National Technical University of Athens
School of Electrical and Computer Engineering
Academic Year 2014-2015

Advanced Topics in Database Systems:
Auction Server with Two Auctioneers and Multiple Bidders

Authors:
Konstantinos-Nektarios Lianos, <03109054>, <kn_lianos@hotmail.com>
Konstantinos Railis, <03109118>, <krailis@hotmail.com>

1. Introduction

This is an assignment for the course “Advanced Topics in Database Systems”. In this assignment we were asked to implement a simplified auction environment using threads and inter-process communication. We chose TCP sockets for communication and implemented our system in C++.

The auction environment includes two auctioneers and multiple bidders. The auctioneers have items in their possesion which they offer for bidding one by one. The bidders are making bids on items they are interested of. At the end of the bidding for an item the bidder with the highest bid is granted the item. If there is no interested bidder the initial price is reduced and if there is still no interested bidder the item is discarded. The auction for each of the items lasts for an amount of time L.

2. User Documentation

At this point the user is supposed to have a minimum undestanding of auctions and the operation of the auction server. For additional information and clarifications regarding the operation of the auction server we recommend that the user should check the “Components” section of this document. The application has been developed, built and tested in Ubuntu 14.04. For the following it is assumed that the user has installed PostgreSQL. Then user can download PostgreSQL and find more information about it at <www.postgresql.org>.

2. 1. Building the Project

Before we start we should compile the project so that it can be executed on our pc. On Linux open a terminal and change your directory to /advdbproj. Run the “make” command. You will notice that the code is being compiled. The output executables are created in the /bin directory. If you with to compile the code again you may first run “make clean” and then run “make”.

2. 2. Setting Up the Databases

To begin with, the username and password of PostgreSQL must be set to the following:

username = postgres
password = postgres

This action is necessary because the Auctioneer is connecting to the database with this pair of username and password. Afterwards, create two databases “advdb1” and “advdb2”. Then the script *initDB.sh* must be run for the databases to be initialized with the tables we want to have. As you will notice in *initDB.sh* one table is created on each database. A row of the table has the form

ItemID	ItemDescription	CurrentHighBid	CurrentHighBidder

2. 3. Starting the Sychronization Server

After setting up the databases the next thing we should do is to start the Synchronization Server. Change your directory to /advdbproj/bin and execute:

./SyncrServer &

2. 4. Starting the Auctioneers

Now that the Synchronization Server is running we should start the Auctioneers. Change your directory to /advdbproj/bin and execute the command.

./BiddingPlatform <configuration_file>

where the configuration file must have the following format:

<value of L, in sec>
<number of items (N)>
<initial price of item 1><description of item 1>
…

When the auctioneers start they will generate two files, auctioneer1 and auctioneer2 which can be used as a configuration file for the Bidder.

2. 3. Connecting with a Bidder

Change your directory to /advdbproj/bin and execute one of the following commands:

./bidder 'cat /path/to/auctioneer_file' <name>
./bidder <host> <port> <name>

where the host is 127.0.0.1, port is the port the auctioneer is listening for connections and name is the bidder's name, and auctioneer_file is the file generated by the auctioneer.

2. 4. Bidder Commands

The bidder offers the following (and only the following) commands to the user:

- **`i_am_interested :`** the user declares his interest in the current item so that he is included in the bidding process.
- **`bid`** `<amount>` : the user bids an amount for the current item on bid. Only integer values are accepted by the bidder.
- **`list_description`** : it shows the description of the current item
- **`list_high_bid`** : it shows the highest bid for the current item
- **`quit`** : the user quits the auction and the bidder is shutting down

The bidder notifies the user with a message in case of a wrong input command and lists the correct.

2. 5. Executing Scenarios

If you wish you may run **./driver.sh** and watch the execution of some scenarios we wrote for testing the program. More information is included in the ReadMe.txt file.

3. Components

In this part of the documentation we present some information accompanied by diagrams that explain the utility and purpose of each of the components. According to our design the auction server is composed by the *Data Base Layer*, the *Communication Service*, the *Synchronization Server*, the *Auctioneer* and the *Bidder*.

3. 1. The Data Base Layer

The Data Base Layer is a component that is included in the Auctioneer and offers the API that the auctioneer needs in order to connect and make updates to its corresponding database. The Data Base Layer's role is to let the auctioneer connect to the DMBS, initialize the database and make any updates that are needed to it. In other words, the auctioneer accesses its corresponding database through the Data Base Layer.

We should hereby mention that the DBMS we are using is PostgreSQL. In a separate bash script (initDB.sh) we initialize the databases that the auctioneers are going to connect to. After the databases' initialization and after the auctioneers' execution begins they are able to use the DBlayer so as to connect to the databases and add the necessary information about the items including item descriptions, initial prices etc.

We can see the way an auctioneer accesses its corresponding database through the Data Base Layer in the following diagram.

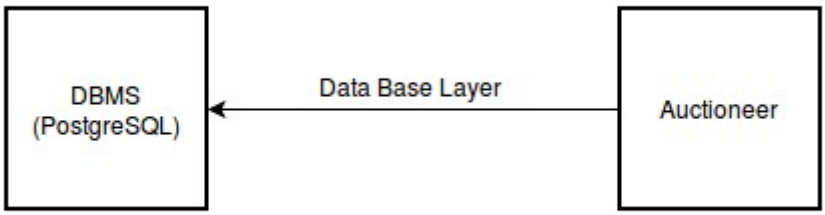

Figure 2.1. *The Auctioneer accessing its corresponding database through the Data Base Layer*

3. 2. The Communication Service

The Communication Service is a component that is included in the Synchronization Server, the Auctioneer and the Bidder and, as its name implies, offers the necessary API for the communication between the Synchronization Server and the Auctioneer, and between the Auctioneer and the Bidder. Although it was mentioned in the introduction we should state again that we used TCP sockets for the inter-process communication.

3. 3. The Synchronization Server

The Synchronization Server is the component which, as its name implies, is responsible for the synchronization of the two auctioneers. The synchronization server does not communicate with any

other component except for the auctioneers.

The Synchronization Server begins its operation before any other component. When the Synchronization Server begins its operation it listens for incoming connections from the two auctioneers. When the connection is established each of the auctioneers can start its execution.

The messages that the Synchronization Server receives from the auctioneer might concern a connection of a new bidder, a new high bid, a new item that is up for bids, the winner of an item, the end of the auction etc. When the Synchronization Server receives such a message it should notify the other auctioneer in an appropriate way. For instance, if the Synchronization Server receives a new high bid message from Auctioneer 1 it should notify Auctioneer 2 about the new high bid.

It is clear that the Synchronization Server has the leading role in the operation of the auction server as a whole. In the following diagram we can see the auction server composed by the databases, the Synchronization Server and two Auctioneers. We should notice that the communication between the Synchronization Server and the Auctioneers is made through the Communication Service.

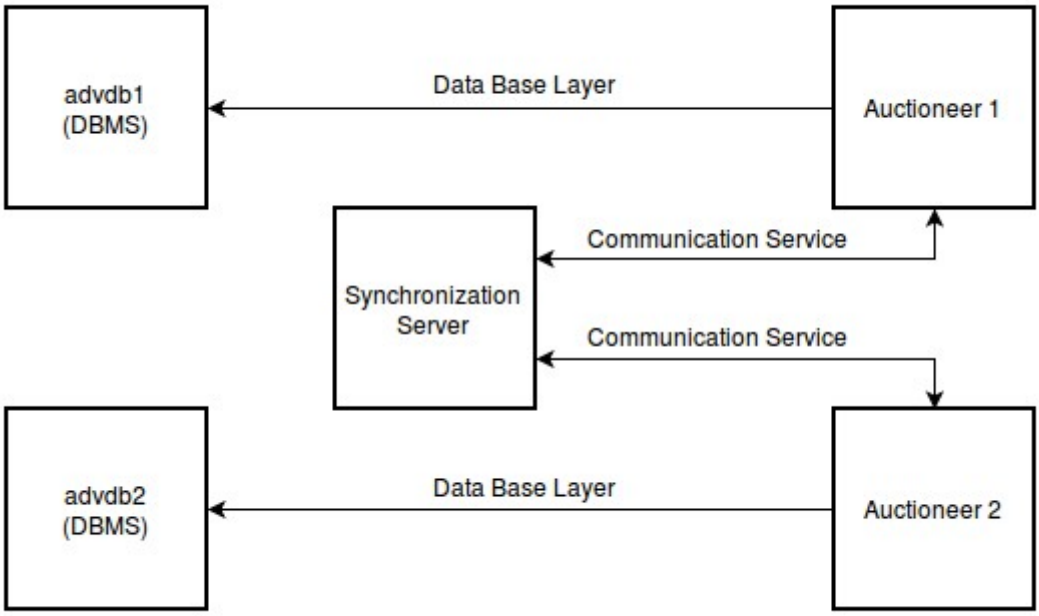

Figure. *The auction server composed by the databases, the Synch Server and two Auctioneers*

3. 4. The Auctioneer

The auctioneer is the component of the auction server that runs the auctions. The auctioneer is implemented by the class Auctioneer which will be described later. After establishing the data bases each auctioneer is connected to one of them. The auctioneer is given a configuration file including the items available for bidding, the time period (L) for which an item is on bid and the number of items (N).

Each Auctioneer connects on and communicates with the Synchronization Server. The Auctioneer sends messages to and is notified by the Synchronization Server (some of those messages have been mentioned before). After the connection the Auctioneer is listening for connections from the bidders. The bidders choose whether they will connect on Auctioneer 1 or Auctioneer 2. Since both auctioneers possess the same configuration file, the item list is the same on both of them.

When the bidders have connected it is time that the auction starts. The auctioneers offer the items on their possession for bidding one by one. They receive I_AM_INTERESTED messages from the bidders that are interested on an item. The interested bidders send their bids or a QUIT message if they wish not to participate in the auction anymore. When (L) seconds have passed and the auctioneer no longer accepts bids for a specific item the winner is anounced. During the bidding for an item the auctioneer communicates with the Synchronization Server.

We should mention again that the Synchronization Server makes most of the critical decisions during the bidding for an item like the winner, the high bid etc.

3. 5. The Bidder

The Bidder is the component that communicates the user's request to the auctioneer. The user executes the bidder with three argmuments which include the name of the host (auct1 or auct2), the port that the host is listening for connections and messages, and the bidder's name.

Once the bidder is executed it tries to connect to the auctioneer through the port that is provided. When the connection is established the bidder sends a CONNECT message to the auctioneer including the bidder's name. Since the bidder is registered by the auctioneer it is capable of receiving messages from and sending messages to the auctioneer.

The bidder offers a number of commands to the user as an interface to the auction server. Specifically the user can `bid amount` for an item, `list_description` of an item, `list_high_bid` for an item and `quit` the auction.

The bidder receives messages from the user and forwards them to the auctioneer in the appropriate format. It gets notifications for a new item that is on bid, the winner of the bidding proccess for an item, the new high bids, the end of the auction etc.

Following is a diagram of the auction server including the database, the synchronization server, the auctioneers and some bidders. As mentioned before the bidder is using the Communication Service for sending and receiving messages.

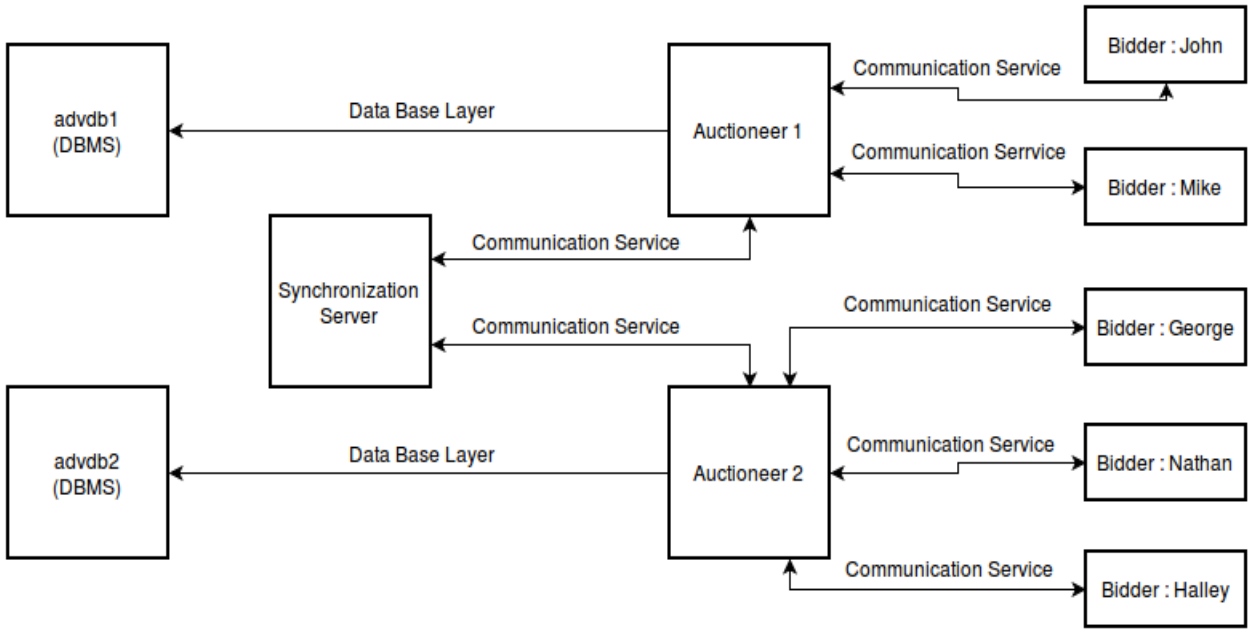

Figure. *The whole system including the most significant of its components*

3. Classes Description

In this part of the documentation we discuss each of the classes separately and we briefly explain the utilities of some of the most important fields and methods.

3. 1. DBlayer

The class Dblayer is a class that is included as a component in the class Auctioneer. Each of the Auctioneers make updates to their corresponding data base through the methods of this class. Some of the class's fields and methods are:

- **`void`** `connect (`**`void`**`)` : lets the aucitoneer connect to its corresponding database
- **`bool`** `insertItem(`**`int, string, int`**`)` : it inserts an new item in the database. The arguments include the item's ID, the item's description and the item's initial price.
- **`bool`** `updateItem(`**`int, string, int`**`)` : updates the last saved high bid for an item. The arguments include the item ID, the bidder's name and the new high bid.
- **`void`** `updateItem(`**`int`**`)` : reduces the initial price of an item if no bid has been made. The arguments include the item's ID.
- **`std::string`** `dbname` : the database's name

3. 2. AbstractServer

The class AbstractServer is an interface which is implemented by the classes SynchServer, Auctioneer and Bidder which will be examined further later. AbstractServer includes the following fields and methods:

- **`int`** `getListeningPort(`**`void`**`)` : returns the port at which the AbstractServer (SynchServer : AbstractServer, Auctioneer : AbstractServer) is listening for incoming socket connections or the port at which the AbstractServer (Bidder : AbstractServer) is connected.
- **`int`** `handleMessage(`**`std::string, int`**`)` : handles a message received by the AbstractServer. The arguments include the message as a string and the socket on which the message was received. The SynchServer handles messages sent from an Auctioneer, the Auctioneer handles messages sent from the SynchServer and the Bidder, and, finally, the Bidder handles messages sent from the Auctioneer.
- **`std::string`** `createMessage(`**`int, int`**`)` : creates a message to be sent from an AbstractServer to another. The arguments include the code of the message to be sent and the item id. Similarly with the `handleMessage()` method the SynchServer sends messages to the Auctioneer, the Bidder sends messages to the Auctioneer, and the Auctioneer sends messages to both the SynchServer and the Bidder.
- **`class`** `CommunicationService *comm` : a component of the CommunicationService class which all the AbstractServers are using. The class

CommunicationService will be discussed further later.

- `logger* Logger` : a component of the logger class which is basically used for creating log files for the auctions in order to track their execution.

3. 3. SynchServer

The class SynchServer implements the class AbstractServer. The most important fields and methods that the class includes are the following:

- **`bool`** `adduser(`**`std::string`**`)` : adds a new user (bidder) to the vector *allusers*. The argument is the new user's name.

- **`void`** `updateAuctioneers(`**`std::string`**`)` : sends a message to all the auctioneers. The argument is the message to be sent.

- **`bool`** `setMax(`**`std::string, int`**`)` : checks if a new bid is the highest and if it is it renews the high bid and the bidder. The arguments include the bidders name and the bid.

- **`std::string`** `maxBidder` : the highest bid's bidder name

- **`std::vector<int>`** `auctSockets` : the sockets used for communication with the auctioneers

- **`std::vector<std::string>`** `allusers` : names of the currently connected users

- **`int`** `activeItemID` : the active item's ID

- **`int`** `finishedAuctions` : the number of the finished auctions

The *getListeningPort()*, *createMessage()* and *handleMessage()* methods are included in the description of the AbstractServer class.

3. 4. CommunicationService

The CommunicationService class offers the API for the connection and message-passing between the SynchServer and the Auctioneer, or between the Auctioneer and the Bidder. The most significant methods and fields of the class are the following:

- **int** sendMessage(**std::string**, **int**) : sends a message through a socket. The arguments include the message as a string and the socket's file descriptor.

- **int** listenClients(**int**) : accepts messages from a client. The argument includes the number of seconds that the server will be waiting for messages.

- **int** listenSocket(**int**) : it listens on a specific socket. The argument is the socket's file descriptor

- **int** connectToServer(**int**) : connects to a server and returns the socket's file descriptor. The argument is the port at which the server is listening for connections.

- **std::vector**<**int**> incomingConnSetup(int,int) : accept connections in the server's listening port . The arguments include a timeout period for which the server is listening for connections and a desired number of clients.

- **std::vector**<**int**> ConnectionsList : a vector of all sockets that are alive

3. 5. Auctioneer & Bidder

The classes Auctioneer and Bidder implement the AbstractServer class so they implement its methods and have some additional fields and methods that we considered redundant to be mentioned. It is obvious that the *handleMessage()* method, for instance, handles messages from the auctioneer in the class Bidder and messages from both the bidder and the synchronization server in the class Auctioneer. The same situation occurs with some other common methods.

3. 6. Conclusion

Some brief conclusions and significant facts concerning the structure of the project are the following:

- The class AbstractServer is implemented by the classes SyncrServer, Auctioneer & Bidder.
- The class AbstractServer has a CommunicationService component which is the channel for the Communication between the SyncrServer and Auctioneer and the Auctioneer and Bidder.
- The class Dblayer is a component of the class Auctioneer and offers the API for the auctioneer to access its corresponding database.