technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 4

M. Sc.-Inf. Dimitri Scheftelowitsch
M. Sc.-Math. Alexander Frank

Winter term 2016/17

Computer networks and distributed systems
Programming exercise

Publications: December 12, **Discussion:** January 10 - January 13, **Deadline January 4**

1 Problem

A. Nonymous knows enough dark corners on the internet, and would like to create an own dark corner in the coming year. For this to happen, he decides to set up a *textboard* with a protocol of his own. A textboard is, as the name may tell, a text-based internet forum; textboards were rather popular in Japan in the early age of the internet. The basic functionality of a textboard is to store a set of text messages, and to receive and relay new text messages.

This exercise concerns itself with writing a TCP-based protocol for a textboard server. The server should implement the functionality described above; for the implementation the protocol described below has to be used.

2 Protocol: General information

The communication protocol between the server and the client is line-based, i. e., a request message is always segmented in lines. As the protocol serves to transmit text messages, an important fragment is the format of a message. A message consists of multiple lines, where the first line contains the *time* and the *topic* of the message. The time of a message is a positive integer which is the number of seconds since January 1 1970 UTC until the reception of the message (also known as the *Unix timestamp*). The format of a message is then

```
<Number of lines>
<Time> <Topic>
<Text>
```

For example, a message can look like this

```
3
0 A question to biologists
Hi, I have stumbled upon a one-horned squirrel.
Does someone know if it is related to the common Central European unicorn?
```

A list of n messages is being sent in the following way.

```
n
<Message 1>
...
<Message n>
```

An example of such a list could be

```
2
2
360 A question to biologists
Hi, I also have seen a one-horned squirrel. It looked at me in a condescending manner.
2
360 Meeting IRL
Does someone else go to the Nuclear Meltdowns concert on Saturday?
```

The time of a message is, again, always the time of its arrival on the server. If a client sends a message, the time value may be arbitrary.

3 Protocol: Client part

The client side may send the following requests.

- `W <Time>` returns a list of a messages that have been sent since `<Time>` (seconds since January 1 1970 UTC).
- `P` posts a list of messages (that begins in the next line) to the textboard.
- `T <Topic>` returns a list of messages that have `<Topic>` as topic, sorted by time in descending order (new ones first).
- `L <Number>` returns a list of `<Number>` topics that were most recently posted to. If `<Number>` is empty, then all topics are returned. The format for a list of topics is related to the format for a list of messages and has the following structure.

  ```
  N
  <Time of the last message in topic 1> <Topic 1>
  <Time of the last message in topic 2> <Topic 2>
  ...
  <Time of the last message in topic N> <Topic N>
  ```

- `X` closes the session. After an `X` is received, the server closes the connection.

4 Protocol: Server part

The server concerns itself with the correct message storage and interpretation of the clients' requests. When a client publishes a message, the server must write to all connected clients (when it is done with processing their requests) a message of the form

```
N <Number of new messages>
<Time of the first new message> <Topic of the first new message>
...
<Time of the last new message> <Topic of the last new message>
```

The server must be able to process several parallel connections. On incorrectly formed requests it must reply with a message

```
E <error message>
```

A request is incorrectly formed if cannot be interpreted within the protocol or does not comply with the described format.

Below you can find examples foir requests and replies. The prefix `>` symbolizes communication from the client to the server, the prefix `<` symbolizes communication from the server to the client. Important: The symbols `>`, `<`, and the space after them do not belong to the protocol!

```
> P
> 1
> 2
> 0 Maintenance
> Tomorrow we will shutdown the textboard for maintenance operations.
< N 1
< 1481101380 Maintenance
```

```
> W 1481101380
< 1
< 2
< 1481101380 Maintenance
< Tomorrow we will shutdown the textboard for maintenance operations.

> T This topic does not exist
< 0

> L
< 3
< 1481101380 Maintenance
< 1481101164 A question to biologists
< 1481101164 Meeting IRL

> HELP
< E Bad request
```

5 Organizational remarks

The solutions may be turned in in Java or, on request and with permission of the teaching assistant, in a different programming language.

5.1 Submission

Submission is managed over AsESS. The solutions may be submitted arbitrarily often before the deadline is reached, only the latest submitted revision is considered.
The following files have to be submitted.

- An archive (.zip) with all source files (in Java: `*.java` files)
- readme.txt: A manual how to compile and start (for Java: `javac`/`java`) your program in the command line
- for backup, all source files that belong to your application

5.2 General approach

Prior to coding, make sure you understand the relevant parts of the (Java) standard library API and consider how you will use them in your code. Develop a model that contains all important elements for your application to function, and only once the model is complete, begin coding.

5.3 Comments and documentation

Comment your sources. Submissions without sensible comments will be graded less points. Furthermore you should document operational modes of your application, especially the command line parameters and control commands to ease testing. It is allowed to describe the commands on application strartup.

5.4 Error handling

The program has to handle exceptions (resp. error states) that may occur while connecting or message transmitting. Insufficient error handling will lead to point deduction.

5.5 Libraries

Only Java libraries from `java.*` and `javax.*` are allowed, for other programming languages only the base/standard library. Additional libraries and code from foreign sources is not allowed.

6 Implementation hints

Here, we shall introduce basic programming techniques for Java that may be helpful for this exercise.

6.1 Sockets

To avoid re-implementation of the network stack (Ethernet, IP, UDP, TCP) by every programmer who has to use the network interface, there exists a software abstaction layer that is called *Socket*.

Java has two kinds of sockets: *Sockets* and *ServerSockets*. These two behave differently.

Sockets are those sockets over which actual messages can be sent. To establish a connection, the following code can be used.

```
Socket mySocket = new Socket();
mySocket.connect(
  new InetSocketAddress( "123.234.111.100", 12345 )
  );
```

This establishes a connection to a computer with an address `123.234.111.100`. The port is the second part of the address over which th correct application on the other side can be selected.

To communicate with text messages, helper objects are needed in Java. These are *PrintWriter* for sending and *BufferedReader* for receiving. The *read functions* in the *BufferedReader* class are blocking. This means that they make the program wait until there is something to read. This behavior is given in the following example.

```
Socket mySocket = new Socket();
mySocket.connect(
  new InetSocketAddress( "123.234.111.100", 12345 )
  );
PrintWriter out = new PrintWriter(
      mySocket.getOutputStream(), true
      );
BufferedReader in = new BufferedReader(
      new InputStreamReader(
        mySocket.getInputStream()
      ) );
```

ServerSockets serve to wait for incoming connections. A *ServerSocket* must be bound to a local port.

```
ServerSocket ssocket = new ServerSocket();
ssocket.bind( new InetSocketAddress( 12345 ) );
```

The *accept* method is a blocking method that waits until there exists an established connection. If a connection is established, the method returns with a *Socket* which can be used for communication with the other side.

```
Socket client = ssocket.accept();
```

As the network interface is part of the operating system, communication with the operating system is needed. This means that each socket "lives" on the operating system level and the operating system must be informed that the connection is not needed anymore. All sockets, ServerSockets including, must hence be closed. Otherwise they stay open even after the application has been shut down. Opening the socket again is then not possible, even if the application is started again. Thus, do not forget to

```
mySocket.close();
```

respective

```
ssocket.close();
```

Further information on sockets can be found in "Learning Java" chapter 13, or other Java literature of your choice.

6.2 Threads

Threads are program parts that run in parallel. They are needed if the program, sayt, waits on a socket for incoming connections or incoming messages. While the program waits, nothing else can be done. This is also known as *blocking* behavior. A thread can help by executing the wait in parallel. A thread can be created in Java in the following manner.

```
public class MyThread extends Thread
{
  @Override
  public void run{
    /* do something useful */
  }
}
```

The method `run()` is the "main" method of the thread. Furthermore, threads behave like usual Java classes. The following code snippet creates a thread and starts it.

```
public class Program
{
  public static void main( String[] args ){
    MyThread t;
    t = new MyThread();
    t.start();
  }
}
```

Be advised that a thread must be started in order to begin working. A further extension of *MyThread* is to insert a main loop.

```
public class MyThread extends Thread
{

  private boolean _terminate;

  MyThread()
  {
    _terminate = false;
  }

  public void terminate()
  {
    _terminate = true;
  }

  @Override
  public void run
  {
    while ( !_terminate )
    {
      /* do something useful */
    }
  }
}
```

It is important to note that the access from a thread to data of a different thread may lead to data inconsistency. For example, a thread may change a string while another thread is reading it. In this case, a nonsensical word may be read. This behavior is known as *race condition*, because two threads are "racing" for data access. In Java packages under `java.util.concurrent.*` data structures can be found that are protected against race conditions.

Further information can be found under "Learning Java" chapter 9, or other Java literature of your choice.

Warning!

Please turn in only personally written programs. Multiple submissions of the same solution (plagiarism) will be graded with 0 points and lead to non-passing.

Submission is performed via ASSESS under https://ess.cs.tu-dortmund.de/ASSESS/.

Submission can be done in groups of two to three, one-person submissions are allowed.