Auction System

Group 18: Philipp Wannke, David Tsoi, and Alexander Gliese

University of Stuttgart

1 Introduction

The following report describes a distributed auction system designed to facilitate real-time auction creation and bidding. Built using Python, the system leverages a client-server architecture with a focus on scalability and robustness in a controlled but distributed environment. The system is designed to handle multiple auctions simultaneously, with dynamic discovery of hosts, fault tolerance, elections among server nodes, and reliable communication. The client user interface is navigated using the command line. The code is available on GitHub under `https://github.com/Alexander848/DS_AuctionSystem`.

2 Project requirements analysis

2.1 Architectural Description

We assume our servers to be set up in a controlled environment like a data-center where we can dictate node behavior and communication topology. Nodes can fail at arbitrary times and messages can get lost at any point, but every node can be reached from any other node eventually.

System components Our project follows a client-server model. In order to ensure high scalability, many server nodes exist at the same time. Said server nodes can be grouped into two main parts: A group of idle nodes and auction nodes participating in an ongoing auction.

- The pool of idle nodes have two types of nodes: One Idle Node Manager(INM) and the Idle Server Nodes(ISN). The INM manages the pool of idle nodes and handles client requests outside of auctions, creates new auctions and replaces failed auction nodes. The ISN wait to be used as an auction node or to replace the INM in case it fails.
- Every auction uses two server nodes. One Active Auction Node(AAN) that handles client bids and informs clients about the state of the auction and one Passive Auction Node(PAN) that replicates the AAN's data and replaces it in case of failure.
- The system's third components are the clients. The clients can start auctions, join existing auctions and participate in auctions.

Communication Model Building upon our assumption of a controlled environment, we assume to have a synchronous message model with a bound message transmission time. Nodes which exceed this message transmission limit are considered as failed. To realize this communication, we provide a versatile middleware used by both our server nodes and the clients for unicast as well as multicast communication.

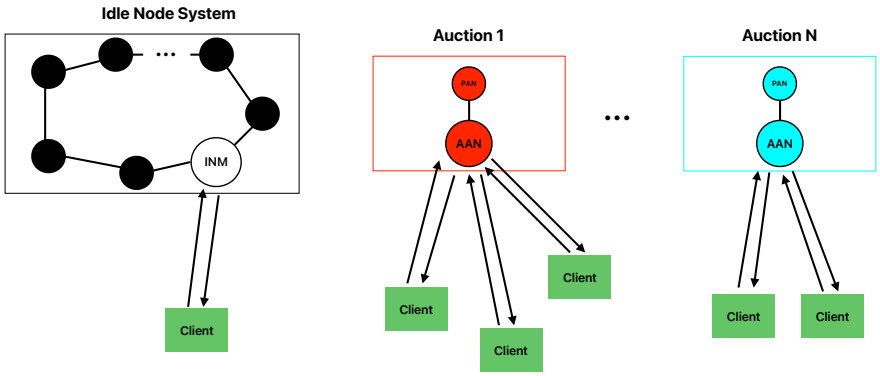

Fig. 1. An illustration of the running system.

2.2 Dynamic discovery of hosts

Dynamic discovery is used within our distributed auction system to locate system nodes whose address is not known and/or stored. This allows for higher flexibility and more robustness. IP addresses are not fixed in place, but all server nodes and clients have a socket to send and receive messages from a common multicast group. Dynamic discovery takes place in both clients and server nodes.

Client Clients have a list of operations they can perform upon user request as detailed in section 4.3. When a client is already part of an ongoing auction, it can communicate with the appropriate AAN. However, whenever a client is prompted to perform operations before taking part in an active auction, it needs to locate the INM. These operations are:

- Listing all available items up for auction: The client sends a multicast message to the multicast group. Only the INM reacts to this message by sending a response with a list of all the items.
- Starting an auction for a certain item: The client sends a multicast message to the multicast group. Upon delivery, the INM prepares a new auction and responds with the address of the newly created AAN for the requested auction.

– Joining an already started auction: The client sends a multicast message to the multicast group. Upon delivery, the INM responds with the address of the AAN for the requested auction.

Server In addition to client discovery, server nodes also rely on dynamic discovery mechanisms to locate the INM. Because the INM is responsible for maintaining a consistent group view of the idle nodes, a newly joined server node must determine the current INM to integrate seamlessly into the distributed network. This happens once at startup and every time after an auction finishes. Upon a server node's discovery request, the INM responds with a group view update, which provides a snapshot of the existing server nodes in the system. This update allows the new node to establish a peer connection and participate in system operations. Additionally, if the INM fails, the remaining server nodes dynamically elect a new INM.

2.3 Fault Tolerance

The system is designed with mechanisms to detect and recover from failures, ensuring that the system remains operational even in the face of component failures and network disruptions. There are two distinct types of failures we want to consider: System component failures and network failures.

System Component Failures All system components may fail at any time due to various issues. Addressing such failures differs from one component to another.

– **Client:** A client that suddenly stops operating does not cause any issues for the rest of the system, hence we don't need to handle it any further
– **INM** INM liveliness is checked through regular heartbeat monitoring from the ISN. In case of a failure to respond, a new election among all idle nodes is started. In order to keep a consistent group view, the group view is replicated from the INM to all ISN.
– **ISN** ISN liveliness is checked through regular heartbeat monitoring from the INM. In case of a failure to respond, the ISN is marked as dead in the group view. ISN do not store any unique data and hence we do not lose any data.
– **AAN** AAN liveliness is checked through regular heartbeat monitoring from the PAN. In case of a failure to respond, the PAN is promoted as to a new AAN and requests a new PAN from the ISN, restoring the replication mechanism. All data from the AAN is replicated to the PAN so we do not lose data.
– **PAN** PAN liveliness is checked through regular heartbeat monitoring from the AAN. In case of a failure to respond, the PAN is marked as dead and the AAN requests a new PAN instead. PAN do not store any unique data and hence we do not lose any data.

Network Failures Any message can get lost at any time in the network. We assume to have control over the network with a synchronous transmission model. Nodes which cannot be reached in the network are considered to be failed and need to be restarted. In order to enhance network resilience, both our unicast and our multicast communication is reliable. This is achieved through positive acknowledgments and retransmission with deduplication of incoming messages.

2.4 Election

Our distributed auction system utilizes the Bully Algorithm to dynamically elect an INM among the Idle Server Nodes(ISN). The election process ensures that there is always exactly one designated INM to manage all other idle nodes, maintaining system stability and coordination. The election process is as follows:

- INM failure detection by ISN.
- Initiating a new election. The ISN with the highest UUID starts a new election process. This choice is made for efficiency purposes, as it reduces the number of competing messages and speeds up the leader election.
- Bully algorithm execution as presented in the lecture.
- The new INM announces itself to the ISN and starts its role.

The bully algorithm was chosen over ring-based election algorithms because of its robustness towards multiple node failures at the same time, where repairing the ring can be quite error-prone.

2.5 Ordered Reliable Multicast

Multicast communication is used in two distinct ways in our project. The first is for dynamic discovery described in 2.2. The second is so the INM can message all INM at once for heartbeat messages or group view updates. This leads to an one-sided communication and it turns out that we do not have requirements for our multicast to be ordered, but only for it to be reliable. In order not to send many unused acknowledgments during dynamic discovery, the reliability aspect of our multicast messages are handled on the application level throughout the various system components instead of the middleware.

3 Architecture design

3.1 Idle Node System

Group view The server nodes create a system based on a group view. The group view holds every servers IP, port and UUID. Because the elements in the group view are not connected in some sort of structure, it is easy to add and remove elements from it. If a node leaves the system, there is only a multicast message from the INM to all the other nodes necessary to update the group view for all nodes.

INM The INM holds no more information than an ISN. This saves us the need for information replication and makes elections faster. We valued this highly, because a potentially high election frequency could interfere with the system's functionality.

3.2 Auction

In a large-scale system, many auctions will be executing at once. It is therefore a high priority to keep every auction pod very light weight. An auction consists of an AAN handling all the communication with the clients and storing auction data like the highest current bidder. The PAN is used for passive replication in case of an AAN failure and replicates all the AAN data. In case either an AAN or a PAN fail, the INM assigns a new node to the auction to always have two nodes per auction.

3.3 Client Server Interaction

The client only saves the address of the AAN of a joined auction. When communicating with the AAN, the client communicates through unicast messages. As a result of this, when sending tasks addressed to the INM, the client sends multicast messages to all the server nodes. This technique is useful, so the clients do not have to be informed about the results of every INM election.

4 Implementation

The project is divided into three main parts. The server implementation, the client implementation and the middleware implementation.

4.1 Middleware Implementation

__init__.py - Communication Basis This file contains the implementation of the Message class. A message has a message type of the type MessageType, content in the form of a string, an IP as string, an integer port and an ID in form of a universally unique identifier(UUID).
Additionally, it holds the MessageType enum. There are 33 values reaching from ELECTION_START and ELECTION_ACK to HEARTBEAT_REQUEST and HEARTBEAT_RESPONSE, to name four of them. These values are the main way to effect the behavior of another node, and primarily depending on them, a server or client sets its handling of a message.
The file also contains a function "get_my_ip()" that returns the IP address of the system that it is called from.

unicast.py - Unicast Socket This class extends the standard socket to perform unicast operations for the application. It changes the operations of message sending and receiving and implements a delivery and a receiving queue.
When sending a message the socket creates a thread that waits for an acknowledgement(ACK) of this message. If it does not receive the ACK after a given timeout, it resends the packet for up to two retries.
The receiving function checks incoming messages for duplicates, informs the send-thread about incoming acknowledgments, and sends out acknowledgements to incoming messages itself. Additionally, it puts incoming messages into the receive queue, to be handled later by the deliver function, that puts received messages into the delivery queue. For testing purposes, the receiving function also has a functionality to simulate network package loss.

multicast.py - Multicast Socket This class extends the standard socket to perform multicast operations for the application. It changes the operations of message sending and receiving and implements a delivery and a receiving queue. The multicast socket itself does not use acknowledgments to ensure reliable message transfer. Instead, message reliability is described in 2.5.
The class implements a "send" function that creates the functionality of sending any message to all participants of the multicast group.
The receive function waits for incoming messages, checks if the message is a duplicate, and, if not, puts it in the receive queue. For testing purposes, the receiving function also has a functionality to simulate network package loss.
The last function in the class is the deliver function that transfers messages from the receive queue into the delivery queue.

4.2 Server Implementation

Initialization Once the constructor is called, the server is being initialized. When initializing a server, the server sets up under a fixed IP address and port. It creates its unicast and multicast sockets. Then it starts an info thread that periodically prints the server status to the console.
After the basic setup, the function "main" is called. In "main" dynamic discovery is done. Additionally, the server starts an election, a heartbeat thread and the listening loop of the uni- and multicast socket. Each of the listening loops are implemented by the "message_parser" function.

__str__ The function is used to get a pretty and human-readable output on str(self) for debugging or logging purposes. The default behavior prints the memory address of the object.

__repr__ Similar to __str__ it pretty-fies the string output. It is used for concatenation with strings such that we can get rid of the explicit str() type conversion.

Message Parser The message parser function, that runs on both, the unicast and the multicast socket, is a loop, that waits for incoming messages to arrive. Depending on the MessageType variable of the message, it handles every message differently. In the following, the behavior, depending on the incoming MessageType value, is described:

UUID_QUERY: Adds the new node to the group view and responds with an UUID_ANSWER message.

ELECTION_START: Only gets handled if not active in an auction. If message does not have its own UUID, the server acknowledges the election. Then the election process gets started and the INM declaration thread gets created and started.

ELECTION_ACK: Only gets handled if not active in an auction. Waits for the INM declaration thread to finish and joins it.

DECLARE_INM: Only gets handled if not active in an auction. Sets the new INM. If the receiving node was INM, and is not the new INM, it sets its status to ISN.

TEST: Simple print out for testing purposes.

CONNECT_REQUEST: If the server is INM, it sends its IP and port to the requester.

STATE_QUERY: Returns a STATE_RESPONSE message containing the requester's UUID and the receiving servers state.

LIST_ITEMS_REQUEST: If the server is the INM, sends back a LIST_ITEMS_RESPONSE message containing the item list.

START_AUCTION_REQUEST: If the server is the INM, it starts an auction for the requested item.

AUCTION_INIT: If the server is an ISN, it becomes an AAN and initializes an auction.

PAN_REQUEST: If the server is the INM, it handles the request to assign a PAN.

PAN_RESPONSE: If the server is an ISN, it becomes a PAN.

PAN_FAILURE: If the server is an AAN, it prints a message about the PAN's failure to the console.

PAN_INFO: If the server is an AAN, it saves the given PAN as its own.

JOIN_AUCTION_REQUEST: If the server is an AAN, and responsible for the sent items auction, it handles the join request.

BID_REQUEST: If the server is the AAN of the bid item, it handles the bid.

REPLICATE_STATE: If the server is a PAN, it saves the given state information.

REPLICATE_STATE_REQUEST: If the server is an AAN, it replicates its state to its PAN.

AUCTION_END: If the server is a PAN, it returns to being an ISN.

HEARTBEAT_REQUEST: The server responds by sending an HEARTBEAT_RESPONSE. If the server is the INM, it updates the sending node's heartbeat value. If the sending node is not yet in the group view, it gets added to it beforehand. If the server is an ISN, it updates the INM's last heartbeat. If the server is an AAN it updates its PAN's heartbeat and if it is a PAN, it updates its AAN's heartbeat.

HEARTBEAT_RESPONSE: The same behavior as for receiving an HEARTBEAT_REQUEST, but without sending a message.

GROUPVIEW_UPDATE: If the server is an ISN, it updates its group view.

Further functions The main function creates a new server.
Every function not yet described implements a functionality of the above-mentioned functions.

4.3 Client Implementation

Initialization When the constructor of the client is called, the initialization begins. The client sets its local variables, creates an unicast and a multicast socket and starts a listening thread that executes the function "listening_for_messages". After the initialization is done, it starts the function "cli" (command line interface) that responds to command line inputs.

CLI The "cli" (command line interface) - function asks the user for input. Depending on the action in the users input it executes different functionalities:

list: Sends a request to the auction system, requesting a list of all the available items (LIST_ITEMS_REQUEST).

start: Sends a request to start an auction regarding a given item (START_AUCTION_REQUEST).

join: Sends a request to join an auction regarding a given item (JOIN_AUCTION_REQUEST).

bid: Sends a request to place a bid to an AAN (BID_REQUEST).

Listening for Messages In the "listening_for_messages" - function, the client continuously waits for incoming messages. Depending on the MessageType variable of the incoming message, it handles it differently:

LIST_ITEMS_RESPONSE: The client parses the incoming message, containing a list of items, into a list format and stores it.

START_AUCTION_RESPONSE: The client stores the incoming AAN connection information.

BID_RESPONSE: The client stores the updated information about the highest bid and highest bidder (can be itself or another client).

AUCTION_END: Prints out highest bidder and leaves the auction.

Further functions The main function checks if the command line - command to create the client is valid. If it is, it creates a new client with the given port. Every function not yet described implements a functionality of the "cli"-function.

5 Discussion and Conclusion

The distributed auction system presented in this report demonstrates a functional and robust platform for real-time auction creation and bidding. Built upon a client-server architecture in Python, the system effectively addresses key challenges in distributed systems, including dynamic host discovery, fault tolerance, and server elections. This section will discuss the strengths and limitations of the implemented system, focusing on the design choices and their implications.

5.1 Client Independence and Middleware Abstraction

A notable design aspect of the system is the clear separation of concerns, particularly the independence of clients from the server infrastructure and the abstraction provided by the middleware. Clients are designed to interact with the auction system solely through message exchanges, without needing to be aware

of the underlying server topology or election processes. This decoupling enhances the system's maintainability and scalability.

Furthermore, the middleware implementation, encompassing the `Message` class and the `unicast.py`, `multicast.py`, and `broadcast.py` modules, effectively abstracts the complexities of network communication from both the server and client logic. This abstraction simplifies the development process, allowing developers to focus on the application-level functionalities rather than low-level socket management. The defined `MessageType` enum provides a structured and extensible way to manage communication primitives within the system.

5.2 Robust Election and Convergence

The implementation of the Bully algorithm for leader election among ISNs contributes significantly to the system's fault tolerance. As designed, the election process ensures that in the event of an INM failure, a new INM is elected from the pool of ISNs. The system is designed to converge to a new INM within a few heartbeat intervals, minimizing downtime and maintaining system availability. This convergence is crucial for ensuring continuous service, particularly in a dynamic distributed environment where node failures are possible. The heartbeat mechanism, integrated with the election process, allows for timely detection of failures and initiation of the election when needed.

5.3 Ordered Reliable Multicast: Basic Implementation and Actual Need

The system incorporates a basic implementation of reliable multicast. However, it is important to note that, upon closer examination of the system's requirements, the need for strict ordered reliable multicast is minimal. The multicast communication is primarily utilized for dynamic discovery, election initiation, and group view updates from the INM to ISNs. In these specific use cases, the order of message delivery and guaranteed reliability are not strictly essential for the system's core functionality. The focus on reliable unicast communication, particularly between clients and AANs, is more critical for ensuring the integrity of bidding and auction state updates. The system's design choice to prioritize robust unicast is therefore well-justified.

5.4 Robust Dynamic Discovery

The dynamic discovery mechanism implemented using multicast messaging proves to be robust. Both clients and servers can seamlessly join the system by broadcasting their presence. The INM's role in responding to discovery requests and managing the group view ensures that new nodes are efficiently integrated into the system. This dynamic discovery capability is crucial for scalability and adaptability, allowing the system to easily accommodate fluctuating numbers of clients and servers without requiring manual configuration or restarts.

5.5 Auction Functionality

The auction functionality implemented in this system is demonstrably functional, allowing for the creation of auctions, bidding, and auction closure. While the core auction logic is complete and operational, it is acknowledged that the system lacks certain features that would be present in a fully realistic auction platform. Notably, the absence of a persistent database for storing auction history, user information, and item details is a simplification. In a production environment, a database would be essential for data persistence, scalability, and providing features such as auction history, user accounts, and item catalogs. The current implementation prioritizes the demonstration of distributed system concepts and core auction mechanics over full-fledged, real-world auction features.

6 Conclusion

In conclusion, the developed distributed auction system successfully meets its design goals, providing a scalable, robust, and functional platform for real-time auctions. The client-server architecture, combined with the middleware abstraction and fault-tolerance mechanisms like the Bully algorithm and heartbeat monitoring, demonstrates distributed system. While certain aspects, such as the ordered reliable multicast implementation and the auction functionality, are basic or simplified, they are appropriate for the project's scope and effectively showcase the core principles of distributed systems. The system's dynamic discovery and election mechanisms contribute to its robustness and adaptability. Future development could focus on incorporating a persistent database, enhancing the auction features to resemble real-world auction platforms more closely, and further optimizing performance and scalability for larger deployments.