SPL181
Assignment 3

Published on: 26.12.2017
Due date: ~~14.1.2017~~ 19.1.2018
Responsible TA's: **Morad Muslimany**, **Matan Drory**

1 General Description

In this assignment you will implement an online movie rental service (R.I.P. Blockbuster) server and client. The communication between the server and the client(s) will be performed using a text based communication protocol, which will support renting, listing and returning of movies. Please read the entire document before starting.

The implementation of the server will be based on the **Thread-Per-Client (TPC)** and **Reactor** servers taught in class. The servers, as seen in class, do not support bi-directional message passing. Any time the server receives a message from a client it can reply back to that specific client itself, but what if we want to send messages between clients, or broadcast an announcment to all clients? The first part of the assignment will be to replace some of the current interfaces with new interfaces that will allow such a case. Note that this part changes the servers pattern and **must not know** the specific protocol it is running. The current server pattern also works that way (Generics and interfaces).

Once the server implementation has been extended you will have to implement an example protocol. We will implement the movie rental service over the User service text-based protocol. The User Service Text-based protocol is the base protocol which will define the message structure and base command. Given an implementation of the protocol we can implement many user service applications. The service you will build is a movie rental service. Since the service requires data to be saved about each user and available movies for rental, we will implement a simple JSON text database which will be read when the server starts and updated each time a change is made.

Note that these kinds of services that use passwords and/or money exchange require additional encryption protocols to pass sensitive data. In our assignment we will ignore security and focus on network programming.

You will also implement a simple terminal-like client in C++. To simplify matters, commands will be written by keyboard and sent "as is" to the server.

2 User Service Text based protocol

2.1 Establishing a client/server connection

Upon connecting, a client must identify themselves to the system. In order to identify, a user must be registered in the system. The **LOGIN** command is used to identiy. Any command (except for **REGISTER**) used before the login is complete will be rejected by the system.

2.2 Message encoding

A message is defined by a list of characters in UTF-8 encoding following the special character '\n'. This is a very simple message encoding pattern that was seen in class.

2.3 Supported Commands

In the following section we will entail a list of commands supported by the User Service Text-based protocol. Each of these commands will be sent independently within the encoding defined in the previous section. (User examples appear at the end of the assignment)

Annotations:

- <x> – defines mandatory data to be sent with the command
- [x] – defines optional data to be sent with the command
- "x" – strings that allow a space or comma in complex commands will be wrapped with quotation mark (more than a single argument)
- x,… - defines a variable list of arguments

Server commands:

All **ACK** and **ERROR** message may be extended over the specifications, but the message prefix must match the instructions (reminder: testing and grading are automatic).

1) **ACK [message]**

The acknowledge command is sent by the server to reply to a successful request by a client. Specific cases are noted in the Client commands section.

2) **ERROR <error message>**

The error command is sent by the server to reply to a failed request. Specific cases are noted in the Client commands section.

3) **BROADCAST <message>**

The broadcast command is sent by the server to all **logged in** clients. Specific cases are noted in the Client commands section.

Client commands:

1) **REGISTER <username> <password> [Data block,…]**

Used to register a new user to the system.

- Username – The user name.
- Password – the password.
- Data Block – An optional block of additional information that may be used by the service.

In case of failure, an ERROR command will be sent by the server: ERROR registration failed
Reasons for failure:

1. The client performing the register call is already logged in.
2. The username requested is already registered in the system.
3. Missing info (username/password).
4. Data block does not fit service requirements (defined in rental service section).

In case of successful registration an ACK command will be sent: ACK registration succeeded

2) **LOGIN <username> <password>**

Used to login into the system.

- Username – The username.
- Password – The password.

In case of failure, an ERROR command will be sent by the server: ERROR login failed
Reasons for failure:

1. Client performing LOGIN command already performed successful LOGIN command.
2. Username already logged in.
3. Username and Password combination does not fit any user in the system.

In case of a successful login an ACK command will be sent: ACK login succeeded

3) SIGNOUT

Sign out from the server.

In case of failure, an ERROR command will be sent by the server: ERROR signout failed
Reasons for failure:

1. Client not logged in.

In case of successful sign out an ACK command will be sent: ACK signout succeeded
After a successful ACK for sign out the client should terminate!

4) REQUEST <name> [parameters,…]

A general call to be used by clients. For example, our movie rental service will use it for its applications. The next section will list all the supported requests.

- Name – The name of the service request.
- Parameters,.. – specific parameters for the request.

In case of a failure, an ERROR command will be sent by the server: ERROR request <name> failed
Reasons for failure:

1. Client not logged in.
2. Error forced by service requirements (defined in rental service section).

In case of successful request an ACK command will be sent. Specific ACK messages are listed on the service specifications.

3 Movie Rental Service

3.1 Overview

Our server will maintain two datasets with JSON text files. One file will contain the user information and the other the movie information. More about the files and the JSON format in the next section. A new user must register in the system before being able to login. Once registered, a user can use the login command to identify themselves and start interacting with the system using the **REQUEST** commands.

Note: the movie rental service is a "protocol in a protocol". Your implementation should consider this fact such that it will be as easy as reconfiguring the REGISTER command additional information and implementing the different sub requests. Adding a different service should be easy given implementation of part 2. Your design should consider this fact.

3.2 Service REGISTER data block command

When a REGISTER command is processed the user created will be a normal user with credit balance 0 by default.

The service requires additional information about the user and the data block is where the user inserts that information. In this case, the only information we save on a specific user that is recieved from the REGISTER command is the users origin country.

REGISTER <username> <password> country="<country name>"

3.3 Normal Service REQUEST commands

The REQUEST command is used for most of the user operations. This is the list of service specific request and their response messages. These commands are available to all logged in users.

1) **REQUEST balance info**

Server returns the user's current balance within an ACK message: ACK balance <balance>

2) **REQUEST balance add <amount>**

Server adds the amount given to the user's balance. The server will return an ACK message: ACK balance <new balance> added <amount>

Note: the new balance should be calculated after the added amount. You may assume amount is always a number greater than zero.

3) REQUEST info “[movie name]”

Server returns information about the movies in the system. If no movie name was given a list of all movies’ names is returned (even if some of them are not available for rental). If the request fails an ERROR message is sent.

Reasons of failure:

1. The movie does not exist

If the request is successful, the user performing the request will receive an ACK command: ACK info <”movie name”,…>.
If a movie name was given: ACK info <”movie name”> <No. copies left> <price> <”banned country”,…>

4) REQUEST rent <”movie name”>

Server tries to add the movie to the user rented movie list, remove the cost from the user’s balance and reduce the amount available for rent by 1. If the request fails an ERROR message is sent.

Reasons for failure:

1. The user does not have enough money in their balance
2. The movie does not exist in the system
3. There are no more copies of the movie that are available for rental
4. The movie is banned in the user’s country
5. The user is already renting the movie

If the request is successful, the user performing the request will receive an ACK command: ACK rent <”movie name”> success. The server will also send a broadcast to all logged-in clients: BROADCAST movie <”movie name”> < No. copies left > <price>

5) REQUEST return <”movie name”>

Server tries to remove the movie from the user rented movie list and increase the amount of available copies of the movies by 1. If the request fails an ERROR message is sent.

Reasons of failure:

2. The user is currently not renting the movie
3. The movie does not exist

If the request is successful, the user performing the request will receive an ACK command: ACK return <”movie name”> success. The server will also send a broadcast to all logged-in clients: BROADCAST movie <”movie name”> <No. copies left> <price>

3.4 Admin Service REQUEST commands

These commands are only eligible to a user marked as admin. They are meant to help a remote super user to manage the list of movies. Any time a normal user attempts to run one of the following commands it will result in an error message.

1) **REQUEST addmovie <"movie name"> <amount> <price> ["banned country",…]**

The server adds a new movie to the system with the given information. The new movie ID will be the highest ID in the system + 1. If the request fails an ERROR message is sent.

Reason to failure:

1. User is not an administrator
2. Movie name already exists in the system
3. Price or Amount are smaller than or equal to 0 (there are no free movies)

If the request is successful, the admin performing the request will receive an ACK command: ACK addmovie <"movie name"> success. The server will also send a broadcast to all logged-in clients: BROADCAST movie <"movie name"> <No. copies left> <price>

2) **REQUEST remmovie <"movie name">**

Server removes a movie by the given name from the system. If the request fails an ERROR message is sent.

Reason to failure:

1. User is not an administrator
2. Movie does not exist in the system
3. There is (at least one) a copy of the movie that is currently rented by a user

If the request is successful, the admin performing the request will receive an ACK command: ACK remmovie <"movie name"> success. The server will also send a broadcast to all logged-in clients: BROADCAST movie <"movie name"> removed

3) REQUEST changeprice <”movie name”> <price>

Server changes the price of a movie by the given name. If the request fails an ERROR message is sent.

Reason to failure:

1. User is not an administrator
2. Movie does not exist in the system
3. Price is smaller than or equal to 0

If the request is successful, the admin performing the request will receive an ACK command: ACK changeprice <”movie name”> success. The server will also send a broadcast to all logged-in clients: BROADCAST movie <”movie name”> <No. copies left> <price>

4. JSON

In this assignment, we will work with the JSON format.

4.1 JSON

The movie rental store will keep data about its customers and its warehouse using the JSON format. You can read about JSON at http://json.org and see general examples at http://json.org/example.html
In JAVA, we recommend using the **GSON** package to read, parse and write in JSON format. It is recommended to generate a java class to represent each JSON file as seen in this example.
In our modern networking world, servers and clients send each other data using JSON format all the time. Therefore, it is a good idea to have an understanding and training with JSON for the important experience it gives.

4.2 Our JSON data

In this assignment, we will have two JSON files that are in the server-side. One is "Users.json", which stores information about the customers registered to the online store. The other is "Movies.json", which stores information about the warehouse, i.e. movies that the online store offers and information about them.
Every change in the state of the store must be updated into the files (movie rented, movie returned, movie removed, user registered etc.)

4.3 Users.json example

Please see the supplied file **example_Users.json**
The file implies that the store currently contains 3 users:

1. User "john", an admin, with password "potato", from the United States, no movies rented and has a $0 balance.
2. User "lisa", a normal user (customer), with password "chips123", from Spain, currently has (by rent) the movies "The Pursuit of Happyness" (movie id 2) and "The Notebook" (movie id 3), and has a balance of $37.
3. User "shlomi", a normal user (customer), with password "cocacola", from Israel, currently has (by rent) the movies "The Godfather" (movie id 1) and "The Pursuit of Happyness" (movie id 2), and has a balance of $112.

4.4 Movies.json example

Please see the supplied file **example_Movies.json**
The file implies that the store currently contains 4 movies:

1. The movie “The Godfather”, of price 25, which is banned in both the United Kingdom and Italy. The immediate amount available for rental is 1, and the total number of copies the store owns is 2 (but one of them is currently rented by the user shlomi as seen in the previous Users.json file)
2. The movie “The Pursuit of Happyness”, of price 14, which is not banned in any country. The immediate amount available for rental is 3, and the total number of copies the store owns is 5 (but two of them are currently rented by users shlomi and lisa)
3. The movie “The Notebook”, of price 5, which is not banned in any country. The immediate amount available for rental is zero (none), and the total number of copies the store owns is 1 (it is rented by lisa)
4. The movie “Justice League”, of price 17, which is banned in Jordan, Iran and Lebanon. The immediate amount available for rental is 4, and the total number of copies the store owns is 4 (no one is renting the movie currently)

Note: you may assume movie prices and user balance is an integer.

5 Implementation Details

5.1 General Guidelines

- The server should be written in Java. The client should be written in C++ with BOOST. Both should be tested on Linux installed at CS computer labs.

- You must use maven as your build tool for the server and Makefile for the C++ client.

- The same coding standards expected in the course and previous assignments are expected here.

5.2 Server

When the server starts listening for new clients (ServerSocket bounded) it **must** print Server started to the screen. This will tell the automatic testing systems that we can start the simulation. Without it the assignment will receive a failing grade!

You will have to implement a single protocol, supporting both the ***Thread-Per-Client*** and ***Reactor*** server patterns presented in class. Code seen in class for both servers is included in the assignment wiki page. You are also provided with 3 new or changed interfaces:

- **Connections** – This interface should map a unique ID for each active client connected to the server. The implementation of Connections is part of the server pattern and not part of the protocol. It has 3 functions that you must implement (You may add more if needed):
    - **boolean send(int connId, T msg)** – sends a message T to client represented by the given connId
    - **void broadcast(T msg)** – sends a message T to **all** active clients. This includes clients that has not yet completed log-in by the User service text based protocol. Remember, Connections<T> belongs to the server pattern implemenration, not the protocol!.
    - **void disconnect(int connId)** – removes active client connId from map.

- **ConnectionHandler<T>** - A function was added to the existing interface.
    - **Void send(T msg)** – sends msg T to the client. Should be used by **send** and **broadcast** in the **Connections** implementation.

- **BidiMessagingProtocol** – This interface replaces the MessagingProtocol interface. It exists to support peer to peer messaging via the Connections interface. It contains 2 functions:
    - **void start(int connectionId, Connections<T> connections)** – initiate the protocol with the active connections structure of the server and saves the

owner client's connection id.
    - **void process(T message)** – As in MessagingProtocol, processes a given message. Unlike MessagingProtocol, responses are sent via the *connections* object **send** function.

Left to you, are the following tasks:

1. Implement ***Connections<T>*** to hold a list of the new ConnectionHandler interface for each active client. Use it to implement the interface functions. Notice that given a connections implementation, any protocol should run. This means that you keep your implementation of *Connections* on T.
   public class ConnectionsImpl<T> implements Connections<T> {…}.

2. Refactor the **Thread-Per-Client** server to support the new interfaces. The *ConnectionHandler* should implement the new interface. Add calls for the new *Connections<T>* interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol<T>.

3. Refactor the **Reactor** server to support the new interfaces. The *ConnectionHandler* should implement the new interface. Add calls for the new *Connections<T>* interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol<T>.

4. **Tasks 1 to 3 MUST not be specific for the protocol implementation**. Implement the new *BidiMessagingProtocol* and *MessageEncoderDecoder* to support the User service text based protocol as described in section 1.2. You will also need to define messages(<T> in the interfaces). You may add more classes as neccesery to implement the protocol (shared protocol data ect…).

<u>Leading questions:</u>

- Which classes and interfaces are part of the Server pattern and which are part of the Protocol implementation?
- When and how do I register a new connection handler to the **Connections** interface implementation?
- When do I call **start** to initiate the connections list? **Start** must end before any call to **Process** occurs. What are the implications on the reactor?
- How do you collect a message? Are all packet types collected the same way?
- How do I implement **BROADCAST**? as it should send a message to all **logged-in** clients (unlike the ***broadcast*** method in Connections that sends a message to all active users).

Testing run commands:

- Reactor server**:**
  mvn exec:java -Dexec.mainClass="bgu.spl181.net.impl.BBreactor.ReactorMain" -Dexec.args="<port>"
- Thread per client server**:**
  mvn exec:java -Dexec.mainClass="bgu.spl181.net.impl.BBtpc.TPCMain" -Dexec.args="<port>"

The **server** directory should contain a **pom.xml** file, a **Database** directory (for Movies.json and Users.json) and the **src** directory. Compilation will be done from the server folder using:
mvn compile

5.3 Client

An echo client is provided, but its a single threaded client. While it is blocking on stdin (read from keyboard) it does not read messages from the socket. You should improve the client so that it will run 2 threads. One should read from keyboard while the other should read from socket. Both threads may write to the socket. The client should receive the server's IP and PORT as arguments. You may assume a network disconnection does not happen (like disconnecting the network cable).

The client should recive commands using the standard input. Commands are defined in previous sections. The client should print to screen any message coming from the server (**ACK**'s, **ERROR**'s and **BROADCAST**'s). Notice that the client should not close until he recives an **ACK** packet for the **SIGNOUT** call.

The **Client** directory should contain a src, include and bin subdirectories and a Makefile as shown in class. The output executable for the client is named **BBclient** and should reside in the bin folder after calling **make**.

Testing run commands: **bin/BBclient <ip> <port>**

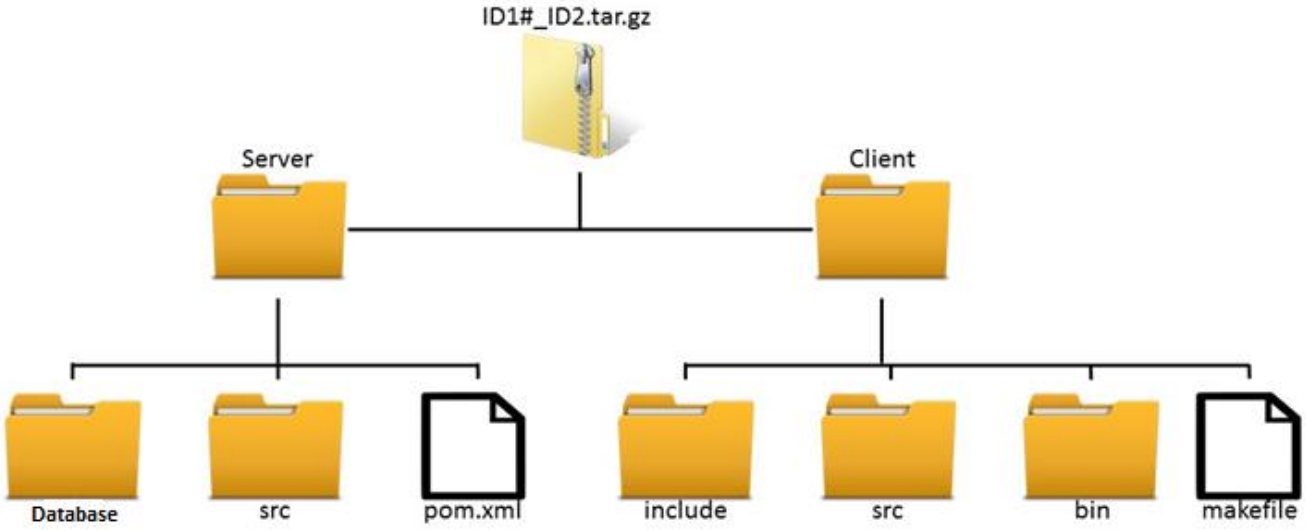
6 Submission instruction

- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You must submit one .tar.gz file with all your code. The file should be named "#ID1_#ID2.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- Extension requests are to be sent to majeek. Your request email must include the following information:

- Your name and your partners name.
- Your id and your partners id.
- Explanation regarding the reason of the extension request.
- Offcial certification for your illness or army drafting.

Requests without a compelling reason will not be accepted

- The submitted file should contain a **Client** directory and a **Server** directory (Their content was explained in the implementation section).



7 Examples

The following section contains examples of commands running on client. It assumes that the software opened a socket properly and a connection has been initiated.
We use “>” for keyboard input and “<” for screen output at the client side only. Server and client actions are explained in between.
Assume that the starting state of the Server is as presented in the example database files shown in section 3. (and it is the case from an example to another example, ie, each example starts from that state)
Always remember to update Movies.json and Users.json if updates are needed as specified in the assignment, even if there is no note in the examples below that mentions that.

7.1 Failed register, login, balance and movie info, rent and return a copy

Further assumptions:

- The current client is not logged in yet.
- The user shlomi is not logged in.

```
> REGISTER shlomi tryingagain country="Russia"
< ERROR registration failed
(registration failed because the username shlomi is already taken)
> REQUEST balance info
(server checks if the user is logged in)
< ERROR request balance failed
(it failed because the user is not logged in)
> LOGIN shlomi mahpass
(server checks user-pass combination)
< ERROR login failed
(it failed because the password is wrong)
> LOGIN shlomi cocacola
< ACK login succeeded
> REQUEST balance info
< ACK balance 112
> LOGIN shlomi moipass
< ERROR login failed
(this client is already logged in as shlomi)
> REQUEST info
< ACK info "The Godfather" "The Pursuit Of Happyness" "The Notebook" "Justice
League"
> REQUEST info "The Notebook"
< ACK info "The Notebook" 0 5
> REQUEST rent "The Notebook"
```

< ERROR request rent failed
(it failed because there are no available copies)
> REQUEST rent "Justice League"
< ACK rent "Justice League" success
(at this point the file Users.json is updated that
shlomi has rented "Justice League", his balance
is lowered from 112 to 95 and the file
Movies.json is updated that there is one less copy
available of Justice League)
< BROADCAST movie "Justice League" 3 17
> REQUEST balance info
< ACK balance 95
> REQUEST changeprice “The Notebook” 22
< ERROR request changeprice failed
(because shlomi is not an admin)
> REQUEST return "The Notebook"
< ERROR request return failed
(because shlomi does not own The Notebook)
> REQUEST info "The Godfather"
< ACK info "The Godfather" 1 25 "united kingdom" "italy"
> REQUEST return "The Godfather"
< ACK return "The Godfather" success
< BROADCAST movie "The Godfather" 2 25
> REQUEST balance info
< ACK balance 95
< BROADCAST movie “The Godfather” removed
(an admin, which is not the current user, removed The Godfather from the available movies)
> SIGNOUT
< ACK signout succeeded
(client’s app closes at this stage)

7.2 Successfully registered, add balance, try to rent a forbidden movie in the country

Further assumptions:

- The current client is not logged in yet.

> REGISTER steve mypass country="iran"
< ACK registration succeeded
(remember to update Users.json)
> REQUEST balance info
< ERROR request balance failed
(it failed because the user has not logged in yet)

> LOGIN steve mypass
< ACK login succeeded
> REQUEST balance info
< ACK balance 0
> REQUEST balance add 50
< ACK balance 50 added 50
< BROADCAST movie "The Godfather" 2 25
(some user, which is not the current user, rented or returned The Godfather)
> REQUEST rent "Justice League"
< ERROR request rent failed
(because Steve is from Iran and Justice League is banned there)
> SIGNOUT
< ACK signout succeeded
(client's app closes at this stage)

7.3 Admin: a simple example

- The client is not logged in yet
- The admin (user john) is not logged in

> LOGIN john potato
< ACK login succeeded
> REQUEST remmovie "The Godfather"
< ERROR request remmovie failed
(because The Godfather has a copy rented by shlomi)
> REQUEST remmovie "Justice League"
< ACK remmovie "Justice League" success
(succeeds because no one has rented this movie yet)
< BROADCAST movie "Justice League" removed
(remember that even the admin is a user, that's why he received a broadcast as well)
> REQUEST addmovie "South Park: Bigger, Longer & Uncut" 30 9 "Israel" "Iran" "Italy"
< ACK addmovie "South Park: Bigger, Longer & Uncut" success
< BROADCAST movie " South Park: Bigger, Longer & Uncut" 30 9
> SIGNOUT
< ACK signout succeeded
(client's app closes at this stage)