SPL - 151 Assignment 4

Boaz Arad; Itay Azaria ; Adiel Ashrov

Published on: 4/1/2015
Due date: 24/1/2015 23:59

1 General Description

Please read the **whole** assignment before you start your work.
This assignment is all about networking, and to get your hands on the network, you will implement a simple version of WhatsApp. WhatsApp is a real time messaging app where everyone can open a user and send messages, receive messages,and create groups. You will implement your WhatsApp in two steps:
1) write a WhatsApp client, 2) write your very own WhatsApp server.

Your WhatsApp client will receive commands from the user and send them to the server. For example your user can use her WhatsApp client to send a message to a group. The server's job in this case it to send the message to all the users registered to that group. This is a rough description and we will extend on these matters later on. The client and server connect to each other using the TCP/IP framework, much like you have seen in class. TCP/IP is how you set a *phone line* between two machines. Once there is a connection between the two we have to decide what is the *communication protocol* used for communicating. A *protocol* is a set of rules for message exchange between computers. In a real life phone call the *protocol* can be an exchange of *Hello* messages and then each side speaks while the other side listens. The *communication protocol* you will implement in this assignment is the HTTP protocol[1], which resides above TCP/IP.

2 Simplified HTTP protocol

2.1 Overview

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.
HTTP is a simple interoperable protocol designed for asynchronous message passing between clients via mediating servers. It defines a text based wire-format for messages passed between these clients and servers.
We will use a simplified version of the HTTP protocol in our assignment for passing messages between the client and the server. This section describes the format of HTTP requests/responses.

The HTTP specification defines *requests* and *responses* for server/client communication. Requests are divided into two typed: *GET* requests for receiving data and *POST* requests for sending data to the server. The server must acknowledge both GET and POST request with a reply, which can contain additional information.
Real HTTP clients/servers always include a "Content-Length" header in order to denote the amount of bytes expected before the end of a request/response. In our simplified version of the protocol, the "Content-Length" will not be required and the end of a message will be denoted by a "$" character.

[1] only the *messages* defined in the assignment should be supported

2.2 HTTP Requests

The basic HTTP request format is as follows:

```
<RequestType> <RequestURI> <HTTPVersion>
<HeaderName_1>: <HeaderValue_1>
...
<HeaderName_n>: <HeaderValue_n>

<MessageBody>
$
```

In this assignment, $< RequestType >$ can be either "GET" or "POST", $< RequesrURI >$ specifies the address of the requested resource and $< HTTPVersion >$ should always be "HTTP/1.1". After the request line, a HTTP request may contain any number of headers with the format $< HeaderName >:< HeaderValue >$, for example "Accept: text/html", a complete list of acceptable headers can be found in the HTTP protocol specification.
GET Requests are used to receive data from the server and should not contain a $< MessageBody >$. When sending messages to the server, a POST request should be used. The $< MessageBody >$ of post requests follows the following format:

```
<name1>=<value1>&<name2>=<value2>&...<nameN>=<valueN>
```

Each $< name >$ should consist only of alpha-numerical characters, and each $< value >$ must be URL-ENCODED, you may use the Java class *URLEncoder* to do so.

2.2.1 HTTP Responses

Each HTTP request must be acknowledged by a HTTP response, the basic HTTP request format is as follows:

```
<HTTPVersion> <StatusCode>
<HeaderName_1>: <HeaderValue_1>
...
<HeaderName_n>: <HeaderValue_n>

<MessageBody>
$
```

As with the HTTP requests, $< HTTPVersion >$ should always be "HTTP/1.1". The $< StatusCode >$ element is a 3-digit integer result code of the attempt to understand and satisfy the request. In our exercise, the following HTTP status codes will be valid:

- 200 - OK
- 403 - Forbidden
- 404 - Not Found
- 405 - Method Not Allowed
- 418 - I'm a teapot

Response headers are specified in a similar manner to request headers, and the $< MessageBody >$ may contain free text corresponding to the result of the request.

2.3 HTTP Server

A HTTP server is modeled as a set of resource URI's, which may be static or dynamic and can be supplied to clients on request. Additionally, HTTP servers may receive complex information from client requests, and change their state, and/or provide an appropriate resource accordingly.

Upon receiving a HTTP request, the server should first check if the specified URI exists on the server. If it does not, a "404" status code should be returned immediately. If the URI exists, the server proceeds to check whether the requesting user should be granted access to the resource, if not, a "403" status is returned. Finally, if the resource exists, and the user is authorized to access it, a "200" status should be returned along with the appropriate headers and response content.

3 WhatsApp Implementation

Using the HTTP protocol, we shall implement a client-server setup that will allow clients to send private and group messages to each other via the server. Clients will issue commands via HTTP requests, and receive acknowledgments and/or results as HTTP responses.

3.1 WhatsApp Server

3.1.1 Availible URI's

The WhatsApp HTTP server will provide the following URI's:

- login.jps - Used for authetication
- logout.jsp - Used to revoke authetication
- list.jsp - Used to list current users and groups
- create_group.jsp - Used to create groups
- send.jsp - Used to send messages
- add_user.jsp - Used to add users to groups
- remove_user.jsp - Used to remove users from groups
- queue.jsp - Used to query available messages

Acsess to all URI's, excluding "login.jsp" should be denied until a user has requested "login.jsp", once a user has requested "logout.jsp", he will again be denied access to all URI's excluding "login.jsp". Below you will find detailed information on the correct way clients should access each URI. If a client attempts to use GET to access a URI which should be accessed via POST (or vice versa), you may return a 405 status code. If a client accesses a resource with the correct method, but supplies partial/incorrect information in the request body - you *must* return a 200 response code, and specify the error in the message body.

3.1.2 URI Specifications

- `login.jsp`

  Accessed via POST
  Request Variables:
  UserName: $< UserName >$
  Phone: $< PhoneNumber >$

  If all headers are supplied correctly, should produce a success message and provide a cookie granting access to all other URI's for the user. Otherwise, an error message should be returned, and access to other URI's denied.

- `logout.jsp`

  Accessed via GET
  Upon access, revokes access to all other URI's, return a goodbye message.

- `list.jsp` Accessed via POST
  Request Variables:
  List:$< ListType >$

  Returns a list of either users or groups in the message body if $< ListType >$ is "Users", "Group" or "Groups", otherwise, returns an error message.

- `create_group.jsp` Accessed via POST
  Request Variables:
  GroupName:$< Name >$
  Users:$< User1 >$,$< User2 >$,...,$< UserN >$

  If all users are valid, create a new group including them with the name $< Name >$. Otherwise, or if the GroupName variable is missing, return an error.

- `send.jsp` Accessed via POST
  Request Variables:
  Type:$< MsgType >$
  Target:$< MsgTarget >$
  Content:$< Msg >$

  If all headers are present and valid, sends a message to a group or individual, otherwise returns an error. $< MsgType >$ can be "Group" or "Direct". If $< MsgType >$ is "Group" $< MsgTarget >$ should specify the target group name, If $< MsgType >$ is "Direct" $< MsgTarget >$ should specify the target users phone number. $< Msg >$ Should contain the message content.

- `add_user.jsp` Accessed via POST
  Request Variables:
  Target:$< GroupName >$
  User:$< PhoneNumber >$

  If all headers are present and valid, adds a user to a group, otherwise returns an error. $< GroupName >$ should be an existing group name, $< PhoneNumber >$ should be a phone number of an existing user. The user requesting the URI must be a member of $< GroupName >$ as well, otherwise an error will be returned.

- `remove_user.jsp` Accessed via POST
  Request Variables:
  Target:$< GroupName >$
  User:$< PhoneNumber >$

  If all headers are present and valid, removes a user to a group, otherwise returns an error. $< GroupName >$ should be an existing group name, $< PhoneNumber >$ should be a phone number of a user in the group. The requesting user must be a member of $< GroupName >$ as well, otherwise an error will be returned.

- `queue.jsp` Accessed via GET

  Provides a list of all messages sent to the user since his last access to queue.jsp.

3.1.3 Message Body Formats

Upon access, each URI should return a response, whether the requested action was successful or not. Below are the success/error formats for each URI, please follow the format exactly for each error, though you are responsible for deciding when to produce each error.

Note that you may assume the syntax for all requests/responses will be **correct**, though your client should be able to handle "correct" requests issued at an "incorrect" state (i.e. attempting to add a non-existent user to a group).

- login.jps
  - Welcome [UserName]@[PhoneNumber]
  - ERROR 765: Cannot login, missing parameters
- logout.jsp
  - Goodbye
- list.jsp
  - ERROR 273: Missing Parameters
  - [PhoneNumber1],[PhoneNumber2],...[PhoneNumberN]
  - [GroupName1]:[PhoneNumber1],[PhoneNumber2],...[PhoneNumberN]
[GroupName2]:[PhoneNumber1],[PhoneNumber2],...[PhoneNumberN]
...
[GroupNameN]:[PhoneNumber1],[PhoneNumber2],...[PhoneNumberN]
  - [UserName1]@[PhoneNumber]
[UserName2]@[PhoneNumber]
...
[UserNameN]@[PhoneNumber]
- create_group.jsp
  - ERROR 675: Cannot create group, missing parameters
  - ERROR 929: Unknown User [User]
  - ERROR 511: Group Name Already Taken
  - Group [GroupName] Created
- send.jsp
  - ERROR 771: Target Does not Exist
  - ERROR 836: Invalid Type
  - ERROR 711: Cannot send, missing parameters
- add_user.jsp
  - ERROR 669: Permission denied
  - ERROR 242: Cannot add user, missing parameters
  - ERROR 142: Cannot add user, user already in group
  - ERROR 770: Target Does not Exist
  - [PhoneNumber] added to [GroupName]
- remove_user.jsp
  - ERROR 668: Permission denied
  - ERROR 336: Cannot remove, missing parameters
  - ERROR 769: Target does not exist
  - [PhoneNumber] removed from [GroupName]

- queue.jsp
  - No new messages
  - From:[PhoneNumber/GroupName]
    Msg:[MessageContent]

    From:[PhoneNumber/GroupName]
    Msg:[MessageContent]

    ...

You may add additional error messages if you like.

3.1.4 Security

It is a common practice in web services to identify users via cookies. Cookies are strings of information provided in the message header of HTTP requests and responses. A web server may request its client record a coookie by providing a `Set-Cookie` header in its response. The client may later provide a cookie for the server by adding a `cookie` header to its HTTP request.
The format for cookie headers is described below, for setting a cookie, the server should add the following to its response headers:

```
Set-Cookie: name=value
```

The client may present a cookie to the server by adding the following to its request headers:

```
Cookie: name=value
```

Once a user requests `login.jsp` correctly, the server should reply with a unique cookie that will identify the user. The client should always present this cookie to the server when requesting other URI's, if a valid cookie is not provided, the server should produce a 403 error.
The cookie name for your authentication cookie should be `user_auth`. Keep in mind that your server will have to maintain a data structure containing all active cookies and their corresponding users.

3.1.5 Graceful Shutdown

The server should provide a way for safely shutting down. When an "`exit`" command is issued to the server console, it should send a shutdown message

3.2 WhatsApp Client

Your whatsapp client should provide a simple interface to your server. It must support the following commands:

- Login [Username] [Phone]
- Logout
- List Users
- List Groups
- List Group [Group Name]
- Send Group [Group Name] [Message]
- Send User [Phone] [Message]
- Add [Group Name] [Phone]

- Remove [Group Name] [Phone]
- Exit

Once the user issues a command to the client, the client should prepare an appropriate HTTP request for the server, and return the result. You may return the body of the result as-is, or parse it to be comfortably readable, but you should not include the response headers.
In addition to these commands that may be supplied by the user, after logging in, the client should periodically check whether the server has any new messages for it. If new messages are found on the server, they will be displayed immediately.
Checking for new messages should be performed *in a seperate thread* by the client, and the process should not interfere with user input, or displaying results of user queries. The polling rate of the client is up to you, but anything on the order of once every 1-2 seconds will suffice.

4 Exchange example

In order to clarify all the above, we will now provide you with an example of a possible exchange between your client and server.
First, the client requests `login.jps` using a `POST` request:

```
POST /login.jps HTTP/1.1

UserName=AldoRaine&Phone=495558298
$
```

The server, satisfied that the requirements for accessing `login.jps` have been met, will reply with a success message and a **unique** cookie generated by our super-secret secure algorithm (you may use any method you choose for generating user cookies, as long as they are unique):

```
HTTP/1.1 200
Set-Cookie: user_auth=9e234af900534eff

Welcome AldoRaine@495558298
$
```

Now that the client has received his authentication cookie, he may attempt to access other URI's, providing the cookie in his request header. Here he will attempt to access `queue.jsp` to check if he has any messages waiting:

```
GET /queue.jsp HTTP/1.1
Cookie: user_auth=9e234af900534eff

$
```

At which point the server will verify that the `user_auth` cookie is valid, and provide him with the appropriate reply.

5 Reactor

5.1 Overview

In this part of the assignment you will change the server implementation from the thread per client design pattern to the reactor design pattern. Except for the Message, ServerProtocol and the MessageTokenizer, the reactor classes are general and may serve other server types as well. In your implementation - keep it general as well.

5.2 A Reactor-based WhatsApp Server

Your objective is to build a WhatsApp server using the Reactor design pattern instead of the thread-per-client model. You should reuse as much as possible the existing code. Assuming you have designed your thread-per-client server correctly, the changes should be minimal.
You may base you submission on the Reactor provided in PS 12.

6 Submission Instructions

Submission is done only in pairs. If you do not have a partner, 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 "assignment4.tar.gz".
- Your compressed file must include the following directory structure:

  ```
  server\
  ...bin\
  ...src\
     ...protocol\
     ...protocol_http\
     ...protocol_whatsapp\
     ...threadPerClient\
     ...tokenizer\
     ...tokenizer_http\
     ...tokenizer_whatsaap\

  client\
  ...src\
  ...bin\
  ```

  Please note that all `bin` folders should be **empty** - do not submit any binary files. You may add additional folders if you like, but must conform to the execution example below.
- The `server` and `client` folders should contain an `ant` file and `makefile` respectively.
- Your submission will be run using the following commands: for
  - Client

    ```
    cd client
    make
    cd bin
    ./client [host] [port]
    ```
  - Server

    ```
    cd server
    ant compile jar
    cd bin
    java server [port]
    java reactor <port> <pool_size>
    ```