Mandatory Hand-in 5 — A Distributed Auction System / Report

5.1 Introduction

This report outlines the architecture, implementation and functionalities of a distributed auction system developed using gRPC in the Go programming language. Further down we also discuss the correctness of our implementation regarding linearizability, sequential consistency and our protocols in the absence and presence of failures.

Our system is architected with a primary-secondary node model to make the auction possible and to assure robustness. This concept helped us managing the complicated dynamics of a real-time auctions, where quick communication of information and concurrent bids are critical.

Our system's primary function is to coordinate and manage auction bids among several nodes while preserving the accuracy and consistency of the data. In a primary-secondary node model one node oversees the auction process and distribute information such as bids, while others are secondary nodes that take part in the auction by getting updates.

Key features of our system include bid handling, real-time updates of bid information across nodes, auction result retrieval, and a heartbeat mechanism for node-monitoring in case of failure. Additionally, our implementation addresses critical aspects such as fault tolerance and node failure recovery, guaranteeing that auctions continue even in the case of individual node failures.

5.2 Architecture

The architecture of our distributed auction system uses a primary-secondary node model to manage the auction. Moreover we use gRPC for managing concurrent data operations and node-to-node communication.

Our Node Structure

1. **Primary Node**: Acts as a coordinator. It is responsible for managing the auction process, accepting bids, determining the highest bid, and broadcasting updates to secondary nodes. The primary node also handles client requests for auction results and maintains a log of all activities.
2. **Secondary Nodes**: These nodes are replicas of the primary node but do not directly handle client bids. Instead, they receive updates from the primary node and maintain a copy of the current highest bid. Also they can respond to client requests for the auction results. Secondary nodes also monitor the health of the primary node via heartbeat messages.

First, we initialise the primary node. This is accomplished by running the `node.go` file without specifying a timestamp flag. During this initial run, the primary node is assigned to port 5450 (by setting the flag when run). This designation is crucial as it distinguishes the primary node from secondary nodes in our network. The primary node generates a timestamp using `time.Now()`. This timestamp becomes a reference point for the entire auction system, ensuring synchronized timing across all nodes.

Secondary nodes are created by passing argument to the cmd. This command is structured to include specific flags such as `port`, `id`, and `timestamp`. Each secondary node is assigned a unique port and ID through the `port` and `id` flags. For each secondary node, this `timestamp` flag is set using the timestamp of the primary node, obtained during its initialization. This ensures that all nodes in the system operates with a consistent understanding of time.

Communication Protocols

1. **gRPC**: The `Bid()` and `UpdateNode()` RPCs are there for the auction management, ensuring that all bids are processed and the auction state is consistently updated across all nodes.The `Result()` RPC as stated in the requirements is there to retrieve the final outcome of the auction, including the highest bid and the winner of the auction. Also we included `HeartBeat()` and `CheckNodeFailure()` RPCs which are there for maintaining the system's robustness. They enable nodes to continuously monitor each other's health and quickly detect and respond to any node failures.
2. **Bid Handling Protocol**:

   When a bid is received by the primary node, it is compared with the current highest bid. Please see the implementation of the function `Bid()`. If it's higher, the primary node updates its state and propagates this update to all secondary nodes to ensure data consistency across the system. It also checks with every Bid if the action has ended.
3. **Heartbeat Mechanism**:

   (**Secondary Node):**

The `startheartBeat()` is executed by secondary nodes. It periodically sends a heartbeat signal to the primary node to indicate that the secondary node is active and functioning correctly. The function sets up a ticker-channel that triggers a tick every 15 seconds. Upon each tick received by the `heartRateMonitor` channel, it sends a `HeartBeatRequest` to the primary node via gRPC. This request contains the secondary node's port and a flag indicating that it's a regular heartbeat check, not a failure verification. If the primary node acknowledges and sends back a `HeartBeatResponse` (ack-message), the secondary node updates its `lastTimeReceivedPrimary` field to the current time, indicating a successful communication with the primary node.

(Primary Node):

The function `HeartBeat()` is invoked on the primary node when it receives a heartbeat request from a secondary node. Upon receiving a `HeartBeatRequest`, the primary node updates its record of the last received heartbeat from the secondary node (tracked in `secondaryNodesTimeCheck`-map. This update is crucial for keeping track of the health status of each secondary node. The primary node responds with a `HeartBeatResponse` message, indicating successful receipt of the heartbeat.

3. **Node Failure and Recovery Protocol:**

   `verifySecondaryNodesHeartbeat` This function continuously monitors the time since the last heartbeat was received from each secondary node. If the time since the last received heartbeat from any secondary node exceeds a predefined threshold (30 seconds in this implementation), it triggers a check for potential node failure. This is done by sending a `CheckNodeFailureRequest` to other secondary nodes. Based on the responses from other secondary nodes (`CheckNodeFailureResponse`), the primary node decides whether the unresponsive node has failed. If confirmed, the primary node can initiate a process to restart the failed node.

   `CheckNodeFailure`**-** This function is called by the primary node on secondary nodes to verify the operational status of a potential failure within the node(s). When the secondary nodes receive this request, they attempt to send a heartbeat to the node in question. If the response is positive, it indicates that the node is still operational.

5. **Leader Failure and current approach**:

   Restoring the leader wasn't a easy task based on our approach... This couldn't be 100% tested based on several problems. However, we tried to implement a verification system that when the secondary nodes notice that the leader is out they ask each other if it happened. If yes and all them agree, every node call an election and returns an authorized bool which would compare the port numbers. The rule we defined was to chose the secondary node with the lowest port number to respawn the leader in the port 5450. However, during the restore process, everything is falling apart when we kill the leader because all other nodes are spawned by it in the beginning (children processes). In other words, all secondary nodes dies with it. Therefore, we tried to detach the children processes from it making them independent. This would solve the problem, but we couldn't make it work so far. That would have been our next step or at least try more in this way. Once detaching the secondary nodes processes we expect to make it work since it is a similar approach of restoring the node in the same port to keep the connections stabilished. `verifyPrimaryNodeAlive` - this is a similar method with the `verifySecondaryNodesHeartbeat` but where every secondary node will check the time difference between now and the last time they received and response from the leader. This is the main method that will trigger all the respawning of the primary node. It runs every 11 seconds and it is initialized as a go routine if the node type is a secondary in the main function. Once triggered by the time difference it will call the `CheckNodeFailure` passing as argument the leader address. If the other secondary nodes can't receive a signal then a `callElectionsToRestartLeader` is executed, executing and `Election` in every secondary node returning a true or false response for authorized. The secondary node that receives all authorized requests will call the `restartLeader` function which was supposed to kill the remains of the primary node process based on its PID and create a new primary node in the same port (5450).

5.3 Correctness

5.3.1 Correctness 1

In order to discuss the correctness of our distributed auction system in terms of linearizability or sequential consistency, we first define these terms and then evaluate our system against these definitions.

1. **Linearizability**:

   - Strong form of consistency for distributed systems. Linearizability ensures that all operations on the system appear to occur atomically and in some order that is consistent across all nodes in the system. That means that each operation takes place at a single point in time between its start and end points, and these operations are instantly visible to all other (in our case secondary) nodes. Linearizability can also be seen as an extension of sequential consistency.

2. **Sequential Consistency**:
    - Weaker form of consistency compared to linearizability. It requires that operations from all nodes must be seen in the same order by all parts of the system, but this order does not necessarily have to match the real-time order in which the operations were issued.

Evaluating Your Implementation

As stated above, given that updates (like new bids) are immediately visible to all nodes in a consistent state, our system is linearizable. The operations appear atomic and the changed state is reflected across the whole network. Our system also demonstrates sequential consistency since it ensures that all nodes see operations like `Bid` in the same sequence. This is because only the primary node handles the order of which the bids are placed. In a scenario where multiple bids are placed at nearly the same time, our system would be able to maintain a consistent state across all nodes because all bids are processed by the primary and in the end the highest bid wins the auction.

5.3.2 Correctness 2

To argue the correctness of our protocol in both the absence and presence of failures, we will examine how our distributed auction system behaves under normal conditions and then compare it how it responds to node failures:

Correctness in the Absence of Failures

As stated above the primary node coordinates the auction process, while secondary nodes are kept up-to-date with the latest auction state. The Heartbeat messages are regularly sent from secondary nodes to the primary node confirming their operational status. The regular receipt of these messages indicates a healthy system state. This is also visible in our log files:

```
2023/11/28 03:10:44 Started node at port: 5450
2023/11/28 03:10:44 Started secondary node 1 with PID 22803
2023/11/28 03:10:44 Started secondary node 2 with PID 22809
2023/11/28 03:10:45 Started secondary node 3 with PID 22900
2023/11/28 03:11:00 Heartbeat received from the secondary node: localhost:5451
2023/11/28 03:11:00 Heartbeat received from the secondary node: localhost:5452
2023/11/28 03:11:00 Heartbeat received from the secondary node: localhost:5453
2023/11/28 03:11:15 Heartbeat received from the secondary node: localhost:5451
2023/11/28 03:11:15 Heartbeat received from the secondary node: localhost:5452
2023/11/28 03:11:15 Heartbeat received from the secondary node: localhost:5453
```

In our system at any given time, all nodes have an up-to-date and uniform view of the auction state. The protocol ensures that communication between nodes, including bid updates and heartbeat signals, occurs in a timely manner as seen in the logs with a 15 seconds delay.

Correctness in the Presence of Failures

As stated when a heartbeat from a secondary node is not received within the expected time frame (30 seconds), the primary node recognizes a potential failure. The protocol initiates a `CheckNodeFailure` process to confirm the status of the node in question:

```
2023/11/28 03:12:45 Heartbeat received from the secondary node: localhost:5451
2023/11/28 03:12:45 Heartbeat received from the secondary node: localhost:5453
2023/11/28 03:12:46 Heartbeat time error found for connection: &{0xc000100400} - port: localhost:5452
2023/11/28 03:12:46 Sending a Check Node Failure request for all other secondary nodes...
2023/11/28 03:12:46 Error checking node failure gRPC connection rpc error: code = Unavailable desc = connection error: desc = "transpor
2023/11/28 03:12:46 Error checking node failure gRPC connection rpc error: code = Unavailable desc = connection error: desc = "transpor
2023/11/28 03:12:46 Check Node Failure by secondary nodes completed.
2023/11/28 03:12:46 The node cannot be reached even for the secondary nodes. Restarting the node...
```

Upon confirming a node failure, our system takes action. This is evident from the protocol file where a secondary node (`localhost:5452`) fails to send a heartbeat, is identified, as seen above and subsequently restarted:

```
2023/11/28 03:12:46 Trying to find the process running for port localhost:5452
2023/11/28 03:12:46 Killing the PID 22809 and respawning the node for localhost:5452 ...
2023/11/28 03:12:46 Started secondary node 2 with PID 24185
2023/11/28 03:12:57 Heartbeat time error found for connection: &{0xc000100400} - port: localhost:5452
2023/11/28 03:12:57 Sending a Check Node Failure request for all other secondary nodes...
2023/11/28 03:12:57 Check Node Failure by secondary nodes completed.
2023/11/28 03:12:57 The node could be reached for the secondary nodes. Updating the heartbeat time and continuing the process...
2023/11/28 03:13:00 Heartbeat received from the secondary node: localhost:5451
```

```
2023/11/28 03:13:00 Heartbeat received from the secondary node: localhost:5453
2023/11/28 03:13:01 Heartbeat received from the secondary node: localhost:5452
```

After a failed node is restarted, it rejoins the system and resumes its role as seen with node 5452. The protocol ensures that the node is resynchronized with the current state of the auction, maintaining consistency across the network.

Test case → In our test case we killed the localhost:5452 process via htop based on its PID.

5.4 Provide a link to a Git repo with your source code in the report

- Link to our Repository:

  https://github.itu.dk/edtr/distributedsystems/tree/main/5Assignment

5.5 Include system logs, that document the requirements are met, in the appendix of your report

- Please look at the log files in our linked repository:
  https://github.itu.dk/edtr/distributedsystems/tree/5assignment/5Assignment/logs