From Verteilte Systeme UE, 184.167

Lab1: Lab1

General Remarks

- We suggest reading the following tutorials before you start implementing
  - **Networking Basics**: Short explanation of networking basics like TCP, UDP and ports.
  - **Java IO Tutorial**: It's absolutely necessary to be familiar with I/O and streams to do sockets operations!
  - **Java TCP Sockets Tutorial**: A very useful introduction to (TCP) sockets programming.
  - **Java Datagrams Tutorial**: Provides all the information you need to send and receive datagram packets using UDP.
  - **Java Concurrency Tutorial**: A tutorial about concurrency that covers threads, thread-pools and synchronization.
- Group work is **NOT** allowed in this lab. You have to work alone. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone. Note that we may perform routine checks using an anti-plagiarism software which detects identical or very similar solutions.
- Be sure to check the **Hints & Tricky parts** section for questions!

Submission Guide

Part A

- You have to successfully complete the **Part A** of this assignment until **13.10.2012, 18:00 CET** - the deadline is hard!
- Part A is the **mandatory** step of the registration process for this course.
- You do not have to upload or submit your solution for the Part A, but you should try to reuse it for solving the Part B of this lab.
- You will be rewarded 3 points after completing the Part A.

Submission

- You must upload your solution for whole assignment (i.e., **Part B**) using the **Teaching Tool** before the submission deadline: **31.10.2012, 18:00 CET** - the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Your solution for whole assignment has to be implemented in **Java 1.6** (i.e., it has to compile/run with a JDK version 1.6) - any later version of Java (including 1.7) is not allowed and can result in you receiving 0 points for this assignment!
- Do not confuse our lab servers (e.g., `six.dslab.tuwien.ac.at`) with the **Teaching Tool**. The lab servers are just for testing purposes. We cannot grade any solutions uploaded there.
- Upload your solution as a **ZIP** file. Please submit only the **sources** of your solution and the **build.xml** file (not the compiled class files and no third-party libraries).
- Your submission must compile and run in our lab environment. Use and complete the provided **ant template**.
- Test your solution extensively in our lab environment. (It'll be worth the time!)
- Before the submission deadline, you can upload your solution as often as you like. Note that any existing submission will be **replaced** by uploading a new one.
- Please make sure that your upload was successful (i.e., you should be able to download your solution - as the tutors will do during the interview).

Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot to the interviews using the **Teaching Tool**.
- You can do the interview only if you submitted your solution before the deadline!

- The interview will take place in the **DSLab**. During the interview, you will be asked about solution that you uploaded (i.e., **changes after the deadline will not be taken into account!**). In the interview you need to explain your code, design and architecture in detail.
- Remember that you can do the interview **only once**!

Important: Please note that Lab 2 and Lab 3 will extend your Lab 1 solution. That means that it will pay to implement your solution in an extensible way (just like you would build 'real' software).

Description

In this assignment you will learn
- the basics of TCP and UDP communication using sockets
- how to program multithreaded applications and synchronization
- how to design a client-server distributed applications

Overview

In this year's first assignment, you have to implement a simple auction system where multiple users bid in auctions similarly to eBay. More formally, the system will use a variant of the **English auction** type with only difference that an auction ends at specific time and date that is determined by the seller when the auction is created. In an English auction, all bids are open and the current highest bid is visible to all participants. Moreover, the current selling price is determined by the highest bid (and not the second highest bid like at eBay). An example of an auction is shown in Figure 1:

1. First, Alice creates an auction for her small notebook computer.
2. Then, Bob bids and sets the first selling price with a 100€ bid.
3. Later, Carl overbids Bob with a 200€ bid and sets a new selling price.
4. At last, Dave overbids Carl and sets the final selling price with a 250€ bid.

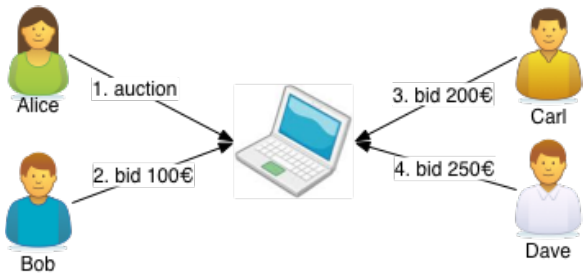

Figure 1

Design

The auction system consists of one server and multiple clients connected to it. Bidders and sellers use clients to create, list, and bid on auctions. The server coordinates execution of all commands that were received from the clients (i.e., clients are not allowed to communicate directly between each other). Because commands require proof and confirmation from the server that they have been received and executed, they must be sent reliably using TCP protocol. Moreover, the server will asynchronously notify clients with some notifications (discussed in detail further below) that are of interest to the currently logged-in user. These notifications are not core to the functionality of the server and are allowed to get lost. Thus, they will be sent unreliably from the server to client using UDP protocol.

In short, the task of the client is to allow its user to issue commands to the auction server and to display notifications from the server as they arrive. Please note that users are not anyhow uniquely connected to a client and can use different clients at different times; i.e. they can use one client and then, in some later time, after logging-out from the first client use another client by logging-in with same credentials. Moreover, the server should reject any log-in command from user that is already logged-in on a different client. Figure 2 shows an example network overview with Alice, Bob, and an anonymous user connected to the server using three different clients.

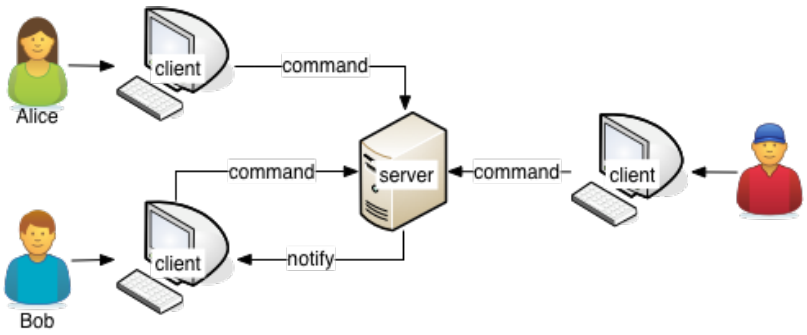

Figure 2

After starting a client, an anonymous user (i.e., user that is not logged in) is allowed to list all currently open auctions (i.e., auction that are still not finished) with each auction displayed in a separate row. For each auction in the list, the client needs to show its identification number (abbreviated as "id" in the rest of this document), description, name of the owner, date and time when auction will end, the current highest bid amount (or 0 if no bid has been placed yet) and the name of the current highest bidder (or "none" if no bid has been placed yet). An example of listing is shown in Figure 3.

```
> !list
1. 'Apple I' wozniak 10.10.2012 21:00 CET 10000.00 gates
2. 'My left shoe' 07.10.2012 12:00 CET 0.00 none
```

Figure 3

For bidding and creating auctions, the user has to first log into the server using a client by specifying its name. After that, the user is allowed to create an auction by specifying the duration of the auction in seconds and its description of the auction. The task of the server is to assign the new auction with a unique id number, which will be used throughout the bidding process. An example of logging in and creation of an auction is show in Figure 4.

```
> !login alice
Successfully logged in as alice!
alice> !create 25200 Super small notebook
An auction 'Super small notebook' with id 3 has been created and will end on 04.10.2012 18:00 CET.
```

Figure 4

Additionally, the logged in user can bid by specifying the auction's id number and a bidding amount. If the bid is higher than the previous highest bid, the bidder becomes the new highest bidder and the amount is set as the new selling price for this auction. Moreover, the previous highest bidder gets notified that he/she has been overbid. An example of the bidding process is show in Figure 5 (note that the output is displayed in a single block, while in fact it represents different terminal windows for "dave" and "carl").

```
carl> !bid 3 200
You successfully bid with 200.00 on 'Super small notebook'.

dave> !bid 3 250.00
You successfully bid with 250.00 on 'Super small notebook'.

carl> You have been overbid on 'Super small notebook'.
```

Figure 5

When an auction ends, the server is responsible for notifying the owner of the auction and its highest bidder that the auction has ended. A notification example is show in Figure 6.

```
alice> The auction 'Super small notebook' has ended. dave won with 250.00.

dave> The auction 'Super small notebook' has ended. You won with 250.00!
```

Figure 6

Furthermore, the user is allowed to log out from its account and then the client can be reused to log in as another user. After logging in, the server is responsible with notifying the logged in user with all pending notifications (i.e., notifications that were created for the user by the server while user was not logged in anywhere). An example of this process is shown in Figure 7.

```
dave> !logout
Successfully logged out as dave!
> !login bob
bob> You have been overbid on 'Super small notebook'.
```

Figure 7

Please note that these figures are for illustration purposes only. Your output should resemble the illustrated output as much as possible (i.e., some small formatting differences are allowed, but all mentioned information in the output **must** be present except for the notifications as they are not reliably sent by the server and can get lost). Details about the commands and notifications can be found in **Part B**.

Part A

Important: This part of the assignment is due until **13.10.2012 18:00 CET** and is **mandatory** for the registration process (i.e., if you do not solve this part in time, you cannot continue with the course!).

Part A represents a simple task to verify your basic Java programming and networking knowledge, which is prerequisite for this course. This task can be solved in several ways. However, we recommend that you implement a rudimentary Java client for this part as you will be able to extend this client for the Part B of this lab.

For this part of the lab, all you have to do is log into our registration server running on host `stockholm.vitalab.tuwien.ac.at` at TCP port 9000 by sending it a command "`!login <username> <password>`" with your dslab account number (e.g., `dslab123`) as username and the initial password that you have received via email after the registration as password. If you have lost your initial password, you can request it from the **DSG Teaching Tool**. You will have to send the command as a simple String. Since *logging in* requires only TCP connections with the server, the UDP communication is not required.

After a successful log in, the server will respond with `"Successfully logged in"` and `"Registration complete"` messages on the TCP connection after few seconds. Then, you are finally fully registered for the course and will be rewarded with 3 points. Please verify this in the **DSG Teaching Tool** - **you are not registered unless you have received the 3 points for Part A in the tool!** In case the server sends the `"Registration complete"` message and you do not receive 3 points, please contact us per **email**!

It should be noted that you do not have to submit your solution for Part A in the **DSG Teaching Tool** as it is necessary in the other labs including Part B of this lab. You just have to implement and invoke your client so that you are able to log into the registration server.

Please also note that any further functionality than *logging in* is disabled on the registration server.

Part B - Server

The auction server application should expect the following arguments:

- `tcpPort`: TCP connection port on which the auction server will receive incoming messages (commands from) clients.

If any argument is invalid or missing print a usage message and exit.

Implementation Details

As it was explained earlier the main task of the auction server is to manage auctions created by the clients. The first thing the auction server does on startup is to open a `java.net.ServerSocket` (initialized with `tcpPort`) in order to be able to receive clients' requests. Since a `java.net.Socket`, which is returned by `ServerSocket.accept()`, provides blocking I/O-operations (via `getInputStream()` and `getOutputStream()`) and we want to serve multiple clients simultaneously each incoming request shall be handled in a separate thread.

Study the java **sockets** and **datagrams** tutorial to get familiar with these constructs. We recommend using *thread pools* (implementations of `java.util.concurrent.ExecutorService`) for implementing the described behavior. They help to minimize the overhead of creating a thread every time a request is received by reusing already existing thread instances. Java provides some sophisticated implementations that can be easily instantiated by using the static factory methods of `java.util.concurrent.Executors`. Anyway you may also manually instantiate new threads on your own without using these classes. In any case, note that threads consume memory (and CPU power), so you should take care that the threads terminate properly. Help can be found in the **Java Concurrency Tutorial**.

After you have performed the steps described above your auction server will be ready to serve clients' requests. Now it is time to add some business logic. The main two responsibilities of auction server are: user and auction management. User management is reduced to a minimum and you don't need to consider user authentication (see `!login` command), as it only requires a single argument, i.e., a username (the authentication part, together with other security measures will be implemented in later assignments). The only "tricky" part here is how to link the notifications with users and not the physical nodes (i.e. terminals).

Considering the notifications, you will need to implement two different types: `!new-bid` and `!auction-ended` (see the detailed description of the notifications below). In order to implement the notifications you will need to use `java.net.DatagramSocket`. DatagramSocket is used to send/receive DatagramPackets. To initialize the destination of a DatagramPacket you will need client's address and an UDP port. The former can be obtained from the established connection with a client by using `socket.getInetAddress()` and the later must be provided by the client (see client description below).

In the auctions management part you will need to address the problems how to create an auction, how to list available auctions, how to place a bid and finally how too terminate an auction after it has expired. The first three parts should not be too complicated, however you should be careful, because there will be multiple clients (threads) sharing the same resources (e.g. a list of running auctions). In these situations you need to pay a special attention to the *synchronization* i.e. you need to make sure your code is *thread-safe*. Study the **Java Concurrency Tutorial** if you are not familiar with threading and/or synchronization.

Considering the problem of expired auction, you will need to prevent further bidding on them, but you will also need to remove them from the list of available auctions (i.e. they don't appear in the response to `!list` command). You can for example use a thread or a `java.util.Timer` in combination with a `java.util.TimerTask` to implement this kind of *garbage collector*. The choice how you want to implement it is completely up to you. However, please note that for the testing purposes the "check interval" should not be longer than 1s.

Important: During the implementation you will need to use different buffered streams (e.g. `java.io.BufferedReader`). In order not to waste the memory careful housekeeping is needed. Therefore, always close buffered streams and sockets after you have finished using them. Additionally, to improve readability and maintainability of the software take a good care of **exception handling** (These things will be checked by our tutors).

The auction server supports only one interactive command. That is in order to shut down the

server a user needs to simply hit the enter key. Do not forget to logout each logged in user. No that as long as there is any *non-daemon thread* alive, the application won't shut down, so you to stop them. Therefore call `ServerSocket.close()`, which will throw a `java.net.SocketException` in the thread blocked in `ServerSocket.accept()`. All other threads currently alive should simply run out. If you are using an `ExecutorService` you have to call its shutdown() method and in case of a `Timer`, call `Timer.cancel()`. Anyway you may not call `System.exit()`, instead **free all acquired resources** orderly.

The two UDP notifications are:

- `!overbid <description>`
  This notification is sent by the auction server to the previously highest bidder, after a new user has placed a higher bid on an auction with description `<description>`.
  E.g.:
  `!overbid Super small notebook`
  on carl's client:
  `carl> You have been overbid on 'Super small notebook'`

- `!auction-end <winner> <amount> <description>`
  This notification is sent by the auction server to an auction owner and the auction winner after the auction has ended.
  E.g.:
  `!auction-end dave 250.00 Super small notebook`
  on alice's client:
  `alice> The auction 'Super small notebook' has ended. dave won with 250.00.`
  on dave's client:
  `dave> The auction 'Super small notebook' has ended. You won with 250.00!`

Part B - Client

The client application should expect the following arguments:
- `host`: host name or IP of the auction server
- `tcpPort`: TCP connection port on which the auction server is listening for incoming connections
- `udpPort`: this port will be used for instantiating a `java.net.DatagramSocket` (handling UDP notifications from the auction server).

If any argument is invalid or missing print a usage message and exit.

Implementation Details

The main task of your client is to read user requests from standard input (`System.in`) and to communicate them to the auction server accordingly. The other important task of the client is to receive and display different notifications, sent by the auction server, to users.

One of the first things to do here is to create a `java.net.Socket` and connect to the auction server. You will need the `host` and `tcpPort` values for this. Outgoing messages are sent each time the user enters one of the interactive commands. The responses from the auction server should be handled in an own thread. Keep the connection open as long as either the client or the auction server shut down.

Another thing you will need to do in order to be able to receive the notifications from the auction server is to create a `java.net.DatagramSocket`. As the auction server needs to know the full address where to send the notifications. Therefore, your client will need to tell it the `udpPort` of the `java.net.DatagramSocket`. The best way to do this is simply to send `udpPort` as a part of `!login` message. Additionally, please note that the `DatagramSocket.receive(packet)` is a blocking operation. Therefore, using a separate thread to receive notifications should be considered.

The list of interactive commands supported by the client are listed below:

- `!login <username>`
  Logs in the user <username> to the auction system.
  E.g.:
  ```
  > !login alice
  Successfully logged in as alice!
  ```

- `!logout`
  This command logs out the currently logged in user.
  E.g.:
  ```
  alice>: !logout
  Successfully logged out as alice!


  > !logout
  You have to log in first!
  ```

- `> !list`
  Lists all the currently active auctions in the system. The output formatting is not important, but the auction id, the auction's description, owner's username, the current highest bid amount and the current highest bidder's username must be displayed. If an auction does not currently have any bids, the listing of that auction should output '`0.00`' and '`none`' for the highest bid and the highest bidder's username respectively.
  E.g.:
  ```
  > !list
  1. 'Apple I' wozniak 10.10.2012 21:00 CET 10000.00 gates
  2. 'My left shoe' 07.10.2012 12:00 CET 0.00 none
  3. 'Super small notebook' 04.10.2012 18:00 CET 250.00 dave
  ```

- `> !create <duration> <description>`
  With this command a user can create new auction. A short description of the auction and its duration (given in seconds) must be specified. Auction id is automatically assigned by the auction system.
  E.g.:
  ```
  > !create 25200 Super small notebook
  An auction 'Super small notebook' with id 3 has been created and will end on 04.10.2012 18:00 CET.
  ```

- `> !bid <auction-id> <amount>`
  With this command users can bid <amount> on a specific auction <auction-id>.
  E.g.:
  ```
  > !bid 3 250.00
  You successfully bid with 250.00 on 'Super small notebook'.


  > !bid 3 220.00
  You unsuccessfully bid with 220.00 on 'Super small notebook'. Current highest bid is 250.00.
  ```

- `!end`
  Shuts down the client. Note, same as for the server, as long as there is any *non-daemon thread* alive, the application won't shut down, so you need to stop them. You may not call `System.exit()`, instead again **free all acquired resources** orderly.

Lab port policy

Since it is not possible to open `ServerSocket`s or `DatagramSocket`s on ports where other services are already listening for requests, we have to make sure each student uses its own port range.
That means that if you are testing your solution in the lab environment (i.e., on the lab server) you have to obey to the following rule: you may only use ports between **10.000 + dslabXXX * 10** and **10.000 + (dslabXXX + 1) * 10 - 1**. So if your account is dslab250 you may use the ports between 12500 and 12509 inclusive. Note that you can use the same port number for TCP and UDP services (e.g., it is possible to use TCP port 12500 and UDP port 12500 at the same time).

Ant template

Ant is a Java-based build tool that significantly eases the development process. If you have not installed `ant` yet, download it and follow the **instructions**.
We provide a template build file (**build.xml**) in which you only have to adjust some properties. Put your source into the subdirectory "`src`". To compile your code, simply type "`ant`" in the directory where the build file is located. Enter "`ant run-server`" to start a server and "`ant run-clientX`" (with X being 1 or 6) to start a respective client.
Note that it's **absolutely required** that we are able to start your programs with these predefined commands!

Also note that build files created by IDE's like Netbeans very often aren't portable, so please u
the provided template.

Hints & Tricky parts

Binding problems

After starting your socket implementation, you receive a message comparable to Error 1:

```
java.net.BindException: Address already in use: JVM_Bind
        at java.net.PlainSocketImpl.socketBind(Native Method)
        at java.net.PlainSocketImpl.bind(Unknown Source)
        at java.net.ServerSocket.bind(Unknown Source)
        at java.net.ServerSocket.<init>(Unknown Source)
        at java.net.ServerSocket.<init>(Unknown Source)
```

Error 1

The address and port, you are trying to bind to is already in use. To solve these issues you can test your solution at home (and don't bind to ports used by other services ;), or you simply change the ports. Our policy to use your dslab account number should prevent these errors! For example: `dslab023 = Bind to port 10230`. If you are strictly following the port convention of the lab and still seeing this problem, it either means that somebody else is not (and blocking your ports) or that there is still a zombie of a previous test run of your application online. Use `ps` to find the zombie and `kill -9` it.

Further Reading Suggestions

- **APIs:**
  - IO: **IO Package API**
  - Concurrency: **Thread API**, **Runnable API**, **ExecutorService API**, **Executors API**
  - Java TCP Sockets: **ServerSocket API**, **Socket API**
  - Java Datagrams: **DatagramSocket API**, **DatagramPacket API**

- **Tutorials:**
  - JavaInsel Sockets Tutorial - Section **21.6**, **21.7**: German tutorial for using TCP sockets.
  - Java Programmierhandbuch und Referenz - Section **13.2.3**: German tutorial for using datagram sockets.
