Mandatory Hand-in 5: **Replication**

1. Introduction

To build an Auction service with the ability to bid and see the stats of an auction, we have built a distributed server system based on Replication. This system uses a leader-based replication to be crash tolerant. Whenever a master crashes, we can notice it because each server peer contacts the server every 8 seconds with a ping message, the bully algorithm is initiated to elect a new master server. This means that our system contains 2 gRPC services. A service for the clients to do auctions, and a service for the peer servers to elect new masters and exchange information about the auction so each peer is always up to date.

2. Architecture

The system consists of server peers (replicants) and clients to test the service while server nodes are crashing.

- **Server**

Each server has a name, IP, port and ID (see appendix 4.1). The IP, port and ID are hardcoded in confFile.csv for connecting the peers with each other. The port represents the one on which the peer will expose its services. During the initialization of a server an election is made by the function *updateMaster* (see appendix 4.2). This function and all others connected to it represents the bully algorithm which is shown by example below in figure 2.1

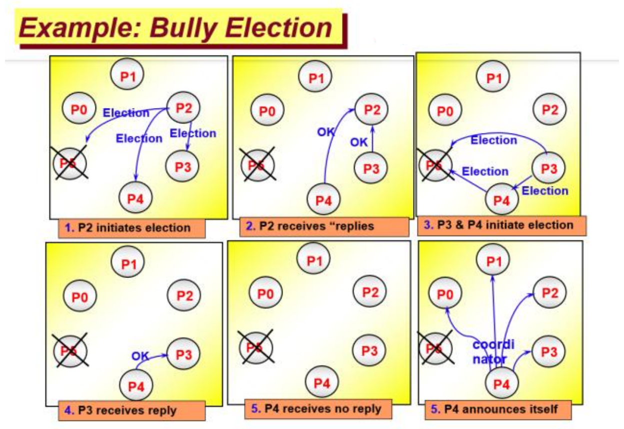


Figure 2.1: Bully algorithm

The *updateMaster* function now calls a new function *sendElectionToBigger* (see appendix 4.3). This function checks whether there is a peer with a higher id (see if anyone confirms receipt of election message), if so the peer will wait for the coordinator message to be received, otherwise it will repeat the procedure after 5 seconds.

The peer receiving the selection message performs the same *updateMaster* function, sending the messages only to the higher peers. When a peer has no other peers above it, it will send everyone below the coordination message and declare itself master. It will use the *sendCoordinatorToLower* function to do this. The peers will then know who the master is, and they will change the global variables with the master IP, port and ID using mutex to lock it while changing it. These global variables can be readen from the clients by using the *AskForMaster* function provided by the DistributedService.
The purpose of the mutex is to prevent these variables from being read during an update of them by creating an IP and port mismatch of the master.

The bidding service works by letting a client type bid followed by a number. The number is then checked by the master if it's larger or smaller than the current highest bid. If the bidding request is sent to any other server than the master, it will return an error (see appendix 4.4) since this defeats the purpose of leader-based replication. When a higher bid has been noted, the master, using the *UpdateAuction* function present in the gRPC service of the server peers, will inform the other peers about the update and only then send confirmation to the client about the bid made.
In case of connection of a new peer with higher id than the master, to be sure that the data on the current auction is not lost, before updating the master, the new peer asks the current master to have an update of the auction data via the gRPC function *GetAuctionData.*

- Client

The client will be responsible for sending messages to the master to place new bids or check the status of the auction. Before doing this, however, based on the configuration file, it will connect to various peers by querying them about who the current master is. Once this information is obtained, it will only interact with the master. In the event of no response from the master, it will repeat the process mentioned earlier. If a response is received from a peer indicating that it is no longer the master, it will use the *AskForMaster* function to obtain the new values.

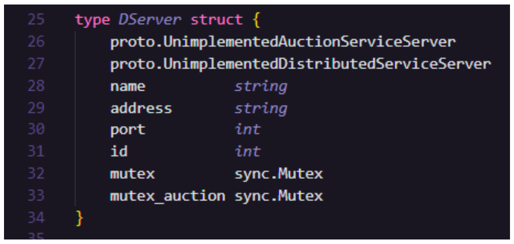
3. Correctness

Linearizability ensures that each write operations on the server appear by client's perspective as if they happen instantly at a specific point in time. Instead, sequential consistency ensures that the overall order of operations is consistent with the order in which they were issued by individual clients.
Our project is neither linearizable nor sequentially consistent. In the first case, clients and servers have two asynchronous clocks, while in the second case, to ensure that the order of execution is the same as the order of arrival, we would need to ensure that our mutex applies a FIFO queue for threads waiting, something that the Golang mutex does not do.

In the absence of failures, our protocol will use only the master, and the architecture is comparable to a classic client-server model. In this case, clients will always communicate with the server, which will handle the bids, ensuring that the update of the amount is protected by a mutex. However, in the case of a master failure, if there is at least one other available peer server, it will detect the failure because the master will no longer respond to the ping message. The peer server will then become the new master, having all the updated auction data.

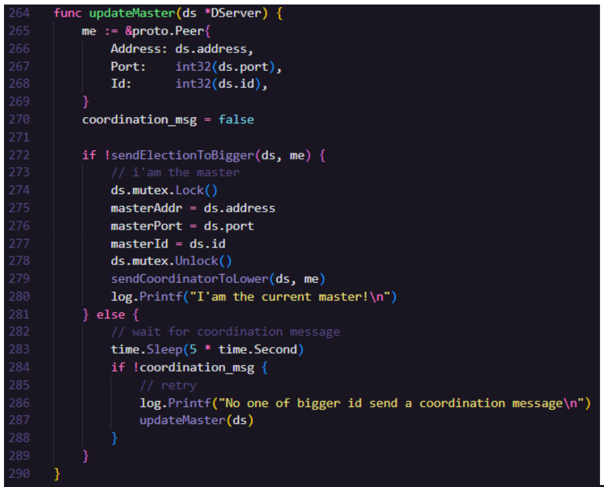
4. Appendix

4.1 Server struct



```go
type DServer struct {
    proto.UnimplementedAuctionServiceServer
    proto.UnimplementedDistributedServiceServer
    name         string
    address      string
    port         int
    id           int
    mutex        sync.Mutex
    mutex_auction sync.Mutex
}
```

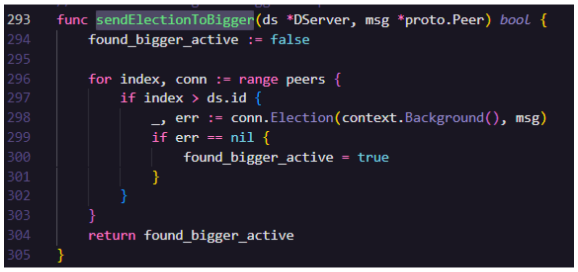
4.2 *updateMaster* function



```go
func updateMaster(ds *DServer) {
    me := &proto.Peer{
        Address: ds.address,
        Port:    int32(ds.port),
        Id:      int32(ds.id),
    }
    coordination_msg = false

    if !sendElectionToBigger(ds, me) {
        // i'am the master
        ds.mutex.Lock()
        masterAddr = ds.address
        masterPort = ds.port
        masterId = ds.id
        ds.mutex.Unlock()
        sendCoordinatorToLower(ds, me)
        log.Printf("I'am the current master!\n")
    } else {
        // wait for coordination message
        time.Sleep(5 * time.Second)
        if !coordination_msg {
            // retry
            log.Printf("No one of bigger id send a coordination message\n")
            updateMaster(ds)
        }
    }
}
```

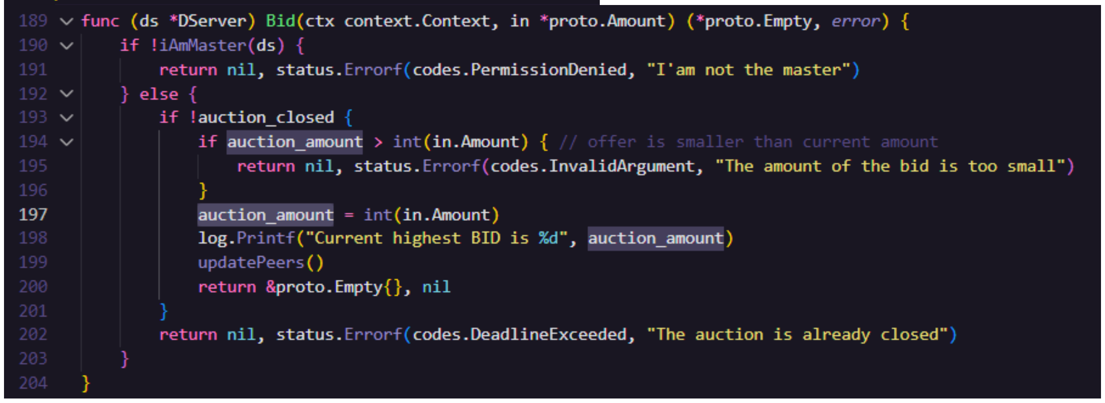
4.3 *sendElectionToBigger* function



```go
func sendElectionToBigger(ds *DServer, msg *proto.Peer) bool {
    found_bigger_active := false

    for index, conn := range peers {
        if index > ds.id {
            _, err := conn.Election(context.Background(), msg)
            if err == nil {
                found_bigger_active = true
            }
        }
    }
    return found_bigger_active
}
```

4.4 *Bid* function



```go
func (ds *DServer) Bid(ctx context.Context, in *proto.Amount) (*proto.Empty, error) {
    if !iAmMaster(ds) {
        return nil, status.Errorf(codes.PermissionDenied, "I'am not the master")
    } else {
        if !auction_closed {
            if auction_amount > int(in.Amount) { // offer is smaller than current amount
                return nil, status.Errorf(codes.InvalidArgument, "The amount of the bid is too small")
            }
            auction_amount = int(in.Amount)
            log.Printf("Current highest BID is %d", auction_amount)
            updatePeers()
            return &proto.Empty{}, nil
        }
        return nil, status.Errorf(codes.DeadlineExceeded, "The auction is already closed")
    }
}
```