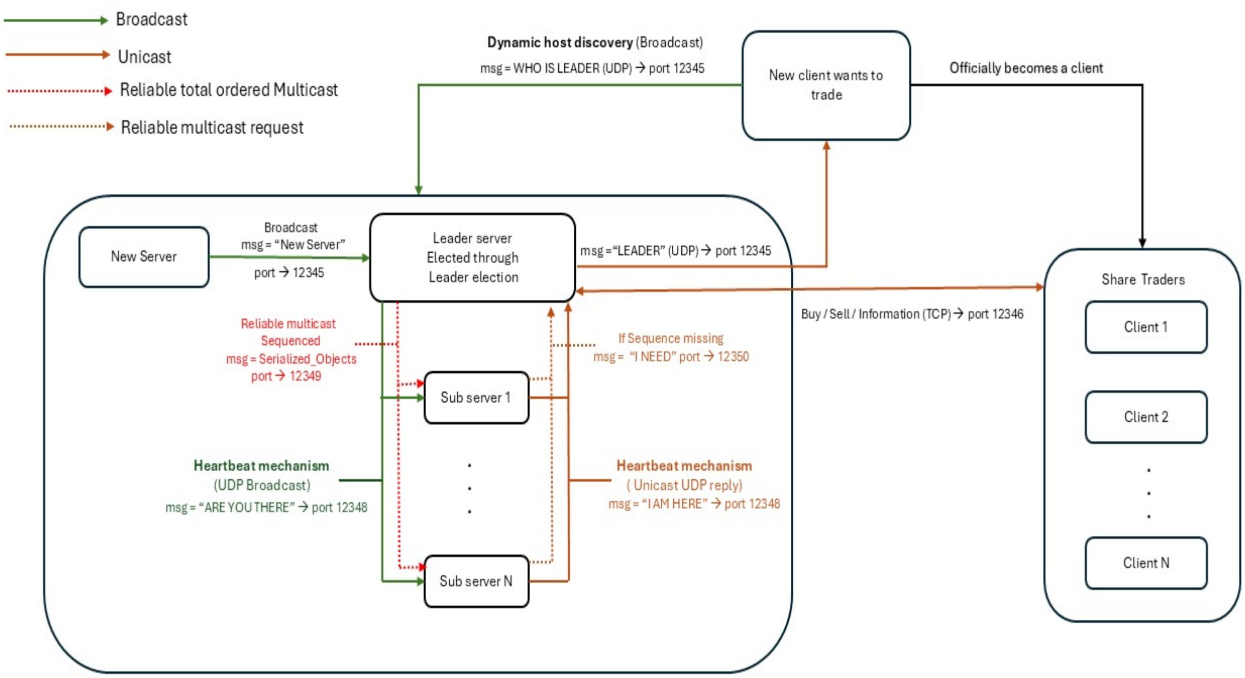
Broadcast
Unicast
Reliable total ordered Multicast
Reliable multicast request
Dynamic host discovery (Broadcast)
msg = WHO IS LEADER (UDP) → port 12345
New client wants to trade
Officially becomes a client
New Server
Broadcast
msg = "New Server"
port → 12345
Leader server
Elected through
Leader election
msg ="LEADER" (UDP) → port 12345
Buy / Sell / Information (TCP) → port 12346
Reliable multicast
Sequenced
msg = Serialized_Objects
port → 12349
If Sequence missing
msg = "I NEED" port → 12350
Sub server 1
Heartbeat mechanism
(UDP Broadcast)
msg = "ARE YOU THERE" → port 12348
Heartbeat mechanism
(Unicast UDP reply)
msg = "I AM HERE" → port 12348
Sub server N
Share Traders
Client 1
Client 2
Client N

Fig. 1. Architecture

Client Join Before a trading client terminal joins the system, it initiates a connection request via UDP broadcast, sending the message 'WHO_IS_LEADER' on a specific port. The leader receives the request. All other servers in the group do not respond to the request and ignore it, as the system join is taken care of by the leader node. Upon receiving the request, the leader responds with its IP address via UDP unicast message, enabling the client to establish a direct TCP connection. With the given IP address, the client requests a TCP connection from the leader server. The leader acknowledges this TCP connection and facilitates further data exchange. The data exchange includes buying, selling, and gathering information requested by the clients to the server.

Server Join Before a new server joins the system, it broadcasts a 'NEW_SERVER' message on the designated port. The leader if already present, updates the server group and the new server joins the system and starts the leader election in a separate thread. The details of the leader election algorithm are discussed in Section 2.3. If no leader is detected (indicating that this is the first server to join the network), the lone server assumes the role of the leader by default.

Server Leave A server node is removed from the system when it is detected as inactive using a heartbeat mechanism. Each server periodically sends heartbeat messages to indicate its active status. If the leader does not receive a heartbeat within a specified timeout, the node is assumed to have failed or left the system. The leader then removes the inactive server from the server group and notifies the remaining servers. If the leader itself fails or disconnects, another server from the pool assumes leadership.

2.3 Leader Election

The system requires a leader among the servers to simplify the coordination and management of features, such as managing the client buy/sell requests, adding new clients/server nodes, ensuring consistency among the servers, maintaining the active servers list etc. If only one server node is available, it will automatically designate itself as the leader. However, when multiple servers exist, the **LeLann-Chang-Roberts (LCR)** algorithm is used to elect a leader. Leader election is triggered in two scenarios: (i) when a new server node joins an existing server pool, (ii) when the current leader node either crashes or gets disconnected. The election process is conducted within a ring topology, where servers are arranged based on their sorted IP addresses. Every server in the system becomes a participant in the ring and share the same group view. The neighboring node of each participant is determined as the node to its left, forming a logical unidirectional ring.

(LCR) Algorithm is used to elect a leader within the unidirectional logical ring. Each node in the ring communicates with its immediate neighbor in a single direction - clockwise direction in this case. The algorithm operates by comparing unique identifiers (UIDs) assigned to each node. During the election process, the nodes forward their UID to the next node in the ring, and only the highest UID continues to circulate. Eventually, the node with the highest UID is elected as the leader and informs all other nodes of its leadership.

2.4 Fault Tolerance

In our stock trading application, every transaction such as buy/sell orders, portfolio updates must be reliably processed and stored to prevent data loss or system downtime. Given the high-stakes nature of trading, failures caused by server crashes or network disruptions are unacceptable. To mitigate these risks, our distributed system is designed to eliminate single points of failure. The fault Detector mechanism to detect whether a node is faulty is implemented by heartbeat mechanism. The leader periodically broadcasts heartbeat messages every 8 seconds to all the nodes in the server group to confirm their availability. Active server nodes respond with a unicast reply to the leader. Conversely, if a server node does not receive a heartbeat from the leader within a specified timeout, it assumes the leader has failed and triggers a new leader election process. This ensures automatic recovery in the event of a leader's failure. To enhance fault tolerance, data replication is implemented across multiple servers in a cluster. We have followed the Passive server(primary backup) model of data replication where a single primary server shares the change in share related data with others. The replication takes place asynchronously, where every transaction is recorded across all servers without time bound. The leader processes incoming orders, logs them, updates the order details, and then multicasts the changes to all other nodes in the server group. By replicating stock trading data across multiple servers, our system ensures that any server in the group can seamlessly take over as the leader if the current leader fails. This approach prevents service interruptions while maintaining data integrity and consistency.

2.5 Reliable Ordered Multicast

UDP multicast is employed in our application to accomplish various tasks in our application for efficiently transmitting data to multiple recipients. We also use it to replicate the data (the order activity) across all servers in the system, but UDP multicast comes with its drawbacks such as No guarantee in the delivery of packets and no built-in ordering of the packets. In order to tackle these drawbacks we ensure total ordered reliable multicast by giving sequence numbers to our multicast messages as necessary. In our stock trading application, the system requires the financial transactions to be executed and stored in the same sequence across all servers to ensure fairness and consistency. Thus it ensures all messages are delivered in the same order to every node in the server group. The leader (sender) assigns a unique sequence number to every message it sends to the server group. Recipients use the sequence numbers to ensure that the messages are received in the correct order. Lost messages can be detected because of gaps in the sequence numbers, allowing the system to request retransmission of the missing message. If a message with a missing sequence number is detected, the latest received message is stored in the hold back queue and it requests the leader for retransmission of the missing messages in its sequence. Once these messages are received, these messages will be delivered, and subsequently, the messages in the hold-back queue will also get delivered. Here delivered refers to the arrival of the messages on the sub-server side. Sequence numbers also prevent duplicate messages from being processed by maintaining a record of already received messages. If the messages with the same sequence number are received, the messages are ignored.

3 Implementation

This section discusses the details of the implementation and the flow of the application. The application is built using a client-server architecture. The clients are the user nodes interested in buying and selling stocks while the servers are the nodes that executes the trades and manage the data. We are utilizing threads to handle the client request, Heartbeat mechanism, Multicast, and Leader election concurrently. We have 2 stocks, ShareA and ShareB, with each having 5000 shares. An example transaction looks like this: Initially client1 connects with the Server and sends a command "client1 b A 100" (The format of this command is described in 3.1). This request is processed by the Leader server and responds to the client that the buy transaction was successful. In the back-end, the Leader updates the current shares under ShareA as 4900 and multicasts the latest state of objects to other servers. Next, let us assume client2 starts a connection with the Leader and sends the command "client2 s A 100" The Leader returns "Transaction failed" because the client2 has not bought any shares yet to make a sell. client1 might be interested to know the status of its purchase of shares. Hence, it will send "client1 i". The Leader responds with the current allocated numbers of shareA and shareB of client1. An important point to note is that we are not considering the price of shares while processing a buy/sell request in this project because we believe bringing the price into the current project would involve additional overhead. We only focus on the buying and selling of stocks irrespective of price.

3.1 Client Program Flow

A client node intending to trade stocks sends a WHO_IS_LEADER message via UDP broadcast to a designated broadcast IP. The client receives a response from the leader server containing the leader's IP address and port. If no leader is found, the client closes the socket and exits. The client establishes a TCP connection with the leader server. Once connected, the client can begin submitting requests: Buy ('b'), Sell ('s'), or Inquire ('i'). Each client request must follow the format: ClientName b/s Name_of_Stock Number_of_Stocks or ClientName i. Upon successful execution of a request, the client receives a confirmation message: "Transaction Successful." from the leader server. If the leader server crashes or changes, the existing TCP connection is closed when the client attempts to place a new request. The client then repeats the leader discovery process, and establishes a new TCP connection with the new leader. The pending client request is executed by the new leader.

3.2 Server Program Flow

When a new server node joins the system, it initiates the leader discovery process by broadcasting a message and waiting for a response. Single Server Case: If no response is received within 3 seconds, the server assumes it is the only node in the system and declares itself as the leader. Multiple Server Case: If a response is received containing the leader's status and server group information, the newly joined server updates its system view and proceeds with the election process. **Leader Election Algorithm:** A logical ring topology is created by sorting the servers based on IP addresses, and each node updates its neighbor information (i.e., the node to its left).

- If there are multiple nodes, the election is initiated using LCR algorithm. The initiator sends its own UID as messages to its neighboring nodes.
- If the received UID is smaller than the node's own UID, the node becomes a participant in the election and sends its own UID as an election message to its left neighbor.
- If the received UID is larger than the node's own UID, the node forwards the message without modifying the UID.
- If the node receives its own UID back, it declares itself as the leader. This means the message has completed a full ring traversal with no other node having a higher UID.

If a new leader is elected, then the current Leader server transfers the latest objects related to share and client information to the new leader via serialization. The elected leader then starts listening to leader discovery requests from the client and responds with the leader information. The leader accepts the TCP connection request with a client when initiated. Whenever the leader receives new trading request from the client, it processes the request and acknowledges the client if the request was successfully processed. Each time when there is a new client request processed by the leader node, there is a change in the objects related to a client and its stock information. This change in the object triggers a multicast where the leader updates all other servers in the group through the data multicast handler module. The transaction data is serialized and a sequence number is assigned to it. This message is then multicasted to all other servers in the system to maintain consistency. The servers receive the multicast message and compare the received sequence number with its expected sequence number. If the sequence is in order, the message is processed immediately. In case of a mismatch (received sequence number is greater than expected), the received message is stored in a holdback queue and the server requests the missing messages from the leader. Once the missing messages are received, the message from the holdback queue is also appended to the server's client data. Thus this ensures **Total Ordered Reliable Multicast**.

The fault detection in the system is implemented by the heartbeat manager. The leader sends a broadcast heartbeat message every 8 seconds to all the servers in the system. Servers respond with unicast replies to confirm their presence. If a server fails to respond for two consecutive heartbeat cycles, it is marked as inactive and removed from the server group. Even with proper connection, heartbeat messages were not received due to UDP from non Leader servers occasionally. Hence, we set the criteria to remove a server from the server group based on the miss of two consecutive heartbeat messages.

4 Discussion and Conclusion

The project successfully implemented a distributed client-server architecture for stock trading, ensuring robustness, fault tolerance, and data consistency through

various mechanisms such as leader election, heartbeat monitoring, and reliable multicast communication. The clients can buy, sell, or inquire about stock holdings by communicating with a designated leader server. The leader handles client requests, updates stock records, and synchronizes changes across all servers in the system.If a leader node becomes unavailable, a new leader is elected dynamically using LCR algorithm, minimizing disruptions and maintaining service continuity. Through the use of multicast messaging, the leader propagates stock transaction updates to all servers, ensuring that every node maintains a consistent view of the system state. The implementation of a sequence-based message ordering mechanism further enhances reliability, preventing inconsistencies due to out-of-order message delivery. Any discrepancies in sequence numbers are efficiently handled using a holdback queue and retransmission requests to the leader. The heartbeat mechanism provides a method for detecting node failures. By broadcasting periodic heartbeat signals, the leader monitors the presence of other servers, ensuring that failed nodes are detected and removed from the server group if they fail to respond within two consecutive cycles. This enhances the resilience of the system and reduces the risk of stale or unresponsive nodes affecting overall operations. The system was tested with two servers and multiple clients submitting buy, sell, and inquiry requests. The testing included simulating leader failures and verifying that a new leader seamlessly took over. Additionally, new servers were introduced into the system without manual intervention. Reliable multicast ensured consistency by synchronizing the newly joined servers and updating all nodes after every transaction. One important limitation of this implementation is the exclusion of fluctuating stock price considerations in buy and sell transactions. While this simplifies the processing logic, incorporating a dynamic pricing mechanism could provide a more realistic trading simulation. Future enhancements could include integrating price fluctuations, demand-based pricing models, and order matching mechanisms to make the system more aligned with real-world stock exchange operations. Another area for improvement is distributing the leader server's workload among other servers, which can enhance scalability, reduce response times, and improve overall system efficiency.
Github repository link: `https://github.com/Bekal-Ramnath-Rao/Distributed_System_Team12`

Total Word Count: 2685 words