Introduction

A client-server architecture has been developed in order to fulfil the task of creating a working distributed auction system. A client has the task of acting on behalf of the input given by its user. A server has the responsibility of setting up an auction and managing the current highest bid as well as the IP address and port of the bidder. Hereof also the responsibility of reacting to a client's requests which is either a request to bid or a request to receive the current auction's highest bid. The server passively replicates its highest bid and highest bidder to a secondary server in order to improve liability if it crashes.

Use-case guide

The system is required to have setup a primary server before the client can connect. In order to guarantee the system being resilient to the failure-stop failure of a node, a backup server should be created after the primary server and before a client connects.

The primary server is created by running the Server class with the arguments:

<int: port> and <boolean: primaryServer>

The port is the port on which the server is created, and the primaryServer states whether it is a primary server or not. It is here important that the port of the primary server is set to 5000, as the backup server has this reference predefined, and that the boolean is set to true to indicate that it is a primary server. A successful creation will output: "Successfully created the primary server!"

The backup server is created by running the Server class with the arguments

<int: port> and <boolean: primaryServer>

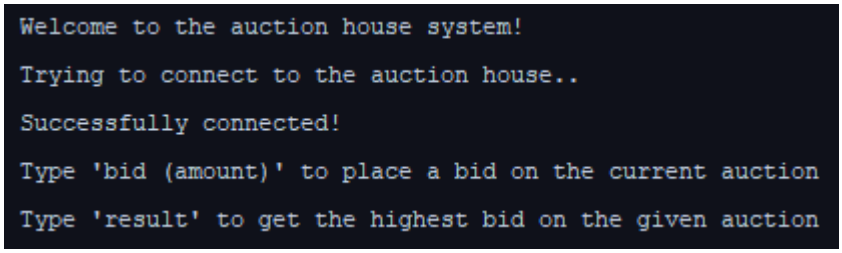
The port must be set to 5001 as it is predefined within the primary server as the port to connect to, and the primaryServer boolean should be set to "false" to indicate that it is a backup server. After having created the backup server, it will output: "Successfully created the backup server!" It then asks for the user through the terminal to type in the IP address of the primary server to which it on a successful input will display:

```
Please write the ip of the primary server to connect to:
192.168.0.58
Successfully created session with /192.168.0.58:62247: 62247
Successfully created session with /127.0.0.1:62248: 62248
```

The client is now ready to connect. The client takes the following 3 parameters:

<String: primary server IP address>, <int: primary server port>, <String: backup server IP address>

Upon the client successfully connecting, the following will be shown:



The client can now either bid on the on-going auction by typing “bid (amount)” or get the result of the auction by typing “result”.

Protocol

The architecture used to define the system is the client-server architecture. The architecture has been chosen as the best fit for the given semantics of the assignment. Hereof every bidder is in this architecture acting as a client with the server functioning as the auction. The need for an auction to have multiple clients bidding on it made client-server the best architecture to fit the semantics. Clients and servers communicate through TCP with an API to further formalize how data is exchanged. More specifically, the API specifies that a client can either make a bid-request or result-request. Upon making a bid-request, a client is required to specify an amount as an integer. If another variable type than integer is specified, the user will be alerted and required to retype the request. Upon making a result-request, it is specified through the API that the client is to expect an integer to be returned, which is the highest bidder of the current auction.

The protocol chosen for communication between the nodes of the system is TCP. The first reason for choosing TCP lies in the security aspect of the protocol. Within the security aspect of TCP is the 3-way handshake which provides a reliable connection between a given client and server. Since the auction system in a real life scenario deals with finances, it is important to secure a reliable connection. Another important aspect of TCP is the congestion control. This ensures that the server (the auction) does not congest, as well as the client. If the server was able to congest I.E. when using UDP, it would result in a loss of IP packets. By having the possibility of losing IP packets, a client could place a bid without knowing whether the response from the server is delayed or if the IP packet was dropped.

However, the just mentioned advantages of TCP does not make up for using TCP instead of UDP in this project. The reliability and message ordering are in this assignment assumed to be guaranteed as stated: “Assume a network that has reliable, ordered message transport, where transmissions to non-failed nodes complete within a known time-limit.” UDP does not by itself guarantee reliable transmission of data, and nor does it guarantee that no congestion happens. However, since reliability and message ordering are said to be guaranteed, the overhead from using TCP is outweighed by the speed of UDP.

UDP in this specific scenario given the assignment’s description as mentioned before is a better choice than TCP. However, the reasoning for still using TCP lies within the fact that UDP cannot guarantee reliable, ordered message transport in a real scenario as these aspects for example is what gives UDP a speed advantage over TCP. First and foremost, UDP does not guarantee ordered messages since every packet is sent independently. The reliability aspect of UDP is also missing, since UDP sends a packet without waiting for a response confirming the receival of the packet. Hence, UDP is a connectionless protocol.

Correctness 1

The correctness of the system is validated through an input-output system between the clients and servers in the system. Upon a request is made, a corresponding response to whether action failed or succeeded will be delivered.

For the server-side aspect of the system, a server will respond with an on-success-message when successfully creating its server socket as follows: “Successfully created the primary / secondary server!”

Furthermore, the server will output a message when a client or another server successfully connects to it in order to make sure the functionality works:

```
Successfully created session with /192.168.0.58:61276: 61276
```

For every request sent to a server from a client, the server do calculations according to the request and reply with a message in correspondence to the result of the calculations. Examples of this are shown further down the page when speaking of the client.

On the final part of the server-side of the system, the primary server passively replicates its highest bid and highest bidder to the backup server. In order to confirm the update on the backup server, the backup server outputs the following:

```
New highest bidder: 192.168.0.58: 61276
Updated highest bid to: 10
```

When a client-process is created it will attempt to connect to the primary-server. If the client successfully connects to the server, the client’s terminal will output an on-success-message as well as a description of the possible actions the client can take as given by the provided API. A client successfully joining a server would look like the following in the terminal on the client-side:

```
Welcome to the auction house system!
Trying to connect to the auction house..
Successfully connected!
Type 'bid (amount)' to place a bid on the current auction
Type 'result' to get the highest bid on the given auction
```

Upon a client placing a bid, the server will respond with a message corresponding to whether the message succeeded or not. The bid-request of a client must contain an integer as the second argument, as stated by the API. The client itself handles the correctness of the request and is asked to re-do the action if giving a wrong input. An example of a client bidding 10 on an auction currently at 0 would result in the following response from the server:

```
Server: Bid accepted, your bid of 10 is now the highest.
```

Upon a client asking for the current highest bid through the result request, the server will again respond with a message corresponding to whether the message succeeded or not. An example of a client requesting for a result of an auction with the current highest bid of 10 results in the following response from the server:

```
Server: The current active auction has the highest bid of: 10
```

However, if a client requests to either place a bid on or receive a result from an auction which has ended due to the time boundary, the following response will be delivered from the server:

```
Server: The auction has already ended with the highest bid of: 10
```

It is here interpreted that a result of an auction is the final highest bid.

Correctness 2

In order to guarantee the system being resilient to the failure-stop failure of a node, replication has been taken into use. The clients of the program connect to a primary server. The primary server has a secondary server which functions as a backup server if the primary server should crash. The primary server uses passive replication to secure the backup server is synced with the current highest bid of the primary server. The overall replication behaviour implements mutual exclusion through the central server system with the Java synchronized method. In the implementation of the auction system, the server has a locking system which permits multiple users form entering the critical section, I.E. the retrieval of the current highest bid of an auction. The server hands out a token if no other client is currently holding it and retrieves it upon a client exiting the critical section.

Furthermore, passive replication is also used to secure the backup server is synced with the same reference to the IP address and port of the current highest bidder. Finally, the primary server has a countdown timer which indicates when the auction has finished. Upon initiating the secondary server, it fetches the current time from the primary server. The time is the result of a variable set to count down on the initialization of the primary server, which starts at 60s. This allows for the auction to still close, should the primary server fail. However, by having the backup server directly fetch the primary server's timer variable, an inconsistency occurs. If the fetch operation is completed fast and requested close to the primary server having just decremented its value, the backup server will finish faster than the primary server. This is due to the backup server starting its decrementing operation immediately and therefore being ahead of the primary server. In order to provide a more consistent solution, the acting of a real auction could be implemented. Hereby to implement an actual global clock time to when a given auction should finish instead of counting down from a specified number. NTP could here be used in order to synchronize the servers accurately to UTC. Hence, it would require a primary server to fetch the time directly from a radio clock. In order to keep the clock crash-resilient, the servers are in a synchronization subnet, in levels named strata. On a server crash, a server goes down 1 stratum and synchronizes. Synchronization would here happen through procedure-call, in order to secure synchronization in a distributed system. Hence, the primary server would accept requests for the current clock reading.

Finally, in order to follow replication transparency for the three above mentioned aspects, clients are not aware of the primary server replicating its data to the replica manager. More specifically, the primary server first updates its own tracking of the highest bid and highest bidder, then updates the backup server, and finally the client is notified with a response to the client's request. This results in increased availability for receiving the current highest bid within the system. Furthermore, crash detection of the server is also detected through the response system. When a client sends a request, it awaits the response of the server before sending further requests and waits 1500ms for the server to respond. If the server has not responded within the given timeframe, it is considered to have crashed. Due to the client only listening after having sent a request, the client will not be notified of a server crash before the client tries to send a crashed server a request. In order to make a seamless recovery for the client, the request sent before the crash detection will be resend after having successfully established a new connection to the backup server. A client notifying a crash by placing a bid as an example, will output the following message:

```
bid 11
Bidding on the current auction with: 11
Server has crashed, attempting to reconnect
Successfully connected!
Server: Bid accepted, your bid of 11 is now the highest.
```

The just mentioned example is if a primary server crashes, which will make the clients connect to the backup server instead. Should a backup server be the one crashing, the clients will not have to reconnect, and the system continues, but now without data replication. Hence, the system is crash-resilient to at most one server crash. In order to counter this short-coming, there could have been implemented a primary server and two replicate managers. (backup servers) which has a leader replica manager. When the primary server crashes, the leader replica manager is chosen as the new primary server. Furthermore, the replica manager not being the leader is now to be the leader and will create a new third replica manager so the same recovery process can take place again. In case of a replica manager crashing, a new replica manager will be created by the leader. If the leader replica manager crashes, the other replica manager will notice, elect itself as the new leader, and create a new replica manger. Finally, each client would have to store a variable with an IP address and port to the leader replica manager, which would be communicated to them from the server noticing the crash and hereby creating the new server. This defines a dynamic system which allows for new replica managers to appear as well as allowing replica managers to crash where they are considered to have left the system. The actual implemented system is defined as a dynamic system but lacks the possibility to create new replica managers during the execution of the program.

In contrast to the scenario in the actual implementation of the system when a client needs to connect to the backup server on a primary server crash, we have the scenario of a new client connecting to the system after a primary server crash has appeared. The new client will attempt to connect to the primary server, should it fail will the client try to connect to the backup server. The backup server IP address is here hardcoded and given as an argument to the client upon creation. The port of the primary server should be 5000 upon creation, and the port of the backup server 5001. An example of a client trying to connect to the system after a primary server crash will output the following:

```
Welcome to the auction house system!
Trying to connect to the auction house..
Attempting to connect to backup server
Successfully connected!
Type 'bid (amount)' to place a bid on the current auction
Type 'result' to get the highest bid on the given auction
```

Having an implementation working as a fully dynamic system as mentioned earlier, would remove the need for the client to specify an IP address for the backup server as arguments, as it would be communicated to it throughout the running of the program.

An account of whether the implementation uses P2P (is it structured or unstructured?), replication, or something else.

The implementation of the distributed auction system uses the client-server architecture. The decision lies within the given semantic of there only being a single auction. Hence, a single server is required where multiple clients can connect. In order to guarantee liability, passive replication is used through a replication manager server being synchronized with the primary server. Due to there only being a single auction, it is limited how much space the server will take up. However, had the system been of a much larger scale, for say, including multiple auctions, would the system architecture of choice change. Having a system managing multiple available servers (auctions) accepting multiple clients would in a large scale result in space issues. In order to prevent this, the P2P architecture could be used instead of the client-server architecture. The P2P architecture takes advantage of the available resources from the multiple participating computers. Furthermore, the objects of the system are replicated into multiple computers in order to further increase performance by distributing the load as well as providing resilience on crashes.

An analysis of the security aspects of the distributed auction system.

The system uses TCP in order to guarantee reliable transmission. This is guaranteed using the 3-way handshake theorem. However, the system is still vulnerable to sequence prediction attack. Hereby a hostile client will try to hi-jack a connection between two nodes. The hostile client will eavesdrop in order to track the traffic between the nodes. Upon a successful result of the eavesdrop, the hostile client will spoof TCP packets in order to make the desired node drop its connection. This could as an example be used on the implemented system to hijack a client's connection to a server. The hostile client could then drop the auction server and keep denying the client requesting to place a bid on an auction through masquerading.

A further security issue within the implementation is denial of service. As the implementation tries to combat race-condition through the use of synchronization on the bid method, a denial of service attack will prevent other clients from accessing the current highest bid.

Finally, in order to help secure the correct recipient and sender between two nodes, encryption can be used. Here, the two nodes establish a secure channel through an exchange of encryption keys. However, this is possible to hijack with a man-in-the-middle attack. Hence, a hostile client uses message tampering to intercept the exchange of encryption keys in order to be able to decrypt later messages between the nodes.