Auction System

Group 3: Peter Hösch, Sena Tarpan, and Yun Ye

1 Introduction

Our goal in this project was to design an auction system which enables a single seller to sell a single product to a group of buyers. The buyers can place individual bids on the product. The product will be sold to the buyer who placed the highest bid at a predefined end date for the price of said last bid.

1.1 Project Requirements

Compare with the other distributed system, an auction system supposed to have the characteristic:

1. The customers put greater value on fairness and transparency over the real-time performance on this system.
2. The server when the auction is running is basically stable, while the clients are often volatile.
3. For some privacy reason, the clients shouldn't have a global view(e.g. information of the other participants) of the whole system, but only the necessary information.

So a distributed system should be built to satisfied all those requirements, the system must always maintain a fixed orders of action among all the participants, dealing with very frequent system crashes(or leaving) and sensitive information passing.

2 Architecture design

The seller defines a start and end date for the auction, describes the product as well as a starting bid. The buyers get informed about the the details of the product, the end date and the current highest bid every time said bid changes. Nodes can join or leave the auction at any time. The system should be able to recognize if a buyer goes offline or if a server crashes and take appropriate action. The server should be able handle high amounts of buyers by delegating work to supplemental servers if needed.

The system is implemented as a many servers-many clients design. The servers are the seller, who functions as the main server, as well as supplemental servers that provide fault tolerance and scalability. The main server also functions as a sequencer. The clients, on the other hand, work like a thin client machine that provides merely an interface with a very restricted logic and data

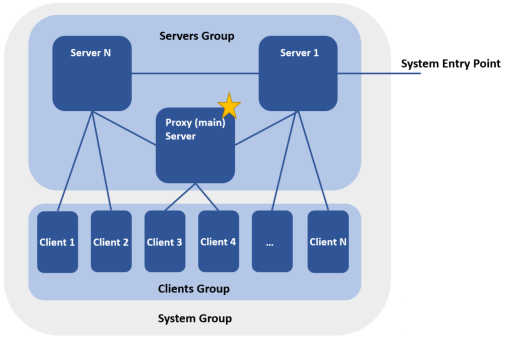

Fig. 1. Architecture diagram

functionality. The supplemental servers exist to take bids, aggregate them and transfer that data to the main server. For this purpose, each server will be connected to a number of clients. The clients only communicate with this server, in the following called their contact server, not directly with the main server or with each other. Bids are placed by using UDP connections from a client to a server.
The servers use UDP with ordered reliable multicast for the host discovery process.

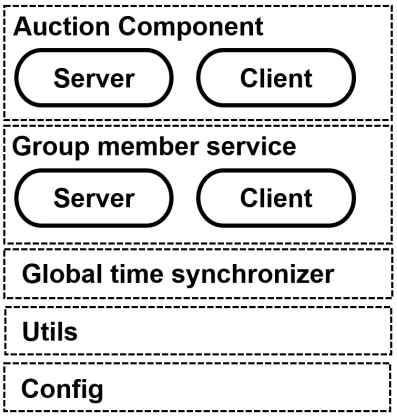

Fig. 2. Object Class in the Implementation

All servers maintain their own data and regularly synchronize with each other. To ensure that we become aware of lost connections or crashes, all participants in the auction send a heartbeat signal to the nodes relevant to them. If

this signal doesn't gets responded to in a certain timeframe, the system assumes that the node that should have sent it is no longer available and proceed to initiate actions to replace them, if possible.
In the case that the main server is no longer available, the other servers will vote for a new main server from among their group. All the servers are organized in a ring structure. The LaLann-Chang-Roberts algorithm. After the election, the information about the new main server will be propagated back to all clients, and the new main server will reorganize the structure of the network.
The servers will use UDP with totally ordered reliable multicast, in addition to the already mentioned for the host discovery process, to inform all interested clients in a raised bid, and to regularly synchronize the current time which is important due to the time critical nature of an auction. If multiple bids with the same value get placed during a short timeframe the system should be able to determine which one was first and as such counts. As mentioned above, the main server serves as the sequencer for this process. When a server gets the information about a raised bid from one of his clients, it will request a sequence number from the main server and then inform all participants of the new highest bid by using a multicast.

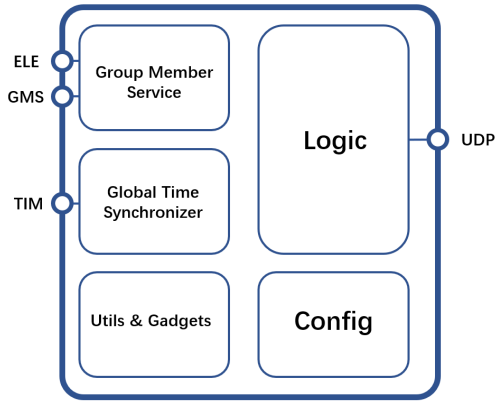

Fig. 3. Process diagram

3 Implementation

This section describes the implementation of the project in it's current form. In case further information about the implementation is required, please refer to our Github repository [1] containing the full code as well as additional content.

[1] https://github.com/Lukasye/ds_project_ws2223

3.1 Dynamic Discovery

The dynamic discovery is triggered automatically when the group member service find out that the server or client is currently not a member of group. It sends out a broadcast to find whether there is a group available.
The recipient can be either a group member or not. If isn't a member of the

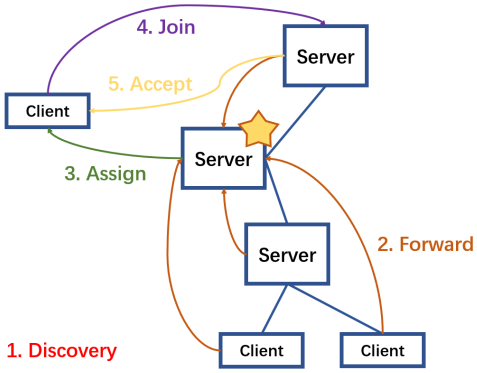

Fig. 4. Dynamic discovery

group, it disregards the message and eventually trigger its own discovery process. On the other hand if it is a member, it forwards the request to the main server, because that is the only node which has the authority to assign a process.The server assigns the process directly or distributes the load, depending on the type of the incoming request and current capacity of the server groups, and replies back with an assign message. The client(server) follows the instruction join the contact(main) server after the process is confirmed with 'accept' message.
In the dynamic discovery phase we prefer the UDP communication without response direct to the sender port. Because it involves message forwarding. The process sends out their contact information(ip, port etc.) and listen to the main incoming port.
For the performance reasons, the clients are distributed equally to all the servers if that is possible. That means the server assigns the new client to the server with the least number of clients. In case that there are new process join the auction in the half way, after the join process, the main server will also send the message with the latest sequence number to trigger a chain reaction of message update in the new process.

3.2 Group Member Service

The group member service is a member class of both servers and clients that keeps track of general connection information. For the clients, this is basically

limited to the question if we're already part of an auction group and if yes, who our contact server is. In the case of servers, the group member service also stores information about all the other servers and clients our server knows about.
The group member service plays an important role when it comes to fault tolerance and recovery. It regularly sends out heartbeats to check if the other nodes of our distributed system are still responsive (clients only send heartbeats to their contact server, servers send heartbeats to all the nodes they know about). In the case that a client looses the connection to it's contact server, it will automatically try to reconnect to the system by sending out discovery messages. Client and server disconnects otherwise don't really need to be handled beyond readjusting the lists of current nodes, unless the connection is lost with the main server. In that case, the election process gets started to determine a new main server, something that is explained in detail inside another subsection down below.

3.3 Global Time Synchronization

In order to enable the clients to know exactly how much time is left until the end of the auction, the clocks of the different nodes in the system need to be synchronized. This service is performed by the global time synchronization, a member class of both servers and clients.
The synchronization is performed in a hierarchical way - the clock of the main server is assumed to be correct, and all the other servers regularly request information about the current time from the main server to set their own clocks. The clients then, also regularly, request information about the current time from their respective contact servers and set their clocks in accordance with said information. The server from which a node gains their data about the current time is called the "synchronization server".
The synchronization operates over it's own port and takes into account the travel time of the messages, by adjusting the received timestamp by half of the time that has passed since the send request was sent. The newly calculated current time is stored in the form of an offset that needs to be added to the raw internal clock data in order to get the actually correct time. This way, the program doesn't interfere with any other applications who use the clock, because no internal clock gets actually changed.

3.4 Elections

We have utilized a ring structure to establish order among servers. This was the initial step for the leader election. The form ring function begins by obtaining a list of servers, which includes the servers' IP addresses and ports. The function then assigns UUIDs to the servers and then sorts them, returning two lists: one containing a sorted list of servers with ports and their UUIDs, and the other containing only a sorted list of servers with ports.

The next step for the election is to acquire neighbors from the ring structure. We have a get neighbor function that takes a sorted server list, which is returned by the form ring function, and a direction boolean. If the direction is true, the

function searches for left neighbors; if it is false, it checks for right neighbors of the given node. The function returns the IP address and port of the next or previous node in the ring based on the value of the boolean variable. If the current node is not found in the ring, the function returns None.

Finally, we have chosen to implement the LaLann-Chang-Roberts (LCR) algorithm for leader election. The function starts by creating a ring and obtaining the neighbor of the current node using the form ring and get neighbor helper functions. The current node then begins sending an election message to the neighbor, and this process continues until a termination signal is received. The election message includes the boolean "IsLeader" value and "Member Id" on which are based on which UUIDs are assigned in the ring form function. The function ultimately returns the leader ID. To provide fault tolerance, we use a heartbeat signal in the LCR algorithm. This allows us to vote for a main server and designate the rest as supplementary servers.

3.5 Reliable Multi-cast

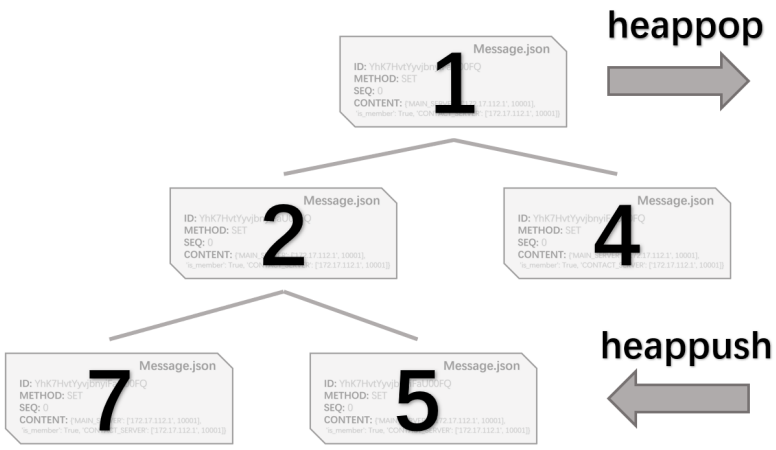

Fig. 5. hold-back-queue

For every message, a sequence number is added. For those who don't need a ordered multi-cast, the sequence number will be default set as 0 and directly delivered to the logic function once received.
If the sequence number is greater than 0, it will be put into the hold-back-queue, which is constructed as a min-heap. The elements in the heap is ordered by the sequence number. Only if the sequence number of the message match the sequence counter of the process, the request can be delivered. The redundant message will be discard and those with greater value will be waited in the queue.

For each process there will be a hold-back-queue checker to determine whether the message in the heap can be delivered and send out negative acknowledgement if it is necessary.

3.6 Fault Tolerance

There are totally three kinds of failures which are expected to occur in the project:

1. Process crash: The Server or Client quit for some reason without notify the others.
2. Message omission: The message loss in the transition.
3. Byzantine Fault: The bid result is miss-read by the Server when a bid is placed.

To deal with those problems we implemented heartbeat to detect the crash of other components, and reliable ordered multi-cast to prevent the message loss. Those will be discussed in separate sections. And for the byzantine fault. We applied the byzantine agreement when there are more than 3 Servers in the group to prevent that there is server with wrong auction information.
In the implementation of the byzantine agreement we applied the phase king

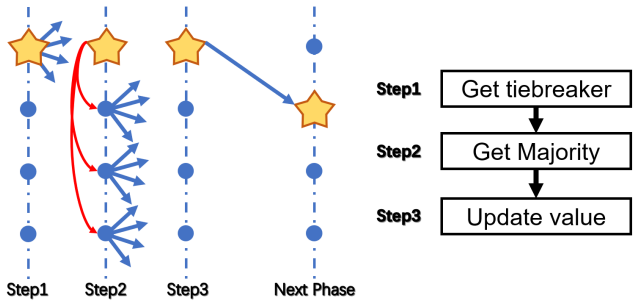

Fig. 6. Recursive phase king algorithm

algorithm, with some slight modify to make use of the infrastructure that is built in this project. The algorithm is implemented in a recursive form. The process that start the agreement will generate a list of servers who are going to be the phase king, what we called king stack. Each time the phase king algorithm is called, an element will be poped out of the stack to be the next phase king. If the stack is already empty, the algorithm will return the results.
Each election phase is divided into 3 steps. First, the phase king will multicast a message to get the tiebreaker value. And then the tiebreaker will be sent along the remote invocation message to all the other processes. After that the new

phase result will be checked by all the members of the agreement groups. Finally the algorithm will be triggered again to go into the next phase.

When the server receive the bid request from the clients, it will see if the initiate requirement is satisfied. If so, a remote method invocation message will be sent out to start the byzantine agreement procedure. A server ask for all the other the value they've got until now and add to a list. After they receive all the response from the others, they choose the majority one as the new value and go on with the following calculation.

3.7 Message Passing

Fig. 7. Message passing example in the system

Dictionary type in python is used in the project as the standard message format. A normal message will includes the identification number of the sender, the request(response) method of the message, a sequence number and the real content(usually will also be a dictionary). And for some special case, like group view synchronisation, a panda data-frame object will be send instead of the regular message format. To enable the sending of complex type of message, data serialisation is needed. In this case we use python pickle package to do this for us. Before the sending of a message, marshalling is applied and afterwards enmashalling for every transaction.
The supported method is stated in the following table:

3.8 Replication Management

There are two types of replication in the system: The server list maintained by each server in the group and the bid history. Because of the totally reliable ordered multi-cast that we applied, the bid history will be consistent is the initial condition are the same.
The server list is mainly used to help the main server balance the number of clients between different servers. Under the that the workload won't suffer from rapidly changes in the auction process, we choose lazy update for the server list.

Table 1. Methods that supported in message passing.

METHOD	description
DISCOVERY	for broadcast dynamic discovery
JOIN	request to join a server
REDIRECT	forward the message to the right consumer
SET	remote parameter setting(now replaced by remote method invocation)
GET	get value from the remote process, also used to do negative acknowledgement
RMI	remote method invocation
BIT	message to raise a bit in the auction
PRINT	print information on remote interface(now replaced by remote method invocation)
PRICE	byzantine agreement, to check the current highest bid
TEST	used for testing the message reach-ability

The server list will be updated each time the heartbeat is received. The signal will carry the information of the current number of clients sent from other server. The main server will summarize all the information and update the server group with the new list. This kind of synchronisation will also be done before the election process to keep the ring consistent.

4 Discussion and Conclusion

This section summarizes our work on this project, it's abilities, limitations, and possible options to counteract said limitations.

4.1 Experiments

We thoroughly tested the application to make sure the varying features function to the point of satisfaction.
One limiting factor we found during said testing is that the eduroam network at the university seems to contain some limiting factor that prevents our nodes from properly making contact with each other - the problem is not reproduceable on any of the other networks we tested our code on. Due to this problem arising from a factor external to our project that we have no control over, there is nothing we can do in order to remove it.
Another problem we encountered is that our code has some parts that rely on specific operating systems, which means that under non-windows or linux operating systems only a reduced amount of features is available. Due to those two having a vast majority of the market share when it comes to personal computers, this was judged to be an acceptable limitation.
Beyond that, the application performs well. The core features (starting an auction, bidding, ending an auction) work, even when using multiple computers or a very high number of nodes. Clients occasionally loose their connection, but can

quickly recover. Servers getting removed, in varying configurations, never damaged the performance of the bigger system beyond a temponary readjustment period.

4.2 Conclusion and Future

In this project we have constructed a functioning multi-server multi-client auction system, which can dynamically form a group in the same network environment and complete an auction process. Variant of faults can be tolerant such as process crash, message omission and simple byzantine fault. The auction process and documentation is consistent for all the participants.