Advanced Topics in Data Bases (2014 - 2015)
Course Term Project
Due Friday, April 3, 2015

Introduction

In this assignment, you will implement a simplified auction environment using threads and inter-process communication (e.g. TCP sockets). You can implement your system in any language of your choice but it may be more straightforward if you use Java or, C++.

Please keep in mind that you need to run two separate Data Bases (one for each auctioneer), and the only information the two auctioneers exchange are the details of the current item value and the corresponding bidder. The auctioneers can run on the same process in two different threads.

Auctions are sales in which no fixed price is set for an item. Instead, an auctioneer receives bids on an item. Bids must be strictly increasing; that is, if the last bid was $25, the next one must be greater than $25. The only exception to this is the initial amount, which may be bid by anybody. Note bids are integer values. The auctioneer continues to receive bids from parties interested in the item until a fixed amount of time (L) has elapsed since the last bid. Whomever makes the last bid pays that amount for the item.

System Requirements: Conceptual

The auction environment will be composed of a *auctioneer* and a number of *bidders*.

The Auctioneer

The auctioneer performs two functions: it handles the connection/disconnection of the bidders, and it handles the bidding. A bidder must first connect to the auctioneer before being allowed to participate in the auction. Connecting is done by setting up a socket connection to the auctioneer and sending a *connect* control message to this server which contains the bidder's name as a parameter. Each such connection, and its associated bidder name are stored by the auctioneer in a registration table.

Bidding is handled first by sending a *new_item* control message to each connected bidder. This message tells the bidder what item is up for bid (item id), along with its initial price and an ASCII description of it. If a bidder is interested, it is then expected to notify the auctioneer of its interest with an *i_am_interested* control message. If a bidder has not responded with an *i_am_interested* control message within L, they are assumed to be uninterested and are left out of the bidding for that item. If no bidders are interested in the item, then the item is discarded and the next item is then used.

The auctioneer then starts the auction by sending a *start_bidding* control message which contains the initial price of the item to each interested bidder. It takes bids as they come; if the bid is greater than the current highest bid, that bid becomes the highest bid and all bidders are notified of this change, along with the holder of the new highest bid, with a *new_high_bid* message. (It is possible to receive a bid which is based on out-of-date information; if a bid is less than or equal to the current

high bid, it is not accepted.) Once L time has elapsed since the last high bid, the auctioneer sends a *stop_bidding* control message to each interested bidder, and awards the item to the highest bidder. All subsequent bids for that item (as identified by its item id) are ignored. (Note that this timeout can be done with the `select()` command; see its man page for more details.)

This process is repeated for each item in the auction; once all items are gone the auctioneer sends an *auction_complete* control message to all connected bidders and terminates. Bidders must then terminate gracefully.

There is one exception to the above process; if nobody bids on the item initially within L, then the initial price of the item is dropped by 10% and the *new_high_bid* message is sent out with the special bid holder `no_holder`. This is repeated until some bidder does bid on the item, at which point the process continues normally, or there have been five rounds with no bidding, at which point the item is discarded and the bidding for that item terminates with the *stop_bidding* control message.

Once the bidding for a particular item has started, no new bidders may join the bidding. Instead, the auctioneer places any new bidders into the registration table, but marks them as non-participating in the current round of the auction (ie: until the *stop_bidding* message has been sent).

A bidder is considered *disconnected* from the auctioneer when a *quit* control message is received from the bidder, or when the auctioneer notices that the socket connection has been terminated. In both cases, the entry corresponding to the bidder is removed from the registration table. Note that the bidder should be able to disconnect at any time; the auctioneer hence must be able to update its registration table for any bidder at any time.

The Bidder

The bidder is responsible for the remainder of the system's functionality, which includes providing a command-line interface to the user which reports the item up for bids and the current bid, and allows the user to make bids on the item.

Each bidder has a name associated with it. This name is specified on the UNIX command line when the bidder is started. Each successful bid is relayed to all bidders with this name (and item id), thereby notifying each bidder what the current high bid is and who has it. (One can easily conceive of circumstances in which the holder of the highest bid would wish to remain anonymous. This, and other security issues, do not need to be addressed. However, you are expected to perform basic error checking (such as detecting if two bidders have the same name) and dealing with error cases appropriately.)

Each time a new item comes up for bids, the auctioneer will tell the bidder what item is up for bids and what its initial price is. The bidder relays this information to the user, and asks if the user is interested in that item. If the user is, then the bidder sends an *i_am_interested* control message to the auctioneer.

Once bidding has begun, the bidder should allow the user to make a bid at any time using the `bid` command. The bidder will pass this bid on to the auctioneer, who will then decide whether the bid is accepted or not. The bidder will be given this information, and should then pass it on the user. Only bids higher than the current high bid should be allowed by the bidder; for example, if the current high bid is \$100, the bidder should not allow the user to make a bid of \$90. (Note that it is possible for the current high bid information to be out of date; if this is the case, then the auctioneer will simply not accept the bid.)

System Implementation Related Specifications

The Auctioneer

You will have two auctioneers offering the same list of products. Different clients connect with one or the other acutioneer. Once a client has connected with an auctioneer it stays for the duration of the whole process with this auctioneer. The problem that arises in such an environment is that two clients may bid for the same item in diferrent auctioneers and therefore the auctioneers have to be synchronized, exchanging the latest bidding values they have seen. For simplicity you can run the two auctioneers in the same process in two threads. In this way you will avoid the necessity of interprocess communication between the two auctioneers.

Each auctioneer can handle a different pool (set) of bidders that can be pre-allocated in advance by the driving process (driving client) and a configuration file that will provide as argument how many bidder you will have, in which auctioneer will be allocated, and how fast (or the frequency) they can bid, and what bidding values each bidder will bid for, so that the system can run automatically, without user intervention requiring manual bidding. Make sure you log each bidding value to a log file so we can then trace during the presentation of your project how the bidding process proceeded.

The exchange and synchronization of auctioneers can be implemented by an algorithm of your choice (e.g. something like a gossip type of algorithm) every time there is a price change or, lazy in the sense that the auctioneers re-synchronize before they announce the winner (lesss accurate approach). Keep in mind that the two auctioneers may not start at exactly the same time so there may be a time delay before one auctioneer reaches its time limit compared to the time the other auctioneer reaches its time limit for the same item. Each auctioneer has to give to its bidders only the preallocated time and not more. In this respect, if for one auctioneer the time runs out for an item, then before it announces a winner or moves to the next item, it has to wait until the other auctioneer finishes, and the two auctioneers synchronize so that they announce the correct winner.

The auctioneers must be run first. For each auctioneer, its service, `auct`, takes a single argument: a file which contains the number of items up for bids, their initial prices, and descriptions of the said items, in the following format:

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

The initial price of item n is an integer separated by its description by any amount of white space. The description of item n consists of the string starting with first non-whitespace character after the initial price and terminating with the last character before the newline. For example, in the following file:

```
10000
3
100 A ceramic bust of Lester B. Pearson.
50000     A 1974 candy-apple-red  Corvette.
5      A peanut-butter sandwich found in John F. Kennedy's desk.
```

item 2 has the initial price $50,000 and the description "`A 1974 candy-apple-red  Corvette.`"

Note that it is up to your system to determine item id's for the items, in order to ensure, among other possible error checks, that bids are applied to the correct items. An obvious method is to use the ordinality of the item in the auction, ie: the first item is given the item id "1", the second one is given "2", and so on. This is sufficient for the purposes of this assignment, and item ids should be transparent to the user anyway. However, you may implement some other scheme if you wish.

`auct` must provide some way for the bidders to connect to it. It does this by establishing a listening socket and publishing its address at a well-known location. You will implement this by having the server create a file named "`auct_name`" in the current directory. This file will contain a single line: "*host port*," where *host* is the name of the host on which `auct` is running and *port* is the port number of the socket it is listening on.

`auct` maintains a registration table which associates socket id's (ie: file descriptors), and bidder names. When `auct` detects a connection attempt (using the `select()` command), it completes the connection establishment and creates a new entry in the table; the first thing a bidder will do is send a *connect* control message with its bidder name, and `auct` will enter this bidder name into the registration table. (Note that `auct` must accept the connections as they come in, due to system limits which prevent any more than five connections pending. However, if bidding is currently going on these new connections may not participate until the next round.) There may be any number of bidders attached to `auct` at any given time; you should be able to alter the size of your registration table appropriately.

In your implementation, `auct` must be able to handle *at least* the following control messages:

`connect:` Bidder wishes to connect to the auctioneer. Arguments include name.

`i_am_interested:` Bidder wishes to participate in the auction for the specified item. Arguments include the item id of the item in question.

`my_bid:` Bidder tenders a bid to the auctioneer. Arguments include the amount of the bid and the item id of the item in question.

`quit:` Bidder wishes to disconnect; connection termination will follow.

Each bidder must handle *at least* the following control messages:

`bid_item:` Auctioneer specifies item description and initial price to connected bidders. Arguments include initial price, item id, and item description.

`start_bidding:` Auctioneer accepts bids from bidders. Arguments include the item id of the item in question.

`new_high_bid`: Auctioneer specifies new high bid. Arguments include the amount of the bid, the holder of the bid, and the item id of the item in question. Holder of the bid may be the special value `no_holder` to indicate that nobody has made a bid on the initial price.

`stop_bidding`: Auctioneer no longer accepts bids from bidders. Arguments include the winner of the item up for bids and the item's id; if the winner's name is the same as the bidder's, then the bidder should announce this to the user. Winner may be the special value `no_holder` to indicate that nobody won the item.

`auction_complete`: Auction is finished, and bidder should be shut down.

`duplicate_name`: Auctioneer has received a registration request with a duplicate name. Bidder should shut down with an error message.

You may implement additional control messages if you wish.

The Bidder

A variable number of bidders can be run once the auctioneer is running. Its executable, `bidder`, takes three arguments: the host on which `auct` is running, the socket port associated with `auct`, and a name to associate with the bidder. The former two arguments are best extracted from the `auct_name` file like the following:

```
unix_prompt-> bidder ‘cat auct_name‘ mike
```

Note the backquotes used above; this tells the UNIX system to replace whatever is between the backquotes with the result of the operation specified (such as `cat auct_name`), then execute the command that way. Note further that the name may *not* be `no_holder`.

`bidder` will start by attempting to connect to the auctioneer at the address provided. If it is successful in establishing a connection, it will send a *connect* control message containing its name. It will then report its success to the user and start its command interface.

The user commands that the bidder should provide are described below. Each command must be on a single line and all parameters are separated by whitespace. In the descriptions, portions in `courier font` are to be typed verbatim, while portions in *italics* are to be replaced by appropriate user-specified information.

`bid` *amount*: Send a bid to the auctioneer in the amount given. If the amount is less than the current high bid, reject the bid without sending it to the auctioneer. This will result in a *my_bid* control message being sent to the auctioneer.

`list_high_bid`: List the current high bid (which should have been saved from the last *new_high_bid* control message).

`list_description`: List the description of the item (which should have been saved from the last *new_item* control message).

`quit`: Disconnect the bidder from the auctioneer. This will result in a *quit* control message being sent to the auctioneer.

Each `bidder` should keep track of which items were successfully bought by it, and report this to the user either when quitting, or when the auction is over.

Notes

- For the purposes of this assignment, you may opt for all of the `bidder`s be running on the same computer architecture as the `auct` process, so that you don't have to worry about issues like byte ordering, word size, or data type sizes. However, your implementation should work correctly regardless of on which machines the auctioneer and any of the bidders are run.
- You will need a "driver" component (e.g. a driver-server) who will initiate the process and run the auction (as in a script). Such a configuration file will contain information on how many bidders in total the session will have, which bidder is allocated to which auctioneer, in what frequency the bidders bid, and what are the values of the bids of each bidder in sequence (so these can be executed as in a script). This configuration file is separate form the one discussed above rerarding the list of the offered items.
- You may need a synchronization server that implements the logic of synchronizing the item values as the bidding evolves in the data bases of the two auctioneers.
- It is mandatory you use good software design prtactices, use of design patterns (e.g. singleton, proxy, adapter, factory method, strategy, observer) when and where you deem appropriate.
- The transaction management at the lowest level can be handled by the DBMS you will be using (e.g. Postgress, MySQL etc.).

Assignment Submission

Submit electronically your source code, basic instructions how to compile/run it, and its executables at the course site at mycourses.ntua.gr

User documentation should contain everything an arbitrary user will need to know to use this program. System documentation should contain all your design decisions, such as message and table details.