CS 351 Project 5

Distributed Auction System

Final Design Document

Group 01

Utshab Niraula
Sushant Bogati
Priyash Chandara
Aditya Chauhan

December 5, 2025

Contents

1 Architecture Overview **3**

2 System Architecture Diagram **3**

3 Bank Component Design **5**
3.1 Bank Object Diagram 5
3.2 Component Descriptions 5
3.2.1 bank.BankServer 5
3.2.2 bank.BankClientHandler (Subcomponent for Communication) 6
3.2.3 bank.Bank 6
3.2.4 bank.BankAccount 6

4 Auction House Component Design **7**
4.1 Auction House Object Diagram 7
4.2 Component Descriptions 7
4.2.1 auctionhouse.AuctionHouseServer 7
4.2.2 auctionhouse.AuctionHouseClientHandler (Communication) 7
4.2.3 auctionhouse.AuctionHouse 8
4.2.4 auctionhouse.AuctionItemManager 8
4.2.5 common.AuctionItem 8

5 Agent Component Design **9**
5.1 Component Descriptions 9
5.1.1 agent.AutomatedAgent 9
5.1.2 agent.Agent 9
5.1.3 common.NetworkClient (Subcomponent for Communication) 9

1 Architecture Overview

The Distributed Auction System follows a strict Client-Server architecture designed to support concurrent bidding and financial transactions across a network. The system is composed of three primary nodes:

1. **Bank:** The central authority for financial state. It acts purely as a server.
2. **Auction House:** A hybrid node that acts as a client to the Bank (for verifying funds) and a server to Agents (for hosting auctions).
3. **Agent:** The client node used by bidders (automated or human). It acts purely as a client.

Communication Protocol

All nodes communicate using TCP sockets exchanging serialized Java objects (extending `common.Messag`
This ensures type safety and structured data exchange compared to raw text protocols.

2 System Architecture Diagram

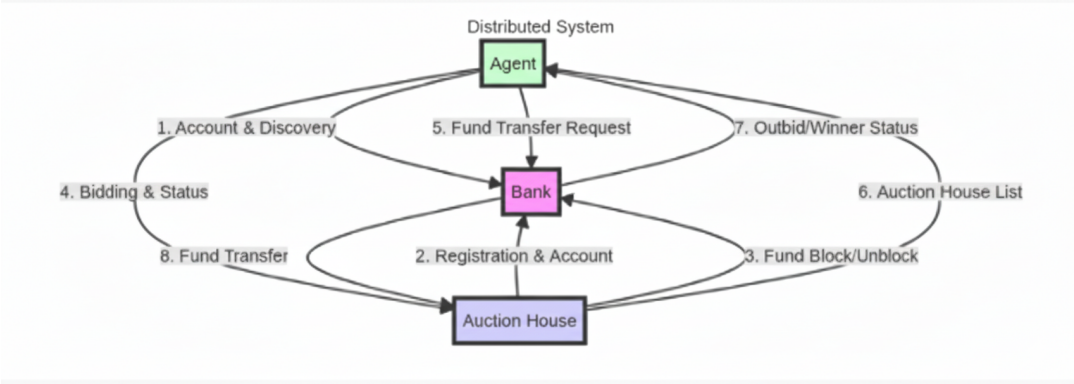

Figure 1: High-level topology showing the Bank, Auction House, and Agent interactions.

Data Flow:

1. **Registration:** Agents and Auction Houses register with the Bank to obtain unique IDs and accounts.
2. **Discovery:** Agents request the list of active Auction Houses from the Bank.
3. **Bidding:** Agents connect to Auction Houses to view items and place bids.

4. **Transaction:** When an auction ends, the Agent initiates a fund transfer via the Bank.
5. **Confirmation:** The Auction House verifies the transfer and releases the item to the winner.

3 Bank Component Design

The Bank component is responsible for thread-safe management of user funds. It must handle concurrent requests from multiple Auction Houses attempting to block funds simultaneously.

3.1 Bank Object Diagram

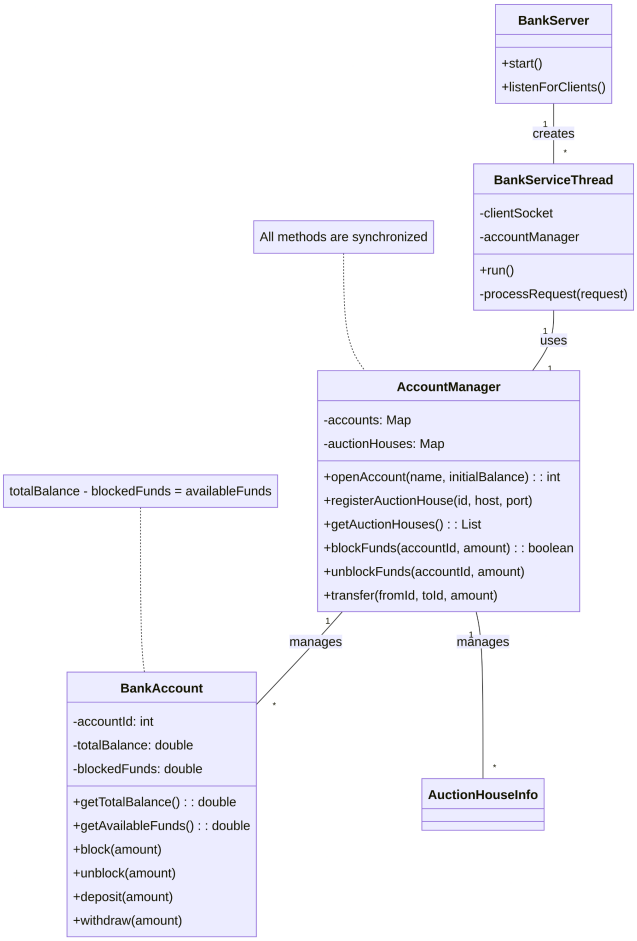

Figure 2: UML Class Diagram for the Bank Component.

3.2 Component Descriptions

3.2.1 bank.BankServer

- **Role:** The entry point for the Bank application.
- **Socket Communication:** Initializes a `ServerSocket` on a specified port (default 5000). It listens for incoming connections in a loop and spawns a new thread (`BankClientHandler`) for each connection.
- **Key Methods:**
  - `start()`: Begins the server loop.
  - `stop()`: Safely shuts down the server.

3.2.2 bank.BankClientHandler (Subcomponent for Communication)

- **Role:** Handles the actual socket communication for a single connected client (Agent or Auction House).
- **Fields:**
  - `Socket socket`: The active connection.
  - `ObjectInputStream / ObjectOutputStream`: Streams for sending serialized `Message` objects.
- **Key Methods:**
  - `run()`: Continuously reads `Message` objects from the stream and dispatches them to the `Bank` logic.
  - `handleMessage(Message)`: Switch-statement logic to route requests (e.g., `BLOCK_FUNDS`, `TRANSFER_FUNDS`).

3.2.3 bank.Bank

- **Role:** The central logic controller containing the state of the system.
- **Fields:**
  - `Map<Integer, BankAccount> accounts`: Thread-safe storage (`ConcurrentHashMap`) of all accounts.
  - `Map<Integer, AuctionHouseInfo> auctionHouses`: Registry of active auction houses.
- **Key Methods:**
  - `registerAgent(...)`: Creates a new account and returns the ID.
  - `blockFunds(...)`: Synchronized method to hold funds for a bid.
  - `transferFunds(...)`: Atomic operation to move funds from one account to another.

3.2.4 bank.BankAccount

- **Role:** A data model representing a single user's financial state.
- **Fields:**
  - `double totalBalance`: The actual money in the account.
  - `double blockedFunds`: Money currently tied up in active bids.
- **Key Methods:**
  - `blockFunds(amount)`: Synchronized method. Checks if available funds are sufficient before increasing blocked funds.
  - `transferFunds(amount)`: Finalizes a transaction by reducing both blocked funds and total balance.

4 Auction House Component Design

The Auction House manages inventory and bidding logic. It must handle concurrency to ensure that bids are processed sequentially per item, preventing race conditions.

4.1 Auction House Object Diagram

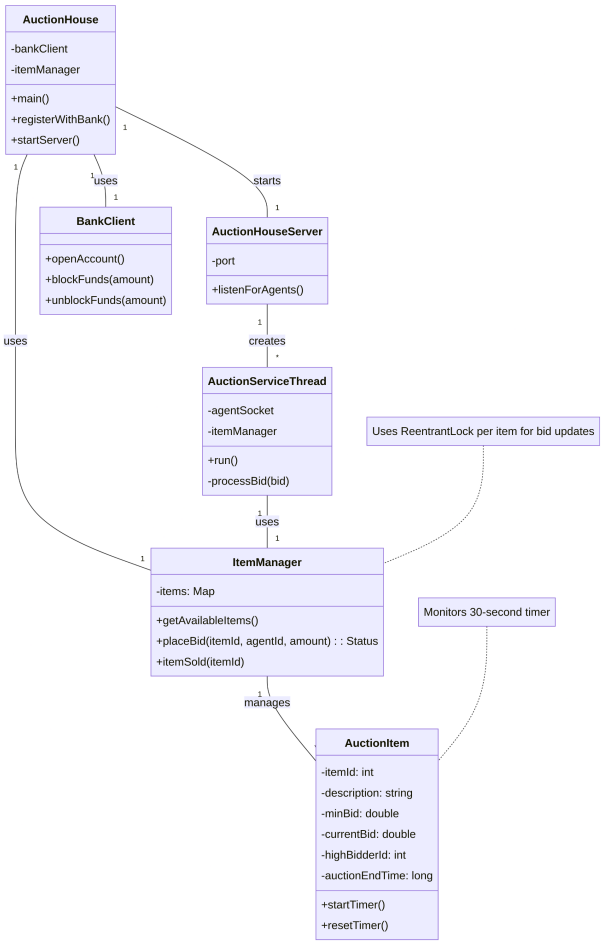

Figure 3: UML Class Diagram for the Auction House Component.

4.2 Component Descriptions

4.2.1 auctionhouse.AuctionHouseServer

- **Role:** The entry point for the Auction House.
- **Socket Communication:** Establishes a server socket to accept incoming connections from Agents. It delegates connection handling to `AuctionHouseClientHandler`.
- **Key Methods:**
  - `listenForAgents()`: Accepts connections and starts handler threads.

4.2.2 auctionhouse.AuctionHouseClientHandler (Communication)

- **Role:** Manages the persistent TCP connection with a specific Agent.
- **Fields:**

- `NetworkClient agentClient`: Wrapper around the socket streams.
- `int agentAccountNumber`: Identity of the connected agent.

- **Key Methods:**
  - `run()`: Listens for `PlaceBidRequest` or `GetItemsRequest`.
  - `handleMessage(...)`: Parses requests and calls methods in `AuctionHouse`.

4.2.3 auctionhouse.AuctionHouse

- **Role:** The main controller. It coordinates between the Item Manager and the Bank connection.
- **Fields:**
  - `NetworkClient bankClient`: The socket connection to the Bank (acting as a client).
  - `AuctionItemManager itemManager`: The logic unit for managing items.
- **Key Methods:**
  - `placeBid(...)`: Validates a bid and updates the item state.
  - `confirmWinner(...)`: Verifies a winner and releases the item.
  - `broadcastToAllAgents(...)`: Sends updates (like `OUTBID` or `ITEM_SOLD`) to all connected agents.

4.2.4 auctionhouse.AuctionItemManager

- **Role:** Manages the collection of auction items and enforces bidding rules.
- **Fields:**
  - `Map<Integer, AuctionItem> items`: Thread-safe map of all items.
- **Key Methods:**
  - `placeBid(...)`: Synchronized method. Checks if the bid is valid (higher than current, auction open) and updates the item.
  - `closeItem(...)`: Marks an item as sold/closed.

4.2.5 common.AuctionItem

- **Role:** Serializable data object representing an item.
- **Fields:**
  - `int currentBidderAccountNumber`: ID of the current high bidder.
  - `double currentBid`: Current highest price.
  - `long auctionEndTime`: Timestamp for when the auction expires.

5 Agent Component Design

The Agent is the client-side application. It can be manual (GUI) or automated (Bot).

5.1 Component Descriptions

5.1.1 agent.AutomatedAgent

- **Role:** An autonomous bot that bids based on a specific strategy.
- **Fields:**
  - `Agent agent`: The underlying client logic.
  - `double bidMultiplier`: Factor to increase bids (e.g., 1.15x).
  - `long bidInterval`: Time delay between actions.
- **Key Methods:**
  - `start()`: Launches the bidding thread.
  - `calculateBidAmount(item)`: Determines the next bid based on current price and multiplier.
  - `run()`: Main loop. Checks financial safety (blocked funds $< 75\%$) before placing bids.

5.1.2 agent.Agent

- **Role:** The core client logic that maintains state (balance, inventory).
- **Socket Communication:** Uses `NetworkClient` to maintain simultaneous connections to the Bank and multiple Auction Houses.
- **Key Methods:**
  - `connectToBank(...)`: Establishes initial identity.
  - `placeBid(...)`: Sends a `PlaceBidRequest` to a specific house.

5.1.3 common.NetworkClient (Subcomponent for Communication)

- **Role:** A generic wrapper for Socket Input/Output streams used by both Agents and Auction Houses.
- **Socket Communication:** Encapsulates `Socket`, `ObjectOutputStream`, and `ObjectInputStream`
- **Concurrency:** Uses internal locks (`readLock`, `writeLock`) to ensure thread-safe sending and receiving of messages, preventing race conditions when multiple threads share a connection.