OOP, exercise 3 – A file sharing network

Goals: to exercise, experience
– Working with sockets and client-server communication
– Working with multiple threads and synchronization
– Understanding network implementation issues

Submission Deadline: 17.05.2011
Note: In this exercise you may use all classes available in **standard** java 1.6 distribution (as introduced in the intro course and in this course so far), including all classes below *java.io*, *java.util*, *java.net* and *java.text*.

Introduction

In the last decade, distributed peer-to-peer file sharing programs became very popular. These programs support the sharing and exchange over the internet of documents, audio and video content, and many other types of data. In this exercise you will implement a simple file sharing network. The file sharing network comprises one or more instances of two basic components: **NameServer** and **FileManager**.

A **FileManager** maintains a set of files. It allows its users to remove\rename existing files and to obtain new files from other **FileManagers**. On startup, a **FileManager F** receives as input its list of files and a list of **NameServers**. Then, **F** sends its list of filenames and **NameServer** names to all the **NameServer**s on its **NameServer**s**'** list (so that each **NameServer** in **F**'s list will know everything about **F**). Following, **F** receives commands from the standard input which asks it either to delete\rename some of his files or download new files (by requesting them from other **FileManager**s). A **FileManager** also listens and performs upload requests from other **FileManager**s.

A **NameServer** maintains information about which files are stored in each of the known **FileManager**s. The **NameServer** also helps to distribute between **FileManagers** information about other **NameServer**s. On startup, the **NameServer** does not possess <u>any information</u> about other **NameServers**, **FileManager**s or their files. It just waits for requests from **FileManager**s. When some **FileManager F** connects to a **NameServer N**, **N** retrieves from **F** the list of **F**'s files and the list of **NameServers** known to **F. N** remembers these lists in some data structure (of your choice) and uses\updates them following subsequent requests from other **FileManager**s.

The main basic communication scenario in this framework is the following:
1) A user of FileManager **F** requests a file which is not in the collection of files of **F.**
2) **F** queries all the NameServers known to it for addresses of other FileManagers that have the desired file.
3) Some NameServer returns the address of FileManager **G**.
4) FileManager **F** connects to FileManager **G** and downloads this file.
5) FileManager **F** updates all NameServers known to it with its new possession.

The functionality of NameServers and FileManagers

- In the following description, the terms “download” and “upload” mean:
  - “download” – get a file from another FileManager
  - “upload” – send a file to another FileManager.
- In addition, capitalized names below are names of messages which should be sent. The exact message format for each of these messages is described in the Protocol section.
- We refer to the collection of files in this directory as ***F****’s database.*

NameServers

A NameServer has no direct interaction with users and never initiates communication. After it is started, it reacts to requests from FileManagers. If it receives **GOAWAY** message from some FileManager, it shuts down[1]. A NameServer is initialized with a port number (int) supplied as command line parameters. Once initialized, it “listens” to this port, waiting for FileManagers to connect.

Example for running NameServer: java oop.ex3.nameserver.MyNameServer 2222

Communication between NameServer and FileManager

Each communication session between a NameServer **N** and a FileManager **F** begins as following (uppercase letters indicate message names):

1.1) **F** identifies itself by sending its ip and port to **N (BEGIN)**

1.2) If **N** doesn’t recognize **F** by ip+port (i.e. it doesn’t know him), it replies with **WELCOME**, otherwise it responds with **DONE** and waits for the next message from **F**.

1.3) If **F** receives a **WELCOME** message it shares his data with **N** by sending it a list of names of all its files and list of names of known NameServers (many **CONTAINFILE** followed by a single **ENDLIST**, and many **CONTAINNAMESERVER** followed by a single **ENDLIST**). **N** responds to each name with **DONE** and stores this information for future use. If **F** doesn’t have any file an empty list should be sent (i.e. only the message **ENDLIST**).

Note: **N** should store the information about the files and other servers only if the entire session ended successfully (meaning that all of the **CONTAINFILE** messages and the **CONTAINNAMESERVER** messages were answered with **DONE**. However, even if all **CONTAINFILE** messages were answered with **DONE** but one of the **CONTAINNAMESERVER** had failed (see later), **N** should not keep any of files sent during this session).

To sum up stages 1.1-1.3: when **F** connects to **N**, **N** wants to know all data about **F**. If **N** already knows **F**, no additional data is sent. If **N** doesn't know this **F**, it asks **F** to provide full lists of files and NameServers.

If **F** doesn’t perform step 1.1 at the beginning of a session or if some of the steps 1.1-1.3 failed, **N** should respond with **ERROR** message and disconnect.

Note: On any **ERROR** message sent from a NameServer N to FileManager F: N should close the connection, without shutting down. F should disconnect and end the current session (though next time this FM should still attempt to connect to this NameServer).

[1] The **GOAWAY** message is for debug purpose only and real systems should not have such a mechanism.

After such initialization **F** may send **N** one or more of the following messages (see next section for details about each scenario):

2.1) **F** notifies **N** about the deletion of one of its files (**DONTCONTAINFILE**). **N** updates its information (if any update is needed) and responds with **DONE** (even if **N** doesn't remember that **F** had this file earlier).

2.2) **F** notifies **N** about the addition of a file to **F**'s database (**CONTAINFILE**). **N** updates its information and responds with **DONE**.

2.3) **F** requests **N** the location of some file (**WANTFILE**). If **N** knows any FileManagers with this file, it sends back to **F** a list of their addresses (many **FILEADDRESS** messages followed by a single **ENDLIST**), otherwise it responds with a **FILENOTFOUND** message.

2.4) **F** requests **N** the additional names of NameServers (**WANTSERVERS**). **N** responds to **F** with the full list of NameServers known to **N** (**CONTAINNAMESERVER**) (this list includes all NameServers already known to **F**, including **N** itself).

2.5)F requests **N** all the file names known to it (**WANTALLFILES**). If no files are known to **N** (for example they were all deleted), **N** responds with a **ENDLIST** message. Otherwise, **N** sends back to **F** a list of its file names (many **NSCONTAINFILE**, **ENDLIST**).

2.6) **F** notifies **N** that it is going down (**GOODBYE**). **N** clears all file list information about **F** (still keeping in memory NameServers list supplied by **F**) and replies with **DONE**.

2.7) **F** sends **N** the **GOAWAY** message. **N** responds **DONE** and shuts down. **N** shouldn't notify anyone else that it's going down.

Comments

1. When a communication session ends , **N** remembers **F** (by ip+port) and remembers all relevant data from **F** **(this holds for all session ending apart from the cases of (2.6), (2.7). In these cases N doesn't remember F anymore. I.e. in the next first time they communicate, N will reply with WELCOME to a BEGIN message coming from F)**. At the end of communication session, **F** sends **N** the **ENDSESSION** message and **N** replies with **DONE**. In other words, on **N**'s side the session finishes after receiving **ENDSESSION** and sending **DONE**. On **F**'s side, the session finishes after sending **ENDSESSION** and receiving **DONE**.
2. If any problem (IO error \ invalid message \ timeout) occurs during the communication session with **F**, **N** should reply to **F** with **ERROR** message and the close connection (note that **F** never sends **N** an **ERROR** message). If **N** responds with **ERROR** to 2.1, 2.2, 2.6 or 2.7, no changes should be made to **N** and **N**'s database. If **F** receives a bad message from **N** (i.e. a message which it doesn't understand, a message which is not included in the protocol or a message that doesn't belong there), the problematic connection should be closed (using **ENDSESSION**), however the server or manager should continue to work properly on next operations and requests. Both FMs and NSs deal with timeoutExceptions (see below) in the same way: A timeout during session means that the current session had failed so you should continue to the next FM\NS according to the protocol.
3. Communication sessions are handled by **N** as follows: **N** listens to requests from FileManagers. When a request arrives from **F**, **N** opens a connection, i.e. creates a socket, and **creates a new thread** to handle the session. After the end (successful or not) of this session, the corresponding communication thread closes the socket and finishes its execution. If **N** receives a **GOAWAY**

message from some FileManager **F**, **N** should send a **DONE** message. Then, **F** should send **ENDSESSION** message, and **N** should send a **DONE** message. At this point **N** should terminate nicely (see next comment) after ongoing sessions are finished. If **N** receives a **GOAWAY** message from **F** there should not be any other messages (apart from **ENDSESSION** message) from **F**, so if **N** receives other messages (apart from **ENDSESSION** message) it should send **ERROR**, and the operation should fail (**N** won't die). In addition N should not accept new requests of communication from other FileManagers.

4. In order to implement a shutdown of a NameServer shouldn't use system.exit(). Rather you should use method learned in class to close all running threads and safely return from the main method.

FileManagers

A FileManager **F** is created from the command line, with three input parameters in the following order:

i) **A name of server list file.** The structure of the file: each line represents an address and a port number of a NameServer, separated by a single % sign.
ii) **A name of directory (**in which its files are stored). This is F's database.
iii) **A port number**.

Example for running F: java oop.ex3.fileManager.MyFileManager ./serverlist1.txt ./mydir1/ 3001

Here's an example of a serverlist file (you can assume that server names do not contain "%" characters and that there is only a single % sign per line):

```
127.0.0.1%2222
localhost%2228
```

Note: The IP address ***127.0.0.1*** is a special purpose address reserved for use on each computer and it represents the host, namely the computer itself. ***localhost*** is another name for the same purpose. In java, you can obtain the actual IP address of your computer by using command:

```
InetAddress.getLocalHost().getHostAddress()
```

You can assume that the format of the data in the server list file is valid, and that it contains at least one entry. We refer to the servers in this list as the NameServers *known* to **F**. In addition you may assume that given directory exists and that it contains only valid and readable files (not links). Moreover, assume that it does not contain any subdirectories (you don't have to check these points).

As mentioned above, the collection of files that F holds is referred as ***F****'s database*. In requests from users, and in sessions between FileManagers and NameServers, files are always denoted by simple names, such as "myfile", "song13.wav", and **not** by paths. You can assume that all file names do not contain any spaces. Other than that, there are no restrictions on file names.

During initialization, **F** first loads the information in the server list file, and the file names list from the directory, into some data structures of your choice. **The order of the NameServers in file is important** - **F** should request NameServers for information in the same order as they appear in the server list file.

On startup, **F** connects to each NameServer known to it, announces its existence and sends it a list of its file names and a list of NameServers (1.1-1.3). Following, **F** creates two threads. The first thread (the main thread) is responsible for handling interaction with the user & performing downloads. The second thread listens to requests from other FileManagers and opens a new thread on each incoming upload request.

FileManager User\download thread

In contrast to NameServers, FileManagers *do interact with the users*. This is done via the main thread, also referred to as the "user thread". For simplicity, we assume that all interactions are via text commands that a FileManager reads from the standard input. For each legal command which requires a connection to NameServers, it initiates new communication sessions with one or more NameServers and\or other FileManagers, performs the request and closes the sessions. The user commands, and the expected responses, are as follows (all commands are **case-sensitive and you can assume all commands are valid**):

Note: All "Print" operations mentioned below, should be directed to the STDOUT (i.e., use System.out.println()).

1) DIR – print the names of all the files in **F**'s database, one file name per line. The order should be lexicographical (for example: 1.txt, afile.m, afile.pl, bfile.mp3, etc).
2) DIRSERVERS – print ip+port number of the NameServers known to **F**, one per line (print them as they appear in the input file or as they are received from other NameServers). NSs that are encountered during the run should come after the input NS, and in the order in which they were encountered**.**
3) ADD filename **–** If the file is found in **F**'s database, it should output the message "File already exists" to the STDOUT. Otherwise it should try and download if from another FileManager (and create a physical copy of the file in the destination folder). The entire process should be conducted as the following (you should follow the specified order):
    a) Go through the NameServers known to it, **sequentially.** For each server, request location of this file (2.3).
    b) If the server responds with a list of FileManagers, then go through the list, **sequentially**, and try to obtain the file. Specifically:
        a. Connect to a FileManager (let's call it FM2) in the list and try to download the file (i.e. copy the file **using the socket stream**. I.e., send to FM2 a **WANTFILE** message and receive from FM2 a **FILE**, see details below message as a respond). This (download) operation should be done in the same thread so no other user commands are possible during this operation.
        b. If this fails (**ERROR** or **FILENOTFOUND**), try the next FileManager on the list.

        Note: In order to send the file to the FileManager who requested it, FM2 spawns a new upload thread. This thread either sends the file (using the **FILE** message) or sends the **FILENOTFOUND** message if this file is no longer there.
    c) If a server responds with a **FILENOTFOUND** message, or the file could not be downloaded from all the FileManagers in the list it has sent, the next NameServer is tried.
    d) If the file has been downloaded successfully, the **F** should not connect to additional FMs\NSs. The file list of **F** will be updated, and a message "File Downloaded Successfully from filemanageraddress:filemanagerport" will be printed (for example, "File Downloaded Successfully from 127.0.0.1:3001"). Also, all known NameServers (including NameServers that were added during the run) will be updated with this information (2.2).
    e) If a file couldn't be downloaded from any of the known NameServers:

a. Request all NameServers for more NameServer names (2.4) and add the obtained names to a list of known NameServers (***don't add them to the given input file and don't notify other known NameServers that you know about these NameServers***). Note that the added NameServers should be used in subsequent **ADD** commands and printed in **DIRSERVER** command and killed with **GOAWAY** command. New NameServers should be added to the end of NameServer list in memory.

b. Repeat the a)-e) with (only) newly obtained NameServers until no new NameServers are found or the file is downloaded. (Note: this should be done recursively, i.e., ask for NameServer names from the new servers in case of failure. The implementation can be either recursive or loop-based).

During this process, F may approach the same FM more than once (for example if F tried to download a file from a FileManager named FM2 and failed. FM2 may be sent by more than one NamseServer and you should consider it again).

f) If the file could not be downloaded (either because it is not stored anywhere, or because all download attempts have failed). Print "Downloading failed".

4) REMOVE filename – If the file is not in its database (i.e. deletion is not possible), **F** should output the message "It is impossible to delete an absent file". Otherwise it should do the following:

a) If the file is being uploaded to some other FileManagers, wait for the all uploads to finish; do not start new uploads for this file. After ensuring that all uploads are done, continue to the next step. On any new upload request during this period, respond with **FILENOTFOUND** message.

b) If the file is currently in the database, delete it from the database and the directory. (Physical deletion form the directory)

c) Update all servers (2.1) (including new servers that were obtained during the run).

d) Print "Removing Done"

5) DIRALLFILES - Print out the list of files in the system:
a) Go through the NameServers known to it, **sequentially.**
b) For each NameServer get the file names known to it (2.5).
c) Request all NameServers for more NameServer names (2.4) and add the obtained names to a list of known NameServers (don't notify other known NameServers that you know about these NameServers). Note that the added NameServers should be used in subsequent ADD commands and printed in DIRSERVER command and killed with GOAWAY command. New NameServers should be added to the end of NameServer list in memory.
d) Again, for each NameServer get the file names known to it. Repeat a)-d) with newly obtained NameServers until no new servers are found. In a similar manner, as described in the ADD command, this should be done recursively in order to get all reachable NameServers (the implementation can be either recursive or loop-based).
e) Print all the retrieved file names alphabetically sorted. Only unique names should be printed; i.e. even if the same file name appears in more than one FM, it should be printed only once.

6) RENAME *oldname newname* – rename the file named *oldname* to *newname*

a) If the file named *oldname* is not in its database (i.e. renaming is not possible), F should output the message “It is impossible to rename an absent file”.

b) If the file named *oldname* is in the database but a file named *newname* is also in the database, **F** should output the message “It is illegal to use an existing file name as a new name”.

c) In case the *oldname* file is not in the database and the *newname* file is in the database (i.e. both a) and b) hold), the program should behave as if it’s a) (and print “It is impossible to rename an absent file”).

d) Otherwise it should do the following:

   i. If the file is being uploaded to some other FileManagers, wait for the all uploads to finish; do not start new uploads for this file. After ensuring that all uploads are done, continue to the next step. On any new upload request during this period, respond with **FILENOTFOUND** message.

   ii. Change its name to be *newname*

   iii. Update all servers using DONTCONTAINFILE (see 2.1) and CONTAINFILE (see 2.2) messages.

   iv. Print “Renaming Done”.

7) FIRESERVERS –send a **GOAWAY** (see 2.7) message to all known NameServers (including ones obtained from NameServers through the **WANTSERVERS** request). **F** should keep running.

   Note: You should not inform other NameServers and FileManagers about it and you shouldn’t delete the NameServers from any list (the reason for this is that the “firing” the server may be temporary, and this server might come back to life at some point (if someone reruns this server with the same address and port), and you do not want the file managers and other servers to lose the information about this server).

8) QUIT

   a) Wait for all ongoing uploads to finish (hint: the join() function), but do not start new ones or accept new commands.

   b) Notify all servers (2.6) that **F** is going down.

   c) Print “Bye-bye!” and exit.

Upload-request thread of the FileManager

F uses a second thread - the upload-request thread. This thread listens on the port of the FM (which it got from the command line) for requests from other FileManagers (see **ADD** command above). When a request arrives, it creates a **separate** upload thread and sends the corresponding file (**FILE** message) to the requesting FileManager. A **FILENOTFOUND** message should be sent if the requested file is no longer in **F**'s database or, in case the file is currently being downloaded by F from another file manager (i.e. F doesn't have it yet as he's still trying to download it).

If communication failed for other reasons **F** should send an **ERROR** message.

Comments:

1. When the user terminates a FileManager (using the QUIT command), the FileManager should wait for all the threads it created to finish before exiting.
2. A FileManager can upload the same file to many different FileManagers at the same time. Each such upload should be done in its own thread.
3. No other message should be sent\printed by FileManagers or NameServers.

Protocol

The protocol is the language via which a server and a client should communicate. In order to communicate with other FileManagers\NameServers your programs should strictly follow the given protocol. The protocol is defined by a set of messages names (which were mentioned above such as **BEGIN** or **WANTFILE**, etc) and by the expected order of the messages (for example after a FM sends a **BEGIN** message it expects to get **WELCOME** or **DONE** but no other message).

A whole message is composed of at least two words:

- The first word is the **MessageName** string (such as "BEGIN" or "WANTFILE", etc)
- The last word is a **MessageEnd** string (which is always the string "END"). This is part of the encoding of the entire message, for each message (see examples below).
- There may be additional words which come between the **MessageName** and **MessageEnd,** depending on the message type. For example
    - **CONTAINFILE** message is followed by a string representing a file name
    - **CONTAINNAMESERVER** is followed by a string and an integer standing for an ip and a port).

Messages Format

Here is the detailed message format for each of the existing messages (the different **MessageName** options). The [A B C] string means order A->B->C and **not** that A B and C should be separated with spaces or some other separator (see examples below).

- ***ERROR, DONE, WELCOME, GOAWAY, GOODBYE, FILENOTFOUND, WANTSERVERS, ENDLIST, ENDSESSION, WANTALLFILES:***
  [MessageName MessageEnd]
- ***BEGIN, FILEADDRESS:***
  [MessageName FileManagerIP FileManagerPort MessageEnd]
- ***CONTAINFILE, DONTCONTAINFILE, WANTFILE, NSCONTAINFILE:***
  [MessageName FileName MessageEnd]
- ***CONTAINNAMESERVER:***
  [MessageName NameServerIP NameServerPort MessageEnd]
- ***FILE:***
  [MessageName FileContents MessageEnd]

In some cases the information is transferred as a list (of NameServers, FileManagers or FileNames). For example when a NameServer announces which files are within the databases of its FileManagers, it sends several **NSCONTAINFILE** messages, one per file. When finishing this report, it sends **ENDLIST** message. When transferring a list of files (such as in the case of **NSCONTAINFILE** or **CONTAINFILE)** the files should be transferred alphabetically (sorted by file name); When transferring a list of NameServers, the list should be supplied in order of their acquisition by FileManager\NameServer. Again, each list should end with **ENDLIST** message, empty list should only contain **ENDLIST** message.
Note that when a NameServer sends a list of addresses (using **FILEADDRESS**) it doesn't wait for a response from the FileManager until sending **ENDLIST** (see examples below).

Data Types

The above mentioned messages are transferred between NameServers\FileManagers using sockets and streams. Some notes:
First, all messages should be decoded by DataInputStream (and encoded by DataOutputStream).
Second, the "additional information" coming as part of the messages (such as a file name or a port number) may be of different data types:

- **MessageName, MessageEnd, IPAddress, FileName, NameServerIP, FileManagerIP** in messages should be encoded as strings and should be transferred using readUTF()\writeUTF() (these are methods of DataInputStream\DataOutputStream).
- **FileManagerPort, NameServerPort** in messages should be encoded as an integer and transferred using writeInt()\readInt() (again, methods of DataInputStream\DataOutputStream).
- **FileContents** in messages should be encoded as Number of bytes (encoded as long and transferred by readLong()\writeLong()) followed by sequence of bytes (transferred by read()\write())

Examples

Example 1

In order to send **FILEADDRESS** message, you should do (sequentially)

```
myDataOStream.writeUTF("FILEADDRESS");  \\ sending  MessageName
myDataOStream.writeUTF(someIPString);   \\ sending  IP address
myDataOStream.writeInt(somePort);       \\ sending port number
myDataOStream.writeUTF("END");          \\ sending MessageEnd
```

and in order to receive such message you should do:

```
String msgName = myDataIStream.readUTF(); \\ and check if it is "FILEADDRESS"
String msgIP = myDataIStream.readUTF();   \\ reading a String
int msgPort = myDataIStream.readInt();    \\ reading an int
String msgEnd = myDataIStream.readUTF();  \\ and check if it is "END"
```

Example 2

In order to send a list of 3 files, do:

```
myDataOStream.writeUTF("CONTAINFILE");  \\ sending  MessageName
myDataOStream.writeUTF("afile1");       \\ sending  1st  file name
myDataOStream.writeUTF("END");          \\ sending MessageEnd
myDataOStream.writeUTF("CONTAINFILE");
myDataOStream.writeUTF("bfile2");       \\ sending  2nd file name
myDataOStream.writeUTF("END");
myDataOStream.writeUTF("CONTAINFILE");
myDataOStream.writeUTF("cfile3");       \\ sending  3rd file name
myDataOStream.writeUTF("END");
\\ the sent files are alphabetically ordered
\\ in addition, since this is a list, it should be followed by the ENDLIST message:
myDataOStream.writeUTF("ENDLIST");    \\  sending MessageName (“ENDLIST”)
myDataOStream.writeUTF("END");        \\ sending MessageEnd
```

Example 3

An example of a communication session between a FileManager **F** (IP 123.123.123.0 and port 3000) and a NameServer **N**. **F**'s database contains only the files “file1” and “file2” and the serverlist file that **F** received when created contains the single row: “123.123.123.0%2222”.

```
F: BEGIN 123.123.123.0 3000 END
N: WELCOME END
F: CONTAINFILE file1 END
N: DONE END
F: CONTAINFILE file2 END
N: DONE END
F: ENDLIST END
N: DONE END
F: CONTAINNAMESERVER 123.123.123.0 2222 END
N: DONE END
F: ENDLIST END
N: DONE END
F: ENDSESSION END
N: DONE END
```

Example 4
An example of a communication session between a FileManager **F** and a NameServer **N**, after **N** got WANTFILE message from **F** (say **F** requested the file “file3” and this file resides within the DB of two FileManagers which hold the ports 3001 and 3002, both on IP 123.123.123.0).
N: FILEADDRESS 123.123.123.0 3001 END
N: FILEADDRESS 123.123.123.0 3002 END
N: ENDLIST END
F: ENDSESSION END
N: DONE END

Note that **F** doesn’t reply with DONE after getting FILEADDRESS message. Only **N** should reply with DONE\ERROR\FILENOTFOUND\some list...

Example 5
Sending the File message (you may assume that your file size can be encoded using a single long)

```
1) writeUTF("FILE")
2) writeLong(fileSize)
3) write(byteFromFile) x fileSize times
4) writeUTF("END")
```

In order to perform step 3 you should open a FileInputStream for the file and read from it using read() . Then you can write (step 3) each byte read.

The opposite operation would be:

```
1) readUTF()                     \\ should be "FILE"
2) readLong()                    \\ size part of file contents (like fileSize above)
3) read()   x fileSize times     \\ many times depending on the received size above
4) readUTF()                     \\ should be "END"
```

Failures

A communication session may fail because no response from the other party is obtained for some time – a timeout period (see *setSoTimeout* in Socket). The default timeout in this exercise is 5 seconds (so you should assign *setSoTimeout* for all created Socket\ServerSocket objects with the value of 5 seconds). Such timeouts for **client sockets** should be treated in the same way as regular I\O communication errors (this could be either sending an ERROR message and\or closing the socket, depending on the exact scenario). Note that after a timeout, **server sockets** may become closed. So you may need to re-initialize server sockets in this case if you wish to continue working with them (re-initialize here means that you may need to initialize your ServerSocket object with a new instance which listens to the required port, and repeat this more than once). In any case make sure that after finishing working with sockets you should close related streams and sockets regardless of outcome or thrown exceptions. You are advised to use **finally** statements to do it.

Note that close() method of sockets and streams throws an IOException. You should ignore such failures (surround with try and an empty catch and add a blocks comment that explains why it’s left empty) and continue to the next action. In addition, remember to close streams before sockets.

Implementation

Packages and main file names

You should implement the NameServer in a class called *MyNameServer.java*. This class and all other NameServer-related files should be in the package *oop.ex3.nameserver***.** You should implement the FileManager in a class called *MyFileManager.java*. This class and other FileManager-related files should be in the package *oop.ex3.filemanager***.** You may (and are definitely advised) to create other packages and classes (in any of the packages) for your design.

Simplifying assumptions

You can assume (and do not need to verify) that

- There are no links or subdirectories in the directory where the set of files of a FileManager is kept.
- The command line parameters are valid (the port number, server list file and directory exist).
- The data in the server list file is in a valid format.
- A file name (without path) identifies the file uniquely – same name always means same file.
- Different FileManagers always use different directories as the database.
- FileManagers and NameServers IP and ports do not change during the run.
- Several NameServers and FileManagers may run on the same computer at the same time (and thus share the same IP address). However, they will have different ports. Also, as noted above, FileManagers will have different directories.
- While the program is running, there will be no external modifications of the content of FileManagers' database (i.e. no manual add\remove\rename operations).
- In the server list file (given as an input argument to FileManagers) the names of servers may be "localhost" or ***127.0.0.1*** another IP number (but not URLs).
- All user commands (such as ADD file, REMOVE file, etc) will be given in a valid format.

However, you should remember that (and have to deal with):

- A file may exist in the databases of several FileManagers.
- Not all FileManagers know all NameServers
- NameServers\FileManagers may be externally killed\restarted at any time, including during requests and file downloads (you can assume that there should be more than five seconds between turning some server\manager down and re-running it on the same port). If a NameServer fails to respond to a request, a FileManager should not wait for it for more than the default timeout (5 seconds), and just continue to the next NameServer on the list. However, the FileManager may still send requests to this NameServer in the future (on next requests), since it has not been erased from its list.
- Note that when a NameServer is restarted, it has no information about FileManagers and files. So, it will respond to the first request from any FileManager with a WELCOME message. You should handle such requests properly.
- Though the format of the serverslist file (see running instructions of FileManager) is valid, each NameServers on the list may go down at any point. In this case some or all of communication session with this NameServer will fail.

- Failure to communicate with some NameServer or FileManager should not cause the program to exit. On any connection failures the algorithm described above should be followed.
- There definitely can be dozens of FileManagers and NameServers in the system
- Your program should work with **any FileManager\NameServer** which follows the protocol, including the school solution. This means that when you build a server or a manager it should be able to communicate with any other manager or server which follows the described protocol.
- All orders which require some changes in NS data or state should be done only if the session ended successfully. For example when a FM begins a session of introducing itself using BEGIN and then sends its data (of files and servers), the NS should store the information about the files and other servers <u>only if the entire session ended successfully</u> and if, for example, an error occur during the server name sending, N should not keep any of files sent during this session.

Synchronization and threads

- In addition, when a NameServer or a FileManager receive a shut down order they should not assume that all their threads are finished - you should use join() in order to wait for the end of the execution of alive threads.
- In some other cases you may find it useful to use some basic synchronized\volatile mechanics (for example, for accessing a shared data structure from different NameServer threads).

OOP

Dealing with message types and communication scenarios should follow open-closed principle: You should be able to add new message types\new scenarios\new commands with minimal effort and code changes. Describe in the README how your system allows addition of (1) a new message type (2) a new command\communication scenario.

Networking

While the proposed system is stable enough, there are many ways to interfere\delay\crash your system using malevolently implemented external FileManager\NameServer. Describe in the README at least two ways that can cause your system to crash\become unresponsive by connecting to it with some external implementation of FileManager or NameServer. Propose (you **don't have to implement**) modification of your system which would allow to avoid each of these attacks.

Some clarifications

- All upload requests are not blocking - even if there are several requests for same file they all should be processed in parallel.
- All user commands (such as ADD, REMOVE, etc) are blocking - no other user command should be processed before the previous command has ended. For example, when the user requests a file he has to wait for the download to finish and no other commands should be processed until the file is downloaded or it is assured that there is no file.
- Error handling: in case an error occurs, from the NS point of view, the NS should reply ERROR and close the session. From the FM point of view, the FM should close the session.

- Handling exiting scenario (GOAWAY message at NS side or QUIT command and FM side) – don't use system.exit() in order to finish the execution of FM\NS. Instead you should just return from the top level methods (i.e., main() method or run() method), after closing everything you need to close, and finishing everything you need to finish without starting new operations.
- An FM sends a BEGIN message for each communication session with an NS. In other words, one BEGIN per one socket creation.
- A NameServer never sends requests to FileManagers. Only FMs send requests and the NSs reply to each request with DONE\ERROR\FILENOTFOUND\list of FMs or NSs
- A NameServer doesn't produce output.
- FileManagers interact only via the messages FILE\WANTFILE\FILENOTFOUND. They don't update each other about changes that occur in their database, etc.

School solution

- Many scenarios are easier to understand if you run school solution. You may run server and clients in different terminals and watch after their work.
- Running the school solution:

```
java -cp ~oop/for_students/school_solutions/ex3 oop.ex3.nameserver.MyNameServer port
java -cp ~oop/for_students/school_solutions/ex3 oop.ex3.filemanager.MyFileManager params
```

- To see additional explanatory messages you should add "demo" as additional command line parameter for school solution server and\or manager. For example:

```
oop.ex3.fileManager.MyFileManager ./serverlist1.txt ./mydir1/ 3001 demo
oop.ex3.nameserver.MyNameServer 2222 demo
```

- For testing purposes, you may want to use a timeout which is longer than 5 seconds. This is an additional parameter that the school solution can get: a number after the demo parameter, serves as timeout (in miliseconds), both for FileManager and NameServer. Obviously you don't have to implement it.

  For example using a timeout of 15 second:

```
oop.ex3.fileManager.MyFileManager ./serverlist1.txt ./mydir1/ 3001 demo 15000
oop.ex3.nameserver.MyNameServer 2222 demo 15000
```

- In addition you are advised to run your server with school solution client and your client with school solution server in various scenarios. Your code will be tested using out client \ server.
- Though the school solution produces several outputs as a response to invalid arguments, you should only print the strings specified in the exercise description.

The pre-submission script \ submission \ forums \ grading \appeals policy is the same as in Ex1-2.

Good Luck!