From LU Verteilte Systeme (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 the lab. You have to work alone. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone.
- 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 **14.10.2010, 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 Part A, but you should reuse it for solving Lab 1.
- 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: **28.10.2010, 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!
- Do not confuse our lab server (`pasta.dslab.tuwien.ac.at`) with the **Teaching Tool**. The lab server is 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 the 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 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 socket communication
- how to program multithreaded
- different connection types

Overview

In this year's first assignment we are going to build a simple replicated Client-server system that can be used to download text files. The architecture is as follows: Fileservers store all the files available to the clients. They are fully replicated, i.e., each single fileserver stores exactly the same files. Files are identified by their filename and do all reside in the same, given directory in the file system.

At any point of communication, the clients only know the address of one particular server. This server stores no files at all but forwards any incoming download request to one of the available fileservers. Due to this task, we will call this server 'Proxy' in the following.

The Proxy stores information about every client and every fileserver in the communication process. Clients are limited in the amount of data they are allowed to download. Because it is forwarding each client request and each fileserver's response, the Proxy can easily keep track of the user's current limit and block download requests where necessary. The same approach is used to balance the upcoming traffic in the private fileserver network: The fileserver to choose for responding to the next client request is always the one with the lowest usage at that time. Figure 1 illustrates a simple example:

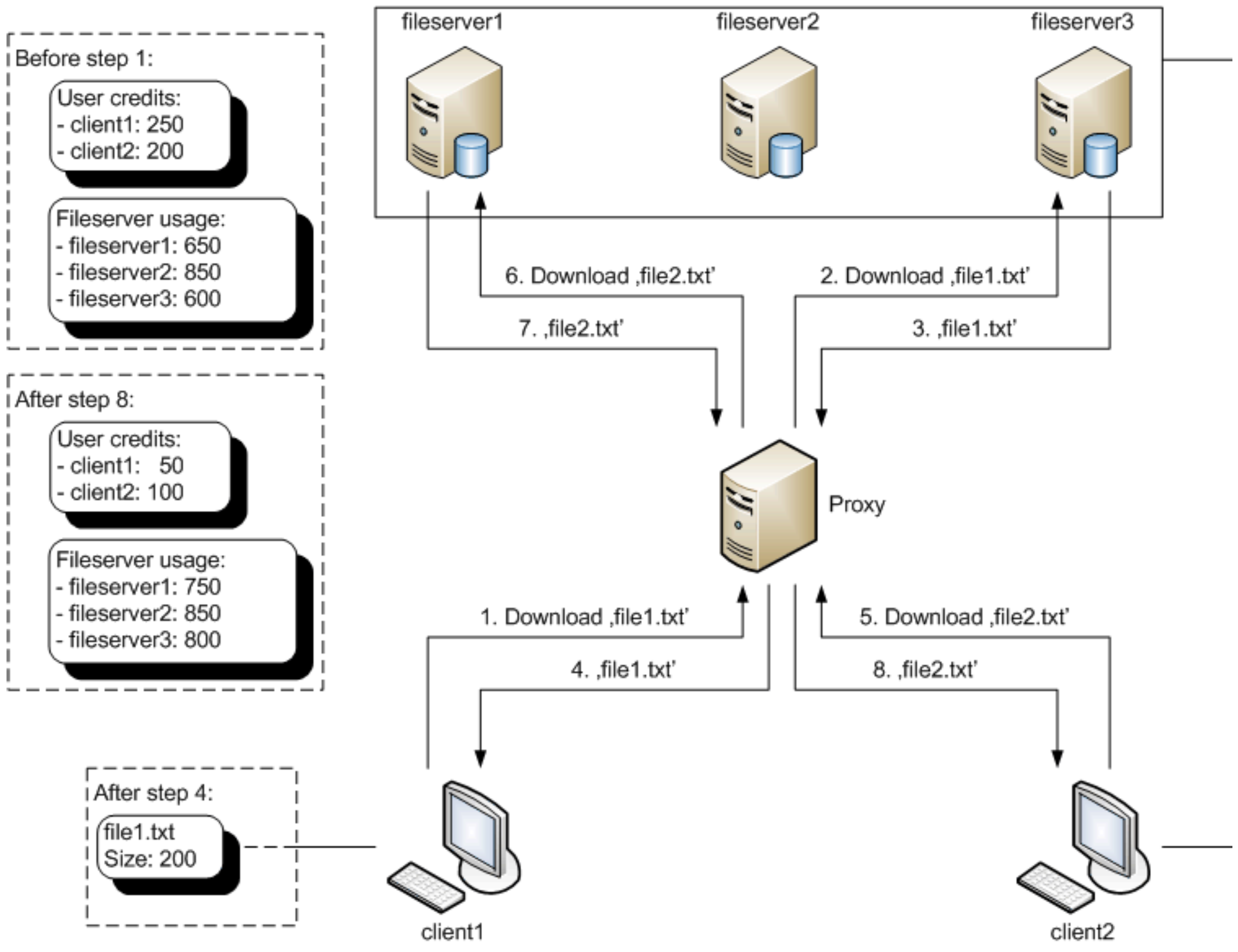

Figure 1

To avoid a waste of network resources, there is no connection being held between the Proxy and a fileserver between two distinct requests. That is, after the fileserver has responded to the Proxy's request, the connection gets closed again. However, to signal that it is still online and ready to handle requests, a fileserver needs to send UDP messages (so called "isAlive" packets) in a recurring manner – any other communication in this assignment is done using TCP. Figure 2 illustrates this behavior: Imagine that in the example above, 'fileserver1' would fail to send alive packets to the Proxy. The Proxy will remove 'fileserver1' from its list of available fileservers and instead forward the second download request to 'fileserver3', which now is the least used fileserver.

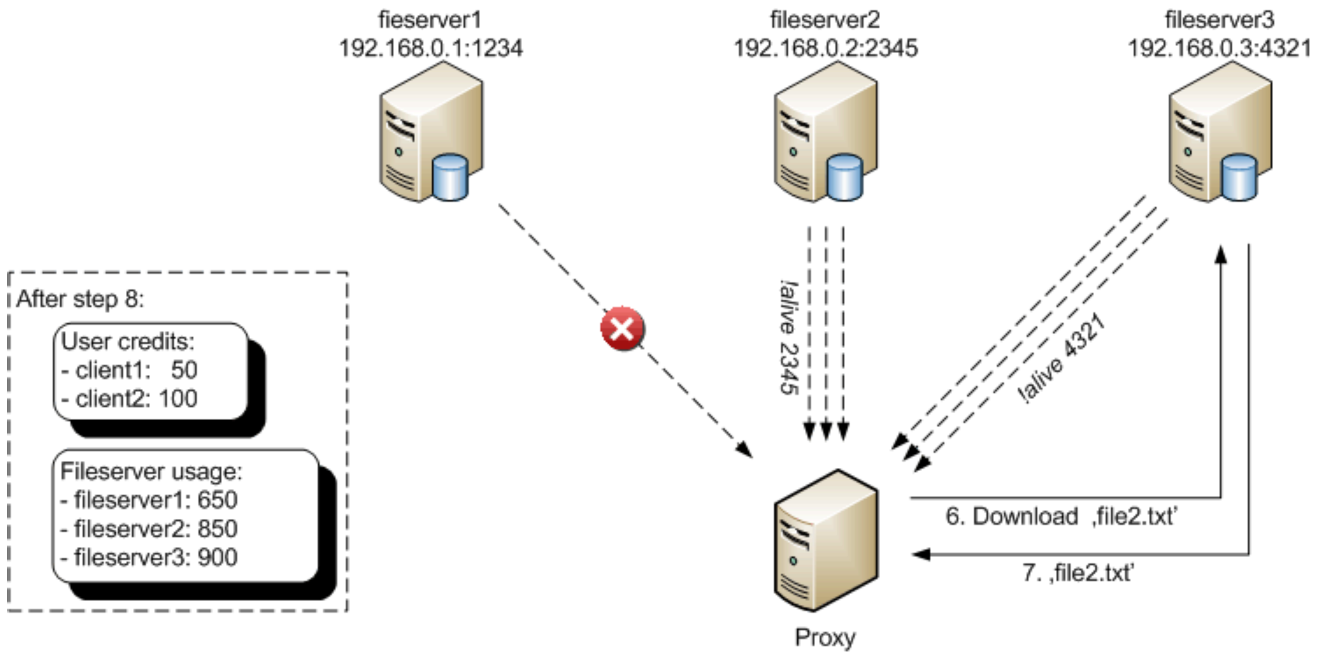

Figure 2

Please note that these figures are for illustration purposes and some details have been omitted for the sake of simplicity. You find these details (parameters, return values etc.) below.

Part A

Important: This part of the assignment is due to **14.10.2010 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. Basically, all you have to do is log into our registration server running on host `stockholm.vitalab.tuwien.ac.at` at TCP port 9000 (see **explanation** of the *login* command). Therefore, use your dslab account number (e.g., `dslab123`) and initial password that you have received via email after the registration. If you have lost your initial password, you can request it from the **DSG Teaching Tool**.

This task can be solved in several ways. However, we recommend that you implement a rudimentary **client** because you need to implement this client for the rest of Lab 1 anyway. Since *logging in* requires only TCP connections with the server, the UDP communication is not required for Part A. **Please also note that any further functionality than *logging in* is disabled on the registration server.**

Send your request as a simple String. After successful log in, the server will respond with `"Successfully logged in"` and `"Registration complete"` messages on the TCP connection after some seconds. Then, you are now finally registered for the course and will be rewarded 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 necessary in the other labs). You just have to implement and invoke your client so that you can login to the registration server.

Part B - Proxy

Arguments

The Proxy application should expect the following arguments:

- `tcpPort`: the port to be used for instantiating a `java.net.ServerSocket` (handling TCP connection requests from clients).
- `udpPort`: the port to be used for instantiating a `java.net.DatagramSocket` (handling UDP requests from fileservers).
- `fileserverTimeout`: the period in milliseconds each fileserver has to send an *isAlive* packet (only containing the fileserver's TCP port). If no such packet is received within this time, the fileserver is assumed to be offline and is no longer available for handling requests.
- `checkPeriod`: specifies that the test whether a fileserver has timed-out or not (see `fileserverTimeout`) is repeated every `checkPeriod` milliseconds.

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

Implementation Details

The first thing the Proxy needs to do on startup is to read the `users.properties` file which must be located in the server's classpath. The properties file is provided and can be downloaded **here**. Each line of a .properties-file stores a single property consisting of *key* and *value*. We will use a .properties-file here to store information about each user, more precise, the user's password required for logging in and the particular credits limiting the user in the amount of data he/she is allowed to download. For instance, an entry in such a .properties-file for user *alice* with password *12345* and *500* credits looks this:

```
alice = 12345
alice.credits = 500
```

How to read in properties-files from classpath can be found in **Hints & Tricky parts**. When communicating with clients and fileservers, the credits of users may change. However, do **not** store these changes back to the .properties file: The credits of each user shall be reset to the initial value after a restart of the Proxy.

Next, create a `java.net.ServerSocket` as well as a `java.net.DatagramSocket` instance. We want to concurrently listen for new connections from clients on the `ServerSocket` and wait for incoming *isAlive* packets on the `DatagramSocket`. Since both relevant methods (`ServerSocket.accept()` und `DatagramSocket.receive()`) are blocking operations, you shall spawn a new *thread* for each in which you call these methods in a loop. This way, the Proxy is still able to listen for command line inputs.

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, again each incoming request shall be handled in an own thread. The same holds for UDP packets: To maximize the concurrency and performance of our server, each incoming `java.net.DatagramPacket` (filled by `DatagramSocket.receive()`) shall also be processed 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. Help can be found in the **Java Concurrency Tutorial**.

After loading the user information and initializing all sockets and threads, your Proxy is able to serve requests.

Concerning the `DatagramSocket`, *isAlive* messages are the only valid packets that may arrive here. Fileservers that did not send such a packet for the specified time (`fileserverTimeout`) must be concerned offline. You can either use a thread or a `java.util.Timer` in combination with a `java.util.TimerTask` to implement this kind of garbage collector. Do a recurring check every `checkPeriod` ms. In case a fileserver goes offline, its usage statistics (defining how many data the fileserver has uploaded to clients) shall not get lost. It is enough to keep this information in-memory, so the usage statistics can finally get lost after stopping the Proxy.

Concerning TCP communication, different client messages may arrive (described in detail in the **Client part**). Some of them require the Proxy to forward them to one of the available fileservers (sometimes the Proxy may need to adapt the message and add additional information first). In this case, the Proxy should always choose the least used fileserver that is supposed to be online, i.e., the fileserver that has the lowest usage statistics. Keep in mind that even though you listen for *isAlive* packets, this does not guarantee a fileserver reported to be online did not go down in the meantime.

Because the Proxy is the communication interface for both clients and fileservers, forwarding each message exchanged between them, the Proxy can easily keep track of the users' credits and the fileservers' usage statistics: Each single download decreases the credits of a single user and increases the usage of a single fileserver. Note that the Proxy needs to assure that the requested file actually exists and the user has enough credits left to download the file. Needless to say, this requires communication with a fileserver.

Since you've got to manage users and fileservers across threads you have to deal with *synchronization* – make sure your code is *thread-safe*. Study the **Java Concurrency Tutorial** if you are not familiar with threading and/or synchronization.

Finally, the Proxy accepts three interactive commands:

- `!fileservers`
  Prints out some information about each known fileserver, online or offline. A fileserver is known if it has sent a single *isAlive* packet since the Proxy's last startup. The information shall contain the fileserver's IP, TCP port, online status (online/offline) and usage.
  E.g.:
  ```
  >: !fileservers
  1. IP:127.0.0.1 Port:10000 offline Usage: 752
  2. IP:127.0.0.2 Port:10000 online Usage: 220
  ```

- `!users`
  Prints out some information about each user, containing username, login status (online/offline) and credits.
  E.g.:
  ```
  >: !users
  1. alice online Credits: 500
  2. bill offline Credits: 180
  ```

- `!exit`
  Shutdown the Proxy. Do not forget to logout each logged in user. Note that as long as there is any *non-daemon thread* alive, the application won't shut down, so you need to stop them. Therefore call `ServerSocket.close()`, which will throw a `java.net.SocketException` in the thread blocked in `ServerSocket.accept()`, and `DatagramSocket.close()`, which will throw a `SocketException` in the thread blocked in `DatagramSocket.receive()`. 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.

Further implementation details can be found in the following parts.

Part B - Fileserver

Arguments

The fileserver application should expect the following arguments:

- `sharedFilesDir`: the directory that contains all the files clients can download.
- `tcpPort`: the port to be used for instantiating a `ServerSocket` (handling the TCP requests from the Proxy).
- `proxyHost`: the host name (or an IP address) where the Proxy is running.
- `proxyUDPPort`: the UDP port where the Proxy is listening for fileserver datagrams.
- `alivePeriod`: the period in ms the fileserver needs to send an *isAlive* datagram to the Proxy.

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

Implementation Details

A fileserver provides files located in a particular directory (`sharedFilesDir`). For the sake of simplicity, your solution needs to deal with ASCII files only. You also do not have to deal with sub-directories and any files that may exist there.

To handle a download request, the fileserver first has to check if the respective file exists. The class `java.io.File` provides all the functionality required for this check. If the file cannot be located, inform the Proxy about it. The Proxy will also need the information whether the user's current credits are enough to download the file.

So, how can you measure whether the user has enough credits for this particular file? In this Lab, we will use the actual size in bytes of the file to download for a comparison. You can simply use the method `File.length()` to get the filesize. All you have to do now is comparing this value with the user's current credits.

If everything is ok, (i.e., the file exists and the user has enough credits), read out the file as a simple String and make it part of the fileserver’s response. To this, the classes `FileReader` and `BufferedReader` inside the `java.io` package may be helpful. You should also make sure the Proxy gets all the information required to update the user's credits and fileserver's usage from this response.

To be able to receive requests from the Proxy, you have to create a `ServerSocket` again. Blocking I/O-operations should be handled in own threads using exactly the same approach described for the **Proxy part**. After a fileserver has completely processed a request and sent the response back to the Proxy, the respective socket should be closed. For any new request, a new Socket needs to be created.

From time to time, each fileserver needs to send *isAlive* packets to the server to demonstrate it is still online and is ready to handle client requests. The very first packet that arrives works as a registration in the fileserver network (i.e., the Proxy then is aware of the new fileserver). Open a `DatagramSocket` on an arbitrary port and send an *isAlive*-packet every `alivePeriod` ms. Use the `DatagramSocket.send()` method for this. Making the fileserver’s TCP port part of the alive message is very important so that the Proxy knows where to forward client requests. Again, you can either use a thread or a Timer to continually send these datagrams.

If you are using any in memory data structures, make sure your code is thread-safe.

The only interactive command the fileserver accepts is `!exit` which shuts down the fileserver. To this, the same rules as for the **Proxy** apply.

Part B - Client

Arguments

The client application should expect the following arguments:
- `downloadDir`: the directory to put downloaded files.
- `proxyHost`: the host name (or an IP address) where the Proxy is running.
- `proxyTCPPort`: the TCP port where the server is listening for client connections.

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

Implementation Details

The client communicates with the Proxy to download files from different fileservers and store them in a private directory on the local host.

One of the first things to do here is to create a Socket and connect to the Proxy. You will need the `proxyHost` and `ProxyTCPPort` values for this. Listen for incoming messages in an own thread. Outgoing messages are sent each time the user enters one of the interactive commands described below. Keep the connection open as long as either the client or the Proxy shuts down.

When it comes to downloading a file, make sure a file with this name does not already exist in `downloadDir`. In case it does, delete this file before (again, the `java.io.File API` will provide all the required functionality). If the client requests a file that does not exist on the fileserver or the user does not have enough credits to download the file, print the respective error message.

The main task of your client is to read user requests from standard input (`System.in`) and send them unchanged to the Proxy. Therefore, your Proxy should be able to deal with unknown commands or missing arguments (reply with a simple usage message in this case). The Proxy should also make sure that clients cannot execute any commands different from `!login` before they have finally logged in successfully.

Interactive commands

- `!login <username> <password>`

  Log in the user. Before the user hasn't successfully logged in, this is the only command that will be executed by the Proxy.
  E.g.:
  ```
  >: !login alice 23456
  Wrong username or password.
  >: !login alice 12345
  Successfully logged in.
  ```

- `!credits`

  Requests the user's current amount of credits. Requires a successfully logged in user.
  E.g.:
  ```
  >: !credits
  You have 500 credits left.
  ```

- `!buy <credits>`

  Allows the user to increase his/her amount of credits. Requires a successfully logged in user.
  E.g.:
  ```
  >: !credits
  You have 500 credits left.
  >: !buy 1000
  You now have 1500 credits.
  ```

- `!list`

  Gets the complete list of files available to download. Requires a successfully logged in user.
  E.g.:
  ```
  >: !list
  file1.txt
  file2.txt
  ```

- `!download <filename>`

  Downloads the specified file into the private download folder (`downloadDir`).
  E.g.:
  ```
  >: !download file1.txt
  File successfully downloaded.
  ```

- `!exit`

  Shutdown the client: Logout the user if necessary and be sure to release all resources, stop all threads and close any open sockets.

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 parameters and class names. 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-proxy`" to start the Proxy, "`ant run-client`" to start the client and "`ant run-fileserverX`" (with X being 1 or 2) to start the respective fileserver.
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 use 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.

Reading user.properties file

This code is just an example and it doesn't do any exception handling:

```
java.io.InputStream in = ClassLoader.getSystemResourceAsStream("user.properties");
if (in != null) {
     java.util.Properties users = new java.util.Properties();
     users.load(in);
     java.util.Set<String> usernames = users.stringPropertyNames(); // get all usernames
     for (String username : usernames) {
          String password = users.getProperty(username); // get password for user with
          username
          ...
     }
} else {
     // users.properties could not be found
}
```

The properties file for this Lab can be downloaded **here**

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 **18.7**, **18.8**: German tutorial for using TCP sockets.
  - JavaInsel Datagrams Tutorial - Section **18.11**: German tutorial for using datagram sockets.

Retrieved from https://www.infosys.tuwien.ac.at/teaching/courses/dslab/index.php?n=Lab1.Lab1
Page last modified on October 05, 2010, at 05:26 PM CET