Assignment 3

Start:	23 October 2017
End:	2 November 2017

Objectives

In this assignment, you will get some experience with UDP communication. To highlight that UDP is connection-less and thus, the delivery and the sequence of the messages cannot be guaranteed, you will develop a mobile application. The mobile application will simulate a chat client that tries to communicate with other clients. The focus will be on understanding how UDP works and how to preserve causality and temporal ordering of the messages, in spite of the unreliability of the underlying UDP protocol (lossy channel and delays). You will implement two concepts to determine the order of events in a distributed system.

With this assignment you can gain 10 points out of the total 45.

Task	Points
1	2
2	4
3	2
4	2
Total	10

Format Warning Please refer to the instructions in this document regarding the formatting of the code and the format of the messages.

Lamport Timestamps represent a simple algorithm to partially order distributed events. The rules that this algorithm follow were determined by L. Lamport[1]. Distributed processes that implement Lamport timestamps satisfy the so-called *clock consistency condition*: if event A happens before event B, then event A's logical clock arrives before event B's. Therefore, if event A's logical clock comes before event B's logical clock, then A may have happened before or at the same time as B, but not after B.

Vector Clocks represent an extension of Lamport timestamps. They guarantee the *strong clock consistency condition* which, additionally to the clock consistency condition, dictates that if the clock of one event arrives before another, then, that event happened before the other, i.e., it is a two-way condition. This is achieved by holding a vector of n logical clocks in each process (where n is the number of processes) and by including these values in all inter-process communication.

To support this assignment, we provide you with a server that you can deploy on your local machine.

- The server must be started from the command line. Once started, it will output the IP address and the port on which you can communicate. The predefined port is **4446**.
- The communication protocol and the message structure is fixed. Please check the appendix of this sheet to understand how messages have to be constructed.

[1] Leslie Lamport - Time, clocks, and the ordering of events in a distributed system; ACM Communications Magazine, volume 21, issue 7, July 1978

- You can also test your chat client using the Android emulator. However, to communicate to the server you must use the following IP address **10.0.2.2**. Using this address, you can test your code on your own machine, without having to use the actual smartphones.

Starting the server

The server can be started from the command line. Use the following command:

```
java -jar chat_server.jar

// And the output

Server started
Server IP address : 192.168.192.38
Server port : 4446
```

Listing 1: Starting the server

1 Getting Familiar with Datagrams (2 Points)

To familiarize yourself with sending and receiving UDP messages, create an Android application that provides a service for client registration and deregistration. This is the first step towards building a chat-like application.

- The server provides a user registration and deregistration service. The client must send a message to the corresponding address and port.
- The UDP packet must follow a specific format which can be seen in the appendix.
- On successful registration or deregistration, the server will reply with an ACK message. Since UDP is connectionless and there is no guarantee on receiving the message, you must specifically handle such situations. Receiving a message is a blocking operation, which can be solved with socket timeout. If there is no message received within the specified time, the client will stop listening for new messages.
- As a result, one flow for registration (or deregistration) is the following: the client sends a "regsiter" message. The server receives the messages and replies with an "ack" message. The client must process this message and display a notification that the registration has been successful.
- A different flow is when the server is unavailable or the packets get dropped. The client sends a "register" message. The server does not reply. By having a timeout mechanism you can stop listening for new messages after a predefined time. When the time limit is reached, the client knows that there is no "ack" message from the server. In such a case, the client will initiate a new attempt to register. You must also implement this behavior in your application.
- The "register" and "deregister" flows are identical, the only difference is the type of the message.
- For this task you do not need to send any Vector or Lamport clocks. You can leave the field empty.

Activity

Create a new Android project called *vs-nethz-chat* and set the application name to *VS nethz chat* (where *nethz* is the group leader's nETHZ account name).

1. Your main activity should provide at least the following functionality: an *EditText* control for user input, a *Join* button and a *Settings* button.
2. The text input should be used for entering a username.
3. The *Join* button will send a "register" message to the server. On successful registration, the application must transition to a new activity. You will use this activity in Task 3. You can call this activity "ChatActivity". If registration is not successful from the first attempt, your chat client must automatically retry to register the user 5 times. If the registration is still not successful, your application must display a corresponding notification.
4. The *Settings* button will transition to another activity, called "SettingsActivity". Here the user should be able to set the server's address and the port.
5. You are free to implement the "deregister" functionality. You can either choose another button or deregister the user on a specific event (e.g. "Back" button pressed in the ChatActivity).

Sending and receiving UDP packets

For sending and receiving UDP messages you will use the *DatagramSocket* class (`http://developer.android.com/reference/java/net/DatagramSocket.html`).

- Socket initialization: *Datagram socket = new DatagramSocket(port);*
- Socket timeout: *socket.setSoTimeout(time);*
- Socket send: *socket.send(packetToSend);*
- Socket receive: *socket.receive(getack);*

For learning more about Datagrams please consult the following tutorial: `https://docs.oracle.com/javase/tutorial/networking/datagrams/`.

Generating a UUID

The messages that you send to the server must contain a unique identifier for each client. UUIDs, Universally Unique IDentifier, can be easily generated using the *UUID* class. You can consult the following URL for usage: `http://developer.android.com/reference/java/util/UUID.html`.

Hint: once the user has been registered on the server with a UUID, the client must use the same UUID for interacting with the server.

2 Implementing Vector Clocks and Lamport Timestamps (4 points)

In this task you have to implement Vector Clocks and Lamport Timestamps. This task does not require the server. To test your implementation, use the included unit tests. You must create two different classes, one called *LamportClock.java* and another one called *VectorClock.java*.

- Both classes that you create must implement the *ch.ethz.inf.vs.a3.clock.Clock* interface. For a more detailed description of the interface have a look at the JAVA file provided in the code skeleton.
- The interface describes the methods that are common to both implementations. Besides the described methods you must implement additional methods for both classes.
- For Lamport timestamps, you must implement the following additional methods: *void setTime(int time)* and *int getTime()*. *void setTime(int time)* overrides the current clock value with the one provided as input. *int getTime()* will return the current clock value.
- For Vector clocks you must implement the following additional methods: *int getTime(Integer pid)* and *void addProcess(Integer pid, int time)*. *int getTime(Integer pid)* will return the current clock for the given process id. *void addProcess(Integer pid, int time)* adds a new process and its vector clock to the current clock.

Lamport Timestamps and Vector Clocks implementation

When implementing the two classes, you can use the following data structures for representing the logical clocks:

- For Lamport timestamps use an integer. **private int time;**
- For Vector clocks use a map between two integers. **private Map<Integer, Integer> vector;**. For each *process id* you associate a *logical time*.

Running the unit tests

You must include an additional library in your project to run the unit tests from Android Studio. You must first create a new folder called "testlibs" in the */app* folder (on the same level with *src*). Copy the "json-20150729.jar" library to the "testlibs" folder. Then you must modify your build.gradle file and add the new dependencies.

```
1 // Unit tests dependencies
2 testCompile 'junit:junit:4.12'
3 testCompile files('testlibs/json-20150729.jar')
```

3 Message ordering based on Vector Clocks (2 points)

In this task, you will have to implement sorting based on Vector Clocks to determine the order of messages in distributed systems. We assume a chat client wants to join a conversation and would like to retrieve the conversation history. The server will send all the messages to the client, one by one. Given that the chat application uses UDP, there is no way to know if the order in which the messages arrive is the correct one. Your task is to implement an ordering mechanism for the incoming messages and to display the content of each message in your application.

- The client sends a "retrieve_chat_log" message to the server.
- The server will send back multiple messages to the client. However, pay attention that these messages are sent in a **random** order.

- The client must buffer all the received messages until no other message is received. This can be handled again using a socket timeout.
- Before displaying the messages, the client must sort the received messages using the attached vector clocks.

For this task, you will use the activity that you have created in Task 1. Once the user has been successfully registered, your application will transition to the "ChatActivity".

ChatActivity

Your activity must contain the following elements:

- A view to display messages. You are free to choose any user interface (for e.g., you can choose a simple TextView).
- A button called "Retrieve chat log". Once the user presses the button, a message is sent to the server. The server will send the client several messages that have to be displayed in the correct order.
- Once the messages have been sorted, the client can display them on the user interface.

Sorting messages

Due to the underlying protocol, the messages that are received from the server can arrive in random order. However, having vector clocks we can ensure that if there is causality between the events, we can display them in the correct order. Your task is to buffer all the messages that are sent from the server and sort them.

One way to sort the messages is using a PriorityQueue (provided in the code skeleton: **package: ch.ethz.inf.vs.a3.queue.PriorityQueue**). A PriorityQueue orders the elements according to their natural order or according to the comparator specified at construction time. If the queue uses natural ordering, only elements that are comparable are permitted to be inserted into the queue. Since the chat client works with messages, they do not have a natural ordering. The solution is to implement a *Comparator*.

You have to implement a class called *MessageComparator.java* that implements the *Comparator<T>* interface. Afterwards, you will have to override the *compare* method. You can then use the priority queue as a message buffer. Whenever a new message arrives, you insert it in the queue. Having a proper comparator, the messages will be inserted in the correct order.

To test the correct ordering of the vector clocks with unit tests, you have to implement an additional class called *VectorClockComparator.java*. This comparator can further be used in your *MessageComparator* implementation.

4 Mini-Test (2 Points)

As part of the assignment, you should answer these questions as briefly as possible. They are related to Task 2. Please recopy the questions and the numbering. Feel free to use either LaTeX or any other document writer, but the submission has to be a .pdf(**answers.pdf**). Please write all the group members' full names on the document.

1. What are the main advantages of using Vector Clocks over Lamport timestamps?

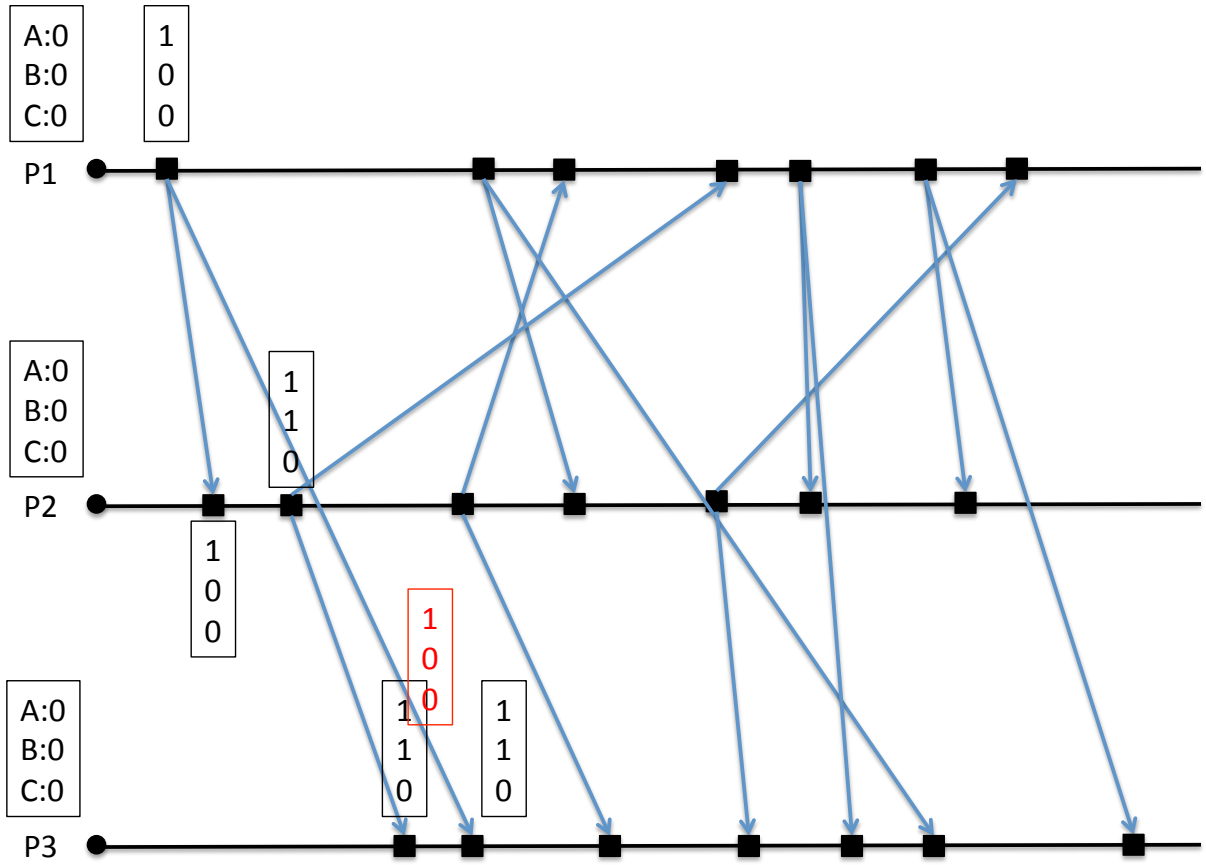

Figure 1: Fill in the corresponding vector clocks for the remaining events

2. Give the two conditions for two Vector Clocks to be causally dependent?

3. Does a clock tick happen before or after the sending of a message. What are the implications of changing this?

4. Fill in the corresponding vector clocks from Figure 1. In this example, the ticking only happens when sending a message. Your task is to write for each event (sending or receiving) the correct vector clock. In black you can see the vector clock for a sending or receiving event, while in red you can see the vector clock attached to the message.

   When filling in the vector clocks, you must write the correct vector clocks for sending or receiving events. At the same time, you must also fill in the vector clocks attached to each message (just like the red vector clock showed in the figure). Use two different colors and use the example as a guideline.

5. Read the paper *Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications* by Tobias Landes[2] that gives a good overview on the discussed methods. In particular, which problem of vector clocks is solved in the paper?

[2]http://vs.inf.ethz.ch/edu/vs/exercises/A3/DVC_Landes.pdf

Deliverables

The following two deliverables have to be submitted by **11:59 P.M., November 2nd, 2017**:

- **code.zip** You should create a zip file containing the Android Studio projects created in this assignment. The projects should have been tested both on the mobile phone and on the emulator. The code must compile on our machines as well, so always use relative paths if you add external libraries to your project. Do not forget to include those libraries in the zip file. Please use UTF-8 encoding for your documents and avoid special characters like umlauts (everything should be in **English**).
- **answers.pdf** the answers to Task 4 in **pdf** format.

Submission

Report and code must be uploaded through:

`https://www.vs.inf.ethz.ch/edu/vs/submissions/`

The group leader can upload the files, and other group members have to verify in the online system that they agree with the submission. Use your `nethz` accounts to log in. The submission script will not allow you to submit any part of this exercise after the deadline. However, you can re-submit as many times as you like until then.

Questions

Any e-mail is required to follow the exact following format, because your emails will be filtered to improve the latency for a response :

- `FROM`: nethz@student.ethz.ch (or the email you're registered to in the ETH system)
- `TO`: mihai.bace@inf.ethz.ch
- `SUBJECT`: [VS_2017] `group leader's nethz` - Topic

Final Remarks

Your apps have to handle the case where there is no connection by displaying an error message and not proceeding with the server interactions (which would fail anyway).

Protocol

The messages accepted by the chat server are describe below. Please read the specifications carefully and pay attention to the formatting. Any other message type will be ignored.

- **register**: the client must first send this JSON message to the server before any other interaction can be initiated. To register, there is no need to send a timestamp.

```
{
    "header": {
        "username": "John",
        "uuid": "ae4e15ff-b589-4e85-a07c-594b16e4e645",
        "timestamp": "{}",
        "type": "register"
    },
    "body": {}
}
```

Listing 2: Client's sends a "register" message

- **deregister**: the clients must send a message to the server if they want to deregister. The server keeps track of all the registered users. There is no need to send a timestamp.

```
{
    "header": {
        "username": "John",
        "uuid": "ae4e15ff-b589-4e85-a07c-594b16e4e645",
        "timestamp": "{}",
        "type": "deregister"
    },
    "body": {}
}
```

Listing 3: Client's sends a "deregister" message

- **retrieve chat log**: the clients wish to retrieve a chat log from the server, which they have to process before displaying.

On the server side:

- **ack**: the server will send an acknowledgement (ack) message to confirm a successful registration or deregistration.
- **error**: the server will send an error message when there is a problem decoding the message. The message type will be "error" and in the body there will be an integer number representing an error code. For a detailed description of each error code check class *ch.ethz.inf.vs.a3.message.ErrorCodes* from the code skeleton.

```
{
    "header": {
        "username": "John",
        "uuid": "ae4e15ff-b589-4e85-a07c-594b16e4e645",
        "timestamp": "{}",
        "type": "retrieve_chat_log"
    },
    "body": {}
}
```

Listing 4: Client sends a "retrieve_chat_log" message

```
{
    "header": {
        "username": "server",
        "uuid": "ac31f345-a8b1-4241-b939-9d3527f14483",
        "timestamp": "{}",
        "type": "ack"
    },
    "body": {}
}
```

Listing 5: Server "ack" message for successfully registering or deregistering a client

- **message**: when the client requests the chat log, once the authentication of the client has been carried out, the server will reply with several messages. The type of the message will be "message" and the body will contain text (e.g. "A2", "A1", etc.).

```
{
    "header": {
        "username": "server",
        "uuid": "ae4e15ff-b589-4e85-a07c-594b16e4e645",
        "timestamp": "{}",
        "type": "error"
    },
    "body": {
        "content":"4"
    }
}
```

Listing 6: Server sending an "error" message. The number from the content field represents an error code.

```
{
    "header": {
        "username": "server",
        "uuid": "ac31f345-a8b1-4241-b939-9d3527f14483",
        "timestamp": "{\"0\":2,\"1\":0,\"2\":0}",
        "type": "message"
    },
    "body": {
        "content":"A1"
    }
}
```

Listing 7: Server's response to "retrieve_chat_log" message