Distributed Auction System

Final Project Report

B032427 (B335) - Distributed Programming for Web, IoT and Mobile Systems
2025-2026

Albert Borchardt
Matricola: 7190123

1. Introduction

My project implements a distributed auction system where multiple nodes can accept bids from clients and replicate the auction state across the cluster. The system uses a centralized coordinator for mutual exclusion, Lamport timestamps for ordering events, gRPC for internal communication and a REST API for external clients.

The architecture consists of one coordinator that manages distributed locking and multiple nodes that handle client requests and broadcast state updates to each other. Through this setup I explore coordination mechanisms, logical clocks, state replication and the tradeoffs that come with distributed system design.

2. Design and Architecture

2.1 Project Structure

```
distributed-auction/
├── cmd/
│   ├── coordinator/main.go   # Coordinator entrypoint
│   └── node/main.go          # Node entrypoint + REST server
├── internal/
│   ├── coordinator/
│   │   ├── client.go         # gRPC client to coordinator
│   │   └── server.go         # Lock service implementation
│   ├── domain/
│   │   └── auction.go        # State, Bid structs
│   └── node/
│       ├── node.go           # Node logic (SubmitBid, Replicate)
│       └── replicator.go     # Broadcast to peer nodes
├── proto/
│   ├── auction.proto         # Service definitions
│   ├── auction.pb.go         # Generated message structs
│   └── auction_grpc.pb.go    # Generated client/server code
├── Dockerfile
└── docker-compose.yml
```

2.2 Components

Coordinator (Port 7000)

The coordinator provides a centralized lock service via gRPC. It implements the centralized mutual exclusion algorithm from the lecture using a channel based token mechanism. Only the node holding the token can process a bid.

```
// internal/coordinator/server.go
type Server struct {
    lockChan chan struct{}  // Capacity 1: token = lock available
}


func (s *Server) Acquire(ctx, req) (*LockResponse, error) {
    select {
    case <-s.lockChan:       // Take token out -> lock acquired
        return &LockResponse{}, nil
    case <-ctx.Done():       // Timeout
        return nil, ctx.Err()
    }
}
```

Nodes (REST: 8080-8082, gRPC: 9090)

Each node exposes a REST API for clients (using Gorilla Mux as shown in the lecture) and communicates with the coordinator and other nodes via gRPC. The node maintains local state and a Lamport clock:

```
// internal/node/node.go
type Node struct {
    lock    coordinator.LockService  // Client to coordinator
    repl    Replicator               // Broadcasts to peers
    mu      sync.Mutex               // Local mutex
    state   domain.State             // Current auction state
    lamport int64                    // Logical clock
}
```

The local mutex (mu) protects the state and lamport fields from race conditions within the node. For example when a REST request to submit a bid and an incoming gRPC Replicate call happen at the same time the mutex ensures only one can access the state.

Replicator

The replicator broadcasts state updates to all peer nodes in parallel using goroutines:

```
// internal/node/replicator.go
func (r *GRPCReplicator) Broadcast(ctx, state) {
    for _, peer := range r.peers {
        go r.sendToPeer(peer, state)  // Parallel, non-blocking
    }
}
```

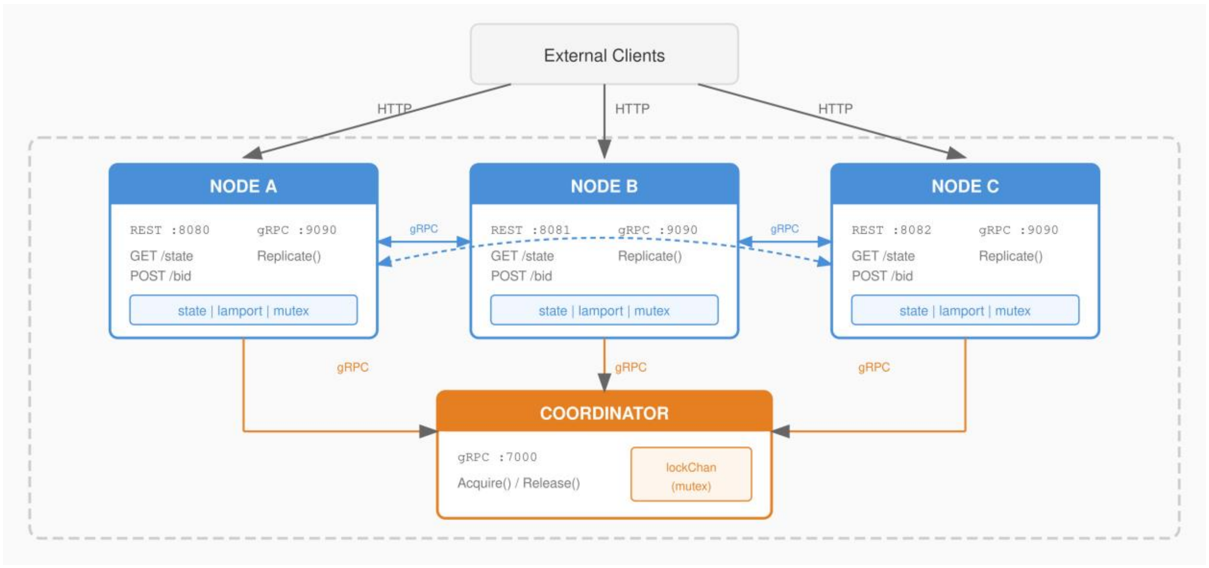

Figure 1: System architecture within the Docker network. Each container has its own IP so all nodes can use port 9090 for gRPC. Without Docker on a single host the gRPC ports would need to be unique (e.g. 9090, 9091, 9092).

2.3 Communication Flow

When a client submits a bid:

1. Client sends POST /bid to any node
2. Node requests lock from coordinator (gRPC Acquire)
3. Node validates bid and (if better) updates local state + Lamport timestamp
4. Node releases lock (gRPC Release)
5. Node broadcasts the updated state to peer nodes (gRPC Replicate)
6. Node responds to the client (includes applied + current local state)

The replication is asynchronous. The globally visible result is observed via GET /state after convergence.

3. Technologies

3.1 gRPC and Protocol Buffers

Internal communication uses gRPC with Protocol Buffers as covered in the lecture for type safe and efficient serialization. I define the services in a .proto file:

```
// proto/auction.proto
service CoordinatorService {
    rpc Acquire(LockRequest) returns (LockResponse);
    rpc Release(LockRequest) returns (LockResponse);
}

service NodeService {
    rpc Replicate(AuctionState) returns (Ack);
}
```

The protoc compiler generates Go structs and client/server interfaces from this definition. I create a client by passing a gRPC connection and then use the generated methods to make remote calls.

3.2 REST API

External clients interact with nodes through REST using Gorilla Mux as introduced in the lecture:

```
// cmd/node/main.go
r := mux.NewRouter()
r.HandleFunc("/bid", func(w, r) {
    var req struct { Amount int64; BidderID string }
    json.NewDecoder(r.Body).Decode(&req)
    state, applied, _ := n.SubmitBid(req.Amount, req.BidderID)
    json.NewEncoder(w).Encode(map[string]interface{}{
        "applied": applied, "state": state,
    })
}).Methods("POST")
```

3.3 Docker

For deployment I used Docker Compose to run the coordinator and three nodes as separate containers on a shared network as it makes testing easier and demonstrates how the system would run in a distributed environment.

4. Course Topics Covered

Coordination and Synchronization:

- Lamport Logical Clocks for ordering events and detecting old data
- Centralized Mutual Exclusion using a token based mechanism

Communication Mechanisms:

- Remote Procedure Call (gRPC) for coordinator service and node to node replication
- Protocol Buffers for message serialization
- REST for external API

Distributed System Architecture:

- Client Server architecture
- State Replication with broadcast

Go Programming:

- Concurrency with goroutines and channels
- Interfaces and dependency injection
- sync.Mutex for local synchronization

5. Use Case: Running an Auction

5.1 Scenario

An online auction for a product where bidders can participate from anywhere over the internet. Multiple servers handle incoming bids and need to stay synchronized so that every bidder sees the same current highest bid.

5.2 Starting the System

```
docker compose up --build
```

This starts 4 containers: 1 coordinator and 3 nodes.

5.3 Bidding

Check initial state:

```
curl localhost:8080/state
-> {"highestBid":0,"highestBidder":"","lamport":0}
```

Alice bids 100 at Node A:

```
curl -X POST localhost:8080/bid -d '{"amount":100,"bidderId":"alice"}'
->
{"applied":true,"state":{"highestBid":100,"highestBidder":"alice","lamport"
:1}}
```

Check Node B (replication):

```
curl localhost:8081/state
-> {"highestBid":100,"highestBidder":"alice","lamport":1}
```

Bob bids 50 at Node B (rejected because lower than current highest):

```
curl -X POST localhost:8081/bid -d '{"amount":50,"bidderId":"bob"}'
->
{"applied":false,"state":{"highestBid":100,"highestBidder":"alice","lamport
":1}}
```

Bob bids 150 at Node B (accepted):

```
curl -X POST localhost:8081/bid -d '{"amount":150,"bidderId":"bob"}'
->
{"applied":true,"state":{"highestBid":150,"highestBidder":"bob","lamport":2
}}
```

All nodes in sync:

```
curl localhost:8080/state && curl localhost:8081/state && curl
localhost:8082/state
-> {"highestBid":150,"highestBidder":"bob","lamport":2}
-> {"highestBid":150,"highestBidder":"bob","lamport":2}
-> {"highestBid":150,"highestBidder":"bob","lamport":2}
```

The README contains additional examples including parallel requests to demonstrate more complex use cases.

6. Design Decisions and Tradeoffs

6.1 Consistency Model

I chose a best effort approach for consistency. When a node broadcasts its state to peers it does not wait for acknowledgments and does not retry on failure. Since every accepted bid triggers a replicate to all peers I expect nodes to converge quickly under normal operation. In this small setup with three nodes on a local Docker network unreachable peers are unlikely.

However, I am aware that for use cases requiring stronger guarantees this approach would not be sufficient. For example in a financial trading system where every transaction must be confirmed one could use a Quorum based approach as covered in the lecture which would guarantee consistency but comes at the cost of availability (if too many nodes are down the system blocks).

6.2 Lamport Clocks and Tie Breaking

I use Lamport timestamps so that each state update has a version number. When a node receives a replicated state it compares the incoming timestamp with its current state. If the incoming timestamp is lower or equal it means the message is outdated and can be ignored. This prevents old messages that arrive late from overwriting newer state.

However during development I discovered that Lamport timestamps alone are not enough. They provide only a partial ordering: if two nodes process bids at nearly the same time they can end up with the same timestamp but different states.

To fix this I added a deterministic tie break: when timestamps and the bid amount are equal the bid with the lexicographically smaller bidder ID wins:

```go
// internal/node/node.go (Replicate)
apply := false
if req.Lamport > n.state.Lamport {
	apply = true
} else if req.Lamport == n.state.Lamport {
	if req.HighestBid > n.state.HighestBid {
		apply = true
	} else if req.HighestBid == n.state.HighestBid {
		apply = req.HighestBidder < n.state.HighestBidder
	}
}
```

This is not an ideal rule because it arbitrarily favors certain bidders in a rare situation like this. But it made me aware of Lamport's fundamental limitation: Lamport clocks cannot capture causality. Two events with the same Lamport timestamp are concurrent and we cannot determine which happened first. For example Vector Clocks could help here by tracking the causal history of each node but implementing them would have required too much restructuring. I chose to keep the simpler approach and document this as a limitation.

6.3 Dependency Injection

The node depends on interfaces rather than concrete implementations:

```
type LockService interface {
    Acquire(ctx, nodeID string) error
    Release(ctx, nodeID string) error
}

type Replicator interface {
    Broadcast(ctx, state domain.State)
}
```

This makes testing easy because I can inject mocks that do not require network access. But it also enables extensibility: one could swap the centralized coordinator for a distributed consensus implementation or replace the gRPC replicator with a message queue without changing the node logic.

7. Limitations and Future Work

- **Single point of failure:** The coordinator is a single point of failure. If it goes down no bids can be processed. A production system would need coordinator replication.
- **No automatic lock release on crash:** If a node acquires the lock and then crashes before releasing it the coordinator remains blocked. A production system would need lease-based locks with automatic expiration or heartbeat mechanisms to detect failed nodes.
- **No persistence:** State is kept in memory. Restarting a node loses all data. Adding a database like SQLite as covered in the lecture would be necessary for durability.
- **Lamport limitations:** As discussed Lamport clocks only provide partial ordering. For example Vector Clocks would enable detecting true concurrency and could support more sophisticated conflict resolution.
- **No retry on broadcast failure:** If a peer is unreachable during broadcast the update is lost for that peer. The peer will only get the latest state when the next successful broadcast happens.

8. Conclusion

This project implements a distributed auction system that demonstrates centralized mutual exclusion, Lamport logical clocks, state replication and consistency tradeoffs. I chose availability over strong consistency which fits certain auction models but would need reconsideration for other domains. The tie breaking problem revealed the limitations of Lamport clocks and pointed toward Vector Clocks as a potential improvement. The modular design with dependency injection keeps the code testable and extensible for future work.

References

[1] M. van Steen and A. S. Tanenbaum, Distributed Systems, 4th edition (course textbook).

[2] Course materials: B032427 - Distributed Programming for Web, IoT and Mobile Systems, 2025-2026.

[3] gRPC Documentation. https://grpc.io/docs/

[4] Protocol Buffers Documentation. https://protobuf.dev/

[5] The Go Blog. https://go.dev/blog/

Declaration on the Use of AI Tools

AI-based language tools were used in a limited manner for technical debugging and linguistic refinement in the report and code documentation. All design decisions, implementation, technical content and analysis of tests and results were performed entirely by me.