אוניברסיטת בן גוריון בנגב – המחלקה להנדסת מערכות תקשורת
רשתות תקשורת מחשבים 2 - קורס 37110211
מעבדה בתקשורת מחשבים

תרגיל תכנות	"Timeline" Application Socket Programming on C & Linux	
חלק 1	(TCP client) (TCP Server)	Communication with "p2p tracker" Web server, only GET support
חלק 2	(TCP Server) (UDP server/client)	Web server, add POST support Peer-to-peer posts send/receive
תאריך הגשה	חלק1: 18/01/2013, חלק 2: 25/01/2013	

הוראות הגשה

1. התרגיל חייב להיות ממומש ב- C תחת Linux.

2. עבור <u>כל חלק</u> עליכם להגיש קובץ zip נפרד שיכיל:
   א) קבצי מקור.
   ב) קובץ makefile.
   ג) מסמך word בן 1-2 עמודים המתאר באופן מפורט כל חלקי התוכנית (למעשה – מסמך design כנהוג בתעשייה)

3. התוכנית חייבת להתקמפל בעזרת הרצת "make" בלבד מתוך ספרייה המכילה קבצי מקור ו- makefile. נא לוודא זאת לפני ההגשה. <u>עבודות שלא יתקמפלו יקבלו 0</u>.

4. הניקוד יתחלק בצורה הבאה:
   א) עבודה נכונה של התוכנית – 80%.
   ב) מבנה התוכנית (מבנה הקוד) ודוקומנטציה (מסמך design) – 20%.
   ג) יורדו נקודות על כתיבה מרושלת של קוד (שמות ללא משמעות של משתנים ופונקציות, חוסר הערות במקומות שיש צורך בהם, בזבוז זיכרון וחוסר יעילות).

5. שם של קובץ executable חייב להיות "timeline". ההפעלה של התוכנית חייבת להיות בצורה הבאה: `timeline tracker_IP username web_port`

6. שימו לב: כשרשום post (באותיות הקטנות) – הכוונה לתמונה או הודעת טקסט שאנו רוצים לשלוח למשתמש אחר. כשרשום POST (אותיות גדולות) – הכוונה לפקודת POST של פרוטוקול HTTP.

Overview

You will be provided with the centralized server (called “p2p tracker” or just "tracker") which keeps information about every user that has logged in: *username*, IPs and listening port (`postPort`) for receiving posts (a post is a picture file or a text message). Since the program should perform many tasks concurrently, the implementation should involve multithreading or/and multiprocessing.

You will have to write a program called “timeline” that interacts with:

1. The tracker
2. Web Browsers
3. Other “timeline” instances.

The program “timeline” should have the following functionality:

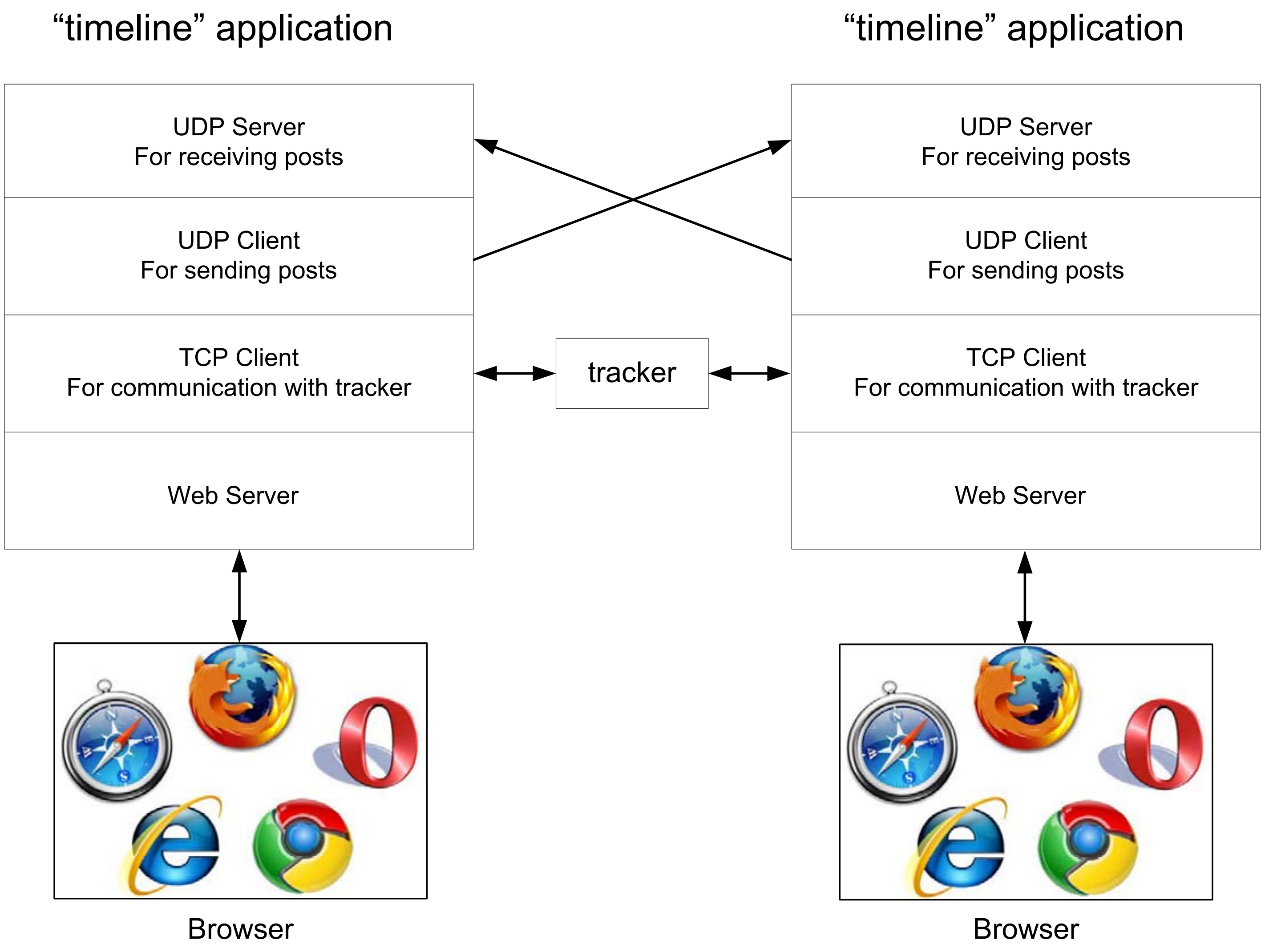
1. Communicating with p2p tracker:
   a. Connect to the tracker using TCP to port number `12345`.
   b. Log in to the tracker. During the login process you will send to the tracker your *username* and `postPort`.
   c. Periodically, every 10 sec, ask the tracker “Who is online?” Parse the tracker’s output and store the reply that will include *username*, IP address, and the `postPort` for each online user. If there is a change in the list of online users, update the `"UsersOnLine.txt"` file appropriately (the purpose of this file is explained in the “Web Server” section).

2. Web server functionally.
   a. Create a TCP listening socket on port `webPort`.
   b. A browser that connects to your web server will get a webpage (html file) with all the posts you have received from other "timeline" peers. Your web server should be able to serve all clients (web browsers) concurrently.
   c. The web server will also allow the browsers to send a post (picture or text message) to other “timeline” peers. The post will be uploaded from the browser to the "timeline" application which in turn will send the post to the user we selected using its peer-to-peer functionality.
   Once the "timeline" application is started (e.g., `timeline 132.72.108.180 311111111 82`), the only interface to it is a web browser that will connect to the web server of the "timeline" app.

3. Peer-to-Peer functionality:
   a. Send/Receive pictures and text messages (posts) to/from other peers. Peer-to-Peer communication will use **reliable** UDP communication (reliability will be achieved by implementing Stop&Wait above UDP).
   b. Note that the trigger for sending a post (picture or text message) to another peer will be your web server that will receive the post from the browser.
   c. When a UDP server receives a post from a peer, it should appropriately update the `“PostsOnMyWall.txt”` file (the purpose of this file is explained in the “Web Server” section).

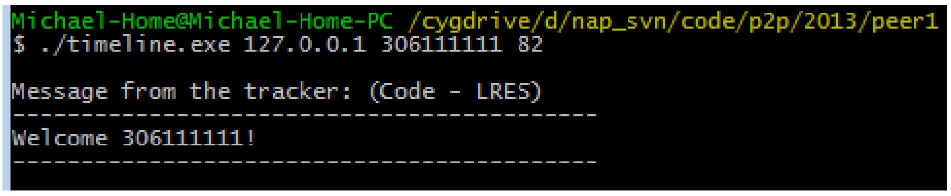
For the Part 1 of the programming assignment you have to implement 1.a, 1.b, 1.c and 2.a. I.e., the communication with the tracker, and a simple web server that will answer on HTTP GET requests by sending: the html file, some auxiliary text files, and locally stored pictures to the browser. You don't need to create the html file by yourself; the file `"myTimeline.html"` will be given to you.
In Part 2 of the assignment you will extend the functionality of the web server by adding it an ability to receive posts uploaded via the browser (2.b). Upon reception, these posts should be sent to the other peer using the p2p functionality (2.c) that you also should implement in Part 2.

The following figure illustrates the functionality blocks of the "timeline" application:



Some screenshots:

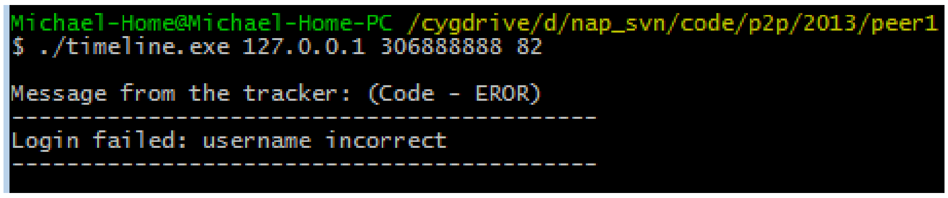
Starting the “timeline” application and printing the welcome message from the tracker:



```
Michael-Home@Michael-Home-PC /cygdrive/d/nap_svn/code/p2p/2013/peer1
$ ./timeline.exe 127.0.0.1 306111111 82

Message from the tracker: (Code - LRES)
-------------------------------------------
Welcome 306111111!
-------------------------------------------
```

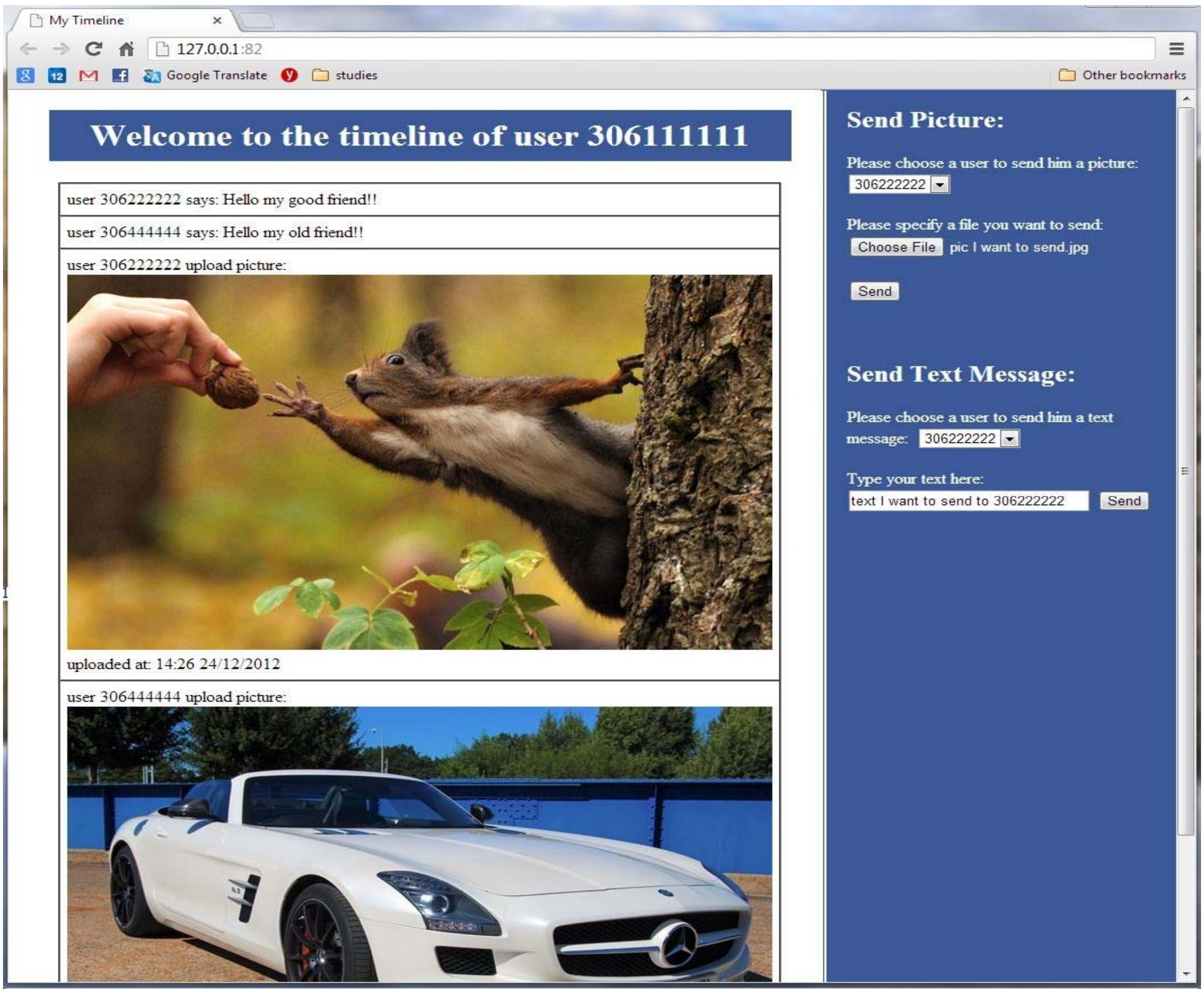
In case of incorrect username, error message from tracker should be printed:



```
Michael-Home@Michael-Home-PC /cygdrive/d/nap_svn/code/p2p/2013/peer1
$ ./timeline.exe 127.0.0.1 306888888 82

Message from the tracker: (Code - EROR)
-------------------------------------------
Login failed: username incorrect
-------------------------------------------
```

Accessing the “timeline” application with browser:



Welcome to the timeline of user 306111111

user 306222222 says: Hello my good friend!!

user 306444444 says: Hello my old friend!!

user 306222222 upload picture:

uploaded at: 14:26 24/12/2012

user 306444444 upload picture:

Send Picture:

Please choose a user to send him a picture: 306222222

Please specify a file you want to send: Choose File pic I want to send.jpg

Send

Send Text Message:

Please choose a user to send him a text message: 306222222

Type your text here:

text I want to send to 306222222 Send

Part 1

1.1 TCP client for communication with p2p server (tracker)

The protocol you will use for client-server communications is defined as follows. Each packet contains ASCII characters only, as with HTTP, so the messages being passed will be in human-readable format. The first four bytes of every packet contain a header value which determines the type of the packet. The data field varies for each type of packet (specifics listed below), but in general fields are split using the character '#' as a delimiter, and all packets are terminated with a '\n' character.

The following packet types should be used for the first part of this exercise.

Login Request

LGIN	**username (your ID)**	**#**	**postPort**	**\n**

- LGIN is the 4-byte header value corresponding to a login packet.
- Username - your ID without the leading 0 (if you have one).
- `postPort` – is a UDP port that you will use to listen for incoming picture uploads. This port will be calculated by your program as following: `(12312 + rand()%128)`.

When the tracker receives a LGIN packet from a client, it will return one of two responses. If the username is invalid, or if the LGIN packet does not conform to the protocol format, an EROR (error) packet will be returned:

Error Message

EROR	message	\n

If your "timeline" application receives an EROR packet from the tracker, it should print out the error message, close the socket, and exit.

If the login packet is received correctly, the server will instead generate a LRES (login response) packet containing a welcome message:

Login Reply (welcome message)

LRES	**message**	**\n**

After you successfully logged in to the tracker, start periodically, <u>every 10 seconds</u>, ask tracker "Who is online?" by sending the following packet:

"Who is online?" request

WHO?	**\n**

Search results are returned one at a time in "WHOR" packets as follows:

Tracker's reply to our "who is online?" request

WHOR	**username (ID)**	#	**IP**	#	**postPort**	**\n**

When the last search result has been returned (last online user in the p2p tracker's list), the tracker will return an empty WHOR packet:

Tracker's last reply to our "who is online" request

WHOR	**\n**

Notice, you have to store the information about the online users (*username*, IP and `postPort` for each user) in some data structure or in a file. This information will be used by the UDP Client – for getting the IP and Port of a user to whom we selected to send a post.

Moreover, once there is a change in the list of online users, you should appropriately update the file `"UsersOnLine.txt"` that is used by the web server – to display the list of online users in the dropdown menu.

1.2 WEB server

You will be provided with an HTML page – the file “`myTimeline.html`”. This file contains HTML code for the timeline web page. This file contains links to three special text files stored in your working directory:

1. `MyUserID.txt`
2. `PostsOnMyWall.txt`
3. `UsersOnLine.txt`

Your “timeline” program should update these files when required:

1. `MyUserID.txt` – On the program startup, the *username* (passed as a command line argument) should be written to this file.

2. `PostsOnMyWall.txt` – Once a new post is received by your UDP server (picture or text message), it should be appended to this file.

   In the case of a text message, append:
   `<tr><td>user` *`username`* `says: message_content</td></tr>`

   In the case of a picture, append:
   `<tr><td>user` *`username`* `uploaded picture:<br><img src="filename.jpg" </td></tr>`

3. `UsersOnLine.txt` – Once there is a change in the “online users” database that you maintain, the new list of currently online users should be written to this file.

For the Part 1 of the assignment you can manually add some example posts to “`PostsOnMyWall.txt`” and some pictures to your local directory, so that you will see them in a browser.

You are free to customize the design of your timeline web page, while keeping the required functionality.

The web server will listen on port `webPort` that will be supplied to the program as an optional argument. The option to change this port allows you to start multiple instance of the application on the same computer, thus allowing easy testing/debugging. It should be possible to connect to your server from a browser by typing in the address bar: `your_ip:webPort`.

Notice, the web server should be able to serve many customers concurrently. Moreover, when a browser detects links to images and the auxiliary text files (3 files listed above) on the HTML page you sent it (“`myTimeline.html`”), it opens concurrent connections for downloading them from your server. Thus, the implementation should involve multithreading and/or multiprocessing.

When a browser connects to your server (TCP three-way handshake + empty HTTP GET request), the server will check if the packet is an initial HTTP GET request (it should start with "`GET / HTTP/1.1`"), and will send back an HTML page – “`myTimeline.html`”.

For sending the HTTP packets you can use the following approach:

1. Use the minimal hardcoded HTTP header for sending an HTML page:
   `"HTTP/1.1 200 OK\r\nContent-Length: X\r\n\r\n"`
   X is the length of the HTML file. Don't forget to fill it. After the header, just send all the character of the HTML page.

2. Once a browser receives your HTML page, it will open new connections with your web server for downloading all the files (images and auxiliary text files) that were specified in the HTML page.

3. For each file a browser will send you an HTTP GET request. You need to distinguish between the initial HTTP GET request and requests for images. This can be done by inspecting the HTTP GET request.
   a. In the initial HTTP GET request, GET will be followed by "`/ `" (slash and space).
   b. In the HTTP GET request for a file, GET will be followed by "/filename".

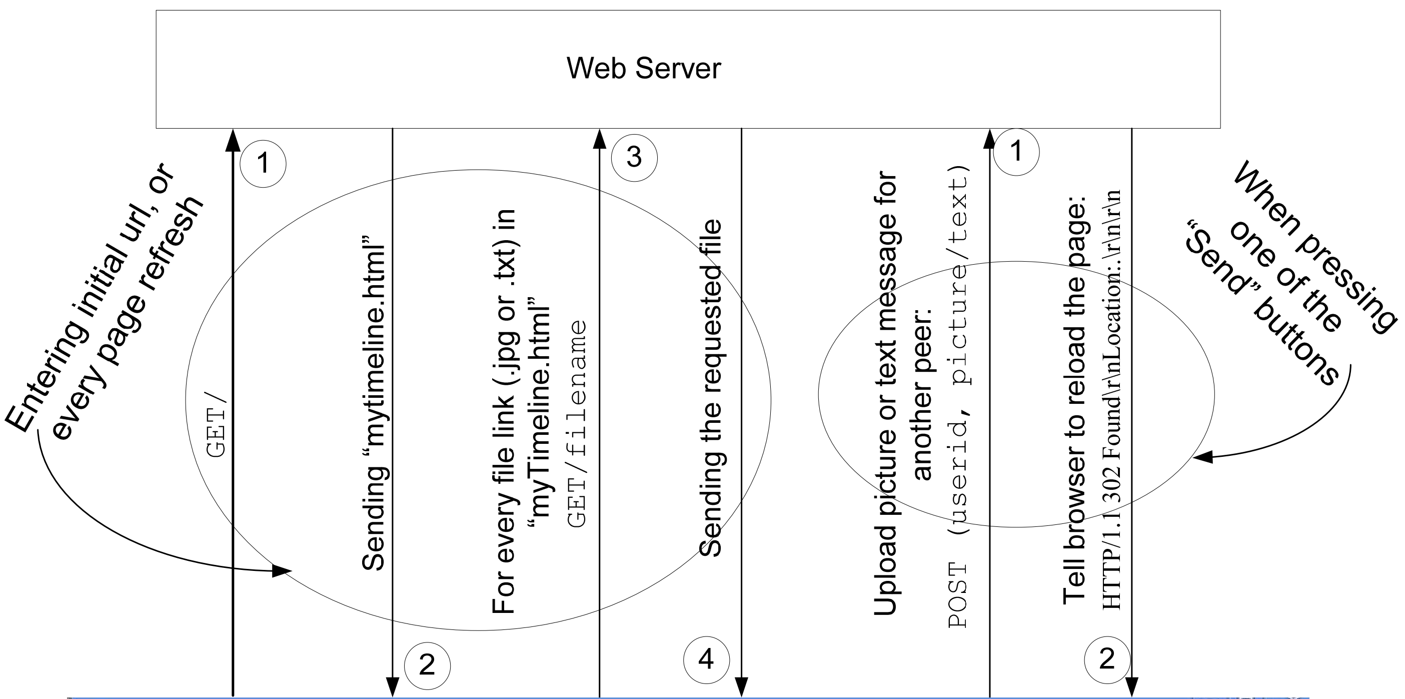
4. Use the minimal hardcoded HTTP header for sending a file to the browser:
   a. In case of a picture file:
      `"HTTP/1.1 200 OK\r\nContent-Type: image/gif\r\nContent-Length: X \r\n\r\n"`.
   b. In case of a text file:
      `"HTTP/1.1 200 OK\r\nContent-Type: text\r\nContent-Length: X \r\n\r\n".`

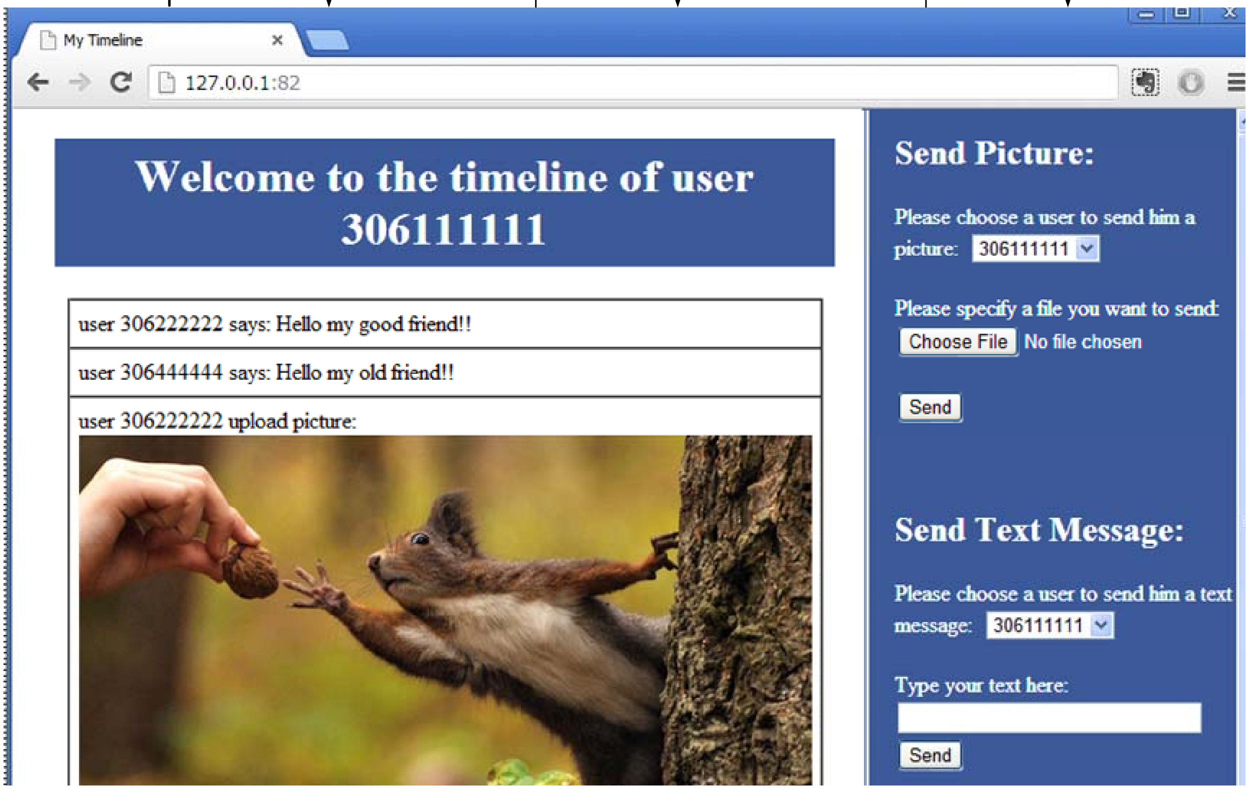
   X is the length of the file. After sending this header, just send all the bytes of the file.

5. In the Part 2, your web server should be able to accept POST requests from the browser. You have to figure out by yourself how to parse these POST requests and extract from them text messages, pictures, and the *usernames*. Wireshark will help you to analyze what exactly a browser sends.
   Tip: you should find in the HTTP header the attribute “boundary” (some text string) which is the limiter of the POST data (the boundary may differ from browser to browser), use it to extract your text or picture from the POST body.
   After receiving a post (picture or text message) from the browser, the web server should invoke the UDP client that will send the post to the specified peer.

6. Your server should answer to every POST message with HTTP redirect in order to make the browser to ask the main HTTP page again. For explanation why this is required see: http://en.wikipedia.org/wiki/Post/Redirect/Get.
   So, the answer to every browser’s POST should be just following message:
   `"HTTP/1.1 302 Found\r\nLocation:.\r\n\r\n".`

The following figure illustrates the interaction between a browser and your web server:


Web Server
Entering initial url, or every page refresh
GET/
1
Sending "mytimeline.html"
2
For every file link (.jpg or .txt) in "myTimeline.html"
GET/filename
3
Sending the requested file
4
Upload picture or text message for another peer:
POST (userid, picture/text)
1
Tell browser to reload the page:
HTTP/1.1 302 Found\r\nLocation:.\r\n\r\n
2
When pressing one of the "Send" buttons



Welcome to the timeline of user 306111111

user 306222222 says: Hello my good friend!!

user 306444444 says: Hello my old friend!!

user 306222222 upload picture:

Send Picture:

Please choose a user to send him a picture: 306111111

Please specify a file you want to send:

Send Text Message:

Please choose a user to send him a text message: 306111111

Type your text here:

Part 2

2.1 UDP server & client for sending/receiving posts to/from other peers

Posts transfer

The listening port of the UDP picture server will be `postPort` that you have calculated in Part 1.
The UDP server will handle only one post upload at a given time, so it will NOT be multithreaded or multiprocessed (of course, the UDP server itself should be a separate process/thread in your "timeline" application).

When a peer wants to sent a post to us, it will do so with the following message. This message will be sent in a special way as described in the following S&W section.

Posts Transfer

TYPE	#	Username	#	Post content

First three bytes of the message is the post type: "PIC" or "TXT".
Then, comes the *username* of the peer that is sending us the post.

Note that posts can be large and be composed of several UDP datagrams.

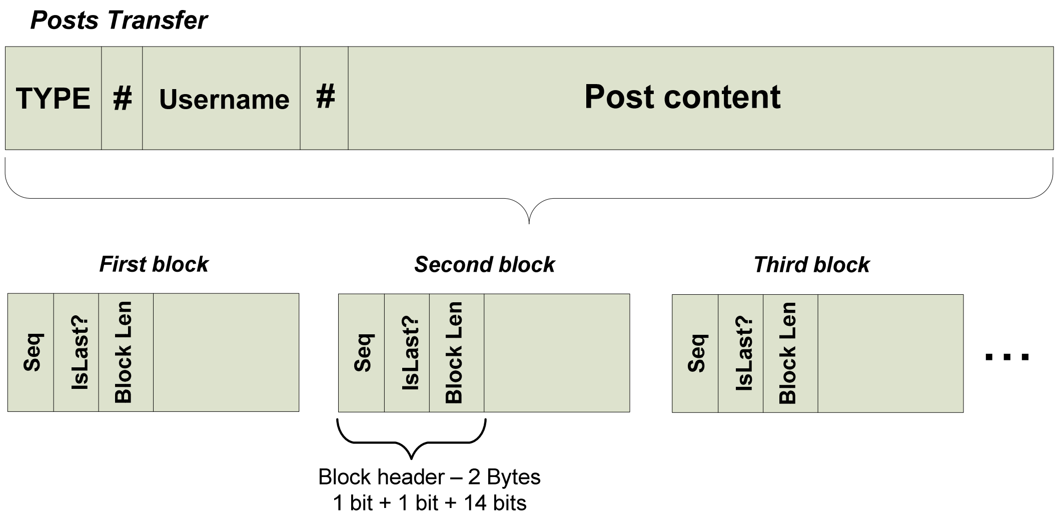
When receiving a picture post, you should create a file with the following name: "*username_hh_mm_ss.jpg*", where *username* is the user who sent you the file and *hh_mm_ss* is the time when the upload was finished. Once the picture file is stored, the UDP server should appropriately update the "`PostsOnMyWall.txt`" file (as described in the Web Server section).

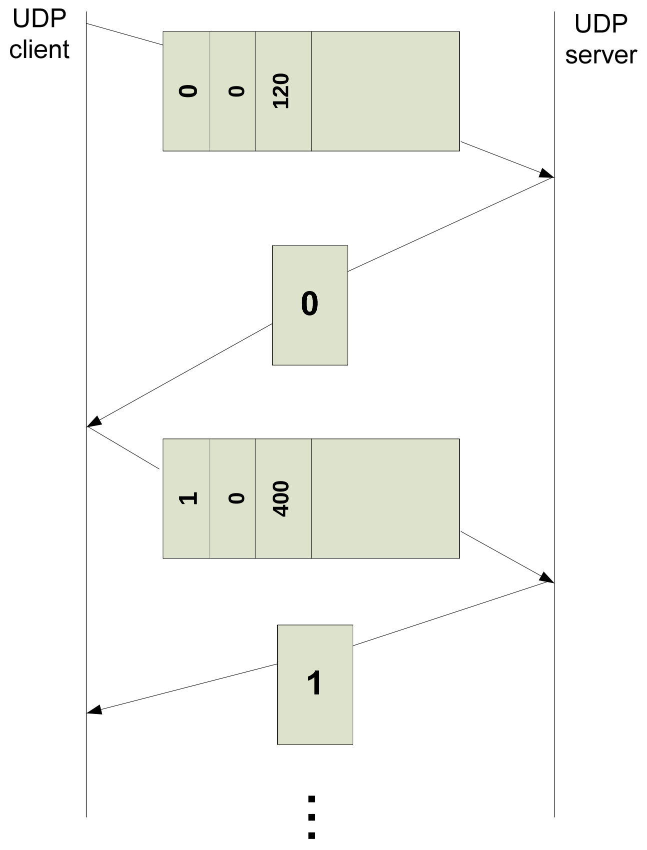
When receiving a text post, just write the received text to the "`PostsOnMyWall.txt`" file (as described in the Web Server section).

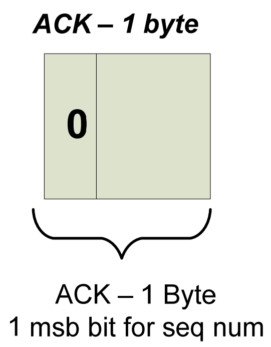
STOP and WAIT

Since the file transfer is done over UDP, it is unreliable. In order to overcome this, we will add the S&W functionality for sending "PIC/TXT" messages.

The "PIC/TXT" message will be sent by blocks of variable length. Each block will have a 2 Bytes header comprised of: 1 msb bit – sequence number, 1 bit – indicator whether the block is the last block of the post, and 14 lsb bits – length of the block in bytes (not including the 2 Bytes header).







At the UDP client side:

1) For sending blocks you can use some fixed value for the block size, but your server should support receiving blocks of any size.
2) Before writing the payload to the socket's buffer, append to it an appropriate 2 Bytes header.
3) After sending a datagram, wait for an acknowledgment for 500 msec. If no ACK is received, retransmit the same datagram. Implement the waiting for ACK using the `select()` function with timeout.
4) Note that the client must verify that the ACK arrived from the expected peer.
5) If after 5 retransmissions there is no ACK, stop the uploading and print an error message to the user.

At the UDP server side:

1) When a message is received, check from which `srcIP` & `srcPort` it arrived (use `recvfrom()`).
2) If currently there is no post reception in progress, start the reception.
   The reception should be initiated and the message processed (see below).
3) If there is a reception in progress:
   a. If the message came from another peer, it should be ignored (we serve only one peer at a time). Notice, a peer is identified by its IP and Port and not only IP, since we may start two peers on the same machine.
   b. If the message came from the expected peer, it should be processed.
   c. If for 2 sec there is no data packet from the expected peer, the reception should be cancelled.

<u>Reception initiation:</u>
1) The `srcIP` and `srcPort` should be saved.
2) A variable for the expected sequence number should be created.
3) In the case of a picture post, an appropriate file should be created.

<u>Message Processing (the peer is validated beforehand):</u>
1) If the received sequence number is the expected sequence number:
   a. The payload should be extracted and stored (for picture post it should be written to file).
   b. An ACK message should be sent to the sending peer. ACK is a single byte message with an msb bit indicating the sequence number (0 or 1).

2) If the received sequence number is **NOT** the expected sequence:
   a. The message should be ignored.
   b. After 5 consecutive out-of-order messages **from the sending peer**, the reception should be cancelled.

Once you've received a post, update the "`PostsOnMyWall.txt`" file appropriately.

Good Luck!