Assignment 3

Start:	20 October 2014
End:	3 November 2014

Objectives

In this assignment, you will get some experience with handling UDP communication. In order to highlight the fact that UDP was thought to be connection-less and thus, that the delivery and the sequence of the messages cannot be guaranteed, you will develop an n-person mobile chat application. This is not a chat service per-se but a way to broadcast messages. You should use this to focus on preserving the causality and the temporal order of 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 that you have learned about in the lecture.

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

Task	Points
1	2
2	6
3	2
Total	10

Format Warning Please refer to the instructions in this document regarding the formatting of the code and the exchanged messages with the server. When we refer to `nethz`, we are referring to the group leader's ETH account identifier. An automated scheme will extract and test your implementation. **All non-complying submissions will fail (beware of the exact hierarchy of your classes and folders and of the spelling of your packages)**!

Lamport Timestamps represent a simple algorithm to partially order distributed events. The rules that this algorithm follows 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 in that 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 messages.

To support this assignment, we have three servers running:

- **`129.132.75.194:4000`** provides a capitalization service: it returns every incoming message changed to upper case. This server is used to test UDP-based communication. It replies to the source port chosen by the client.

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

- For the chat exercise, a deployment and a test chat servers are available. They both distribute all messages received from a registered client to all other registered clients. Each server randomly delays messages to simulate communication unreliability inherent in real systems. While they are listening for incoming datagrams (registration commands or chat messages) on their dedicated port, they respond to registration messages to the source port of the client (that should remain the same throughout the session), and distribute the chat messages to the registered clients to that same port.
  - **`129.132.75.194:4999`** provides the test chat server. It provides all services that are available on the deployment server. Use this in the testing phase, especially when learning to implement the Lamport time and the vector clocks.
  - **`129.132.75.194:5000`** provides the deployment chat server. Use this when your implementation is sound in order not to crash the other groups' clients.

1 Getting Familiar with Datagrams (2 Points)

To familiarize yourself with the sending and receiving of UDP messages, create an Android application that provides a capitalizing service to its user by relying on the server at `129.132.75.194:4000`. Please make sure that you're on the ETH subnet (`129.132.0.0/16`) to communicate with the server. For configuring the ETH VPN on Android, please follow the instructions on `https://www1.ethz.ch/id/servicedesk/guide/vpn/index_EN`, this should yield an IP address in 129.132.0.0/16[2]. You're provided with a code skeleton in the *vs-nethz-capitalize.zip*. Pay attention to all files (classes and XML files) that are in the archive: They contain useful hints and implementation guidelines. Be careful and use at least SDK 2.3 (API 9). You can reuse parts of it in Task 2.

1. Use the provided archive and modify it so that you have a project called `vs-`*`nethz`*`-capitalize` with the package name `ch.ethz.inf.vs.android.`*`nethz`*`.capitalize`[3][4].
2. Pay attention to all the files that are provided in the archive. Use the classes that are provided and the UI.
3. You should enable the user to enter and submit a text message and also display the response by our corresponding service in a way similar to popular messaging services.
4. The UI needs to display incoming messages and let the user send chat messages. For displaying chat messages, you should use the classes we provide: `DisplayMessage` for the object to be displayed and `DisplayMessageAdapter` that extends the Android `ArrayAdapter` that provides an implementation of an `Adapter` and uses an array of arbitrary objects to manage a `ListView`. The way the message should be displayed is handled by the XML file also included in the archive.
5. Before displaying the messages in the UI, you should use the `Logger` class that is included and log the message that you are displaying.
6. Use UDP sockets `DatagramSocket(int port)` for communication with the server. Giving `0` as source port will assign a random *ephemeral port* on the client side. Fill in `UDPCommunicator` that you will be able to re-use in Task 2. All of the communication should happen in the class `UDPCommunicator`.

[2] If you're using the WiFi at ETH (*eth* or *eth-5*, you would most likely get an IP address from Switch (195.176.111.1/24)
[3] Reminder: When we refer to `nethz`, we refer to the group leader's ETH account identifier
[4] Reminder: When we refer to `nethz`, we refer to the group leader's ETH account identifier

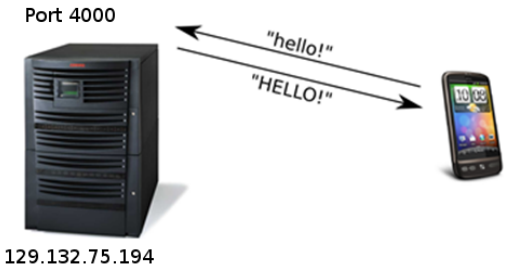

Figure 1: The setup of the capitalization service

2 Overcoming The Desequencer (6 Points)

In this task, you will create a chat application that communicates with the server at `129.132.75.194:5000`. It will use Lamport timestamps and vector clocks to mitigate the fact that the server is delaying the delivery of messages or that they might get lost and to restore the sequence of the messages you received. Be careful and use at least SDK 2.3 (API 9). We expect you to build on top of Task 1.

Please make sure that you're on the ETH subnet (`129.132.0.0/16`) to communicate with the server. For configuring the ETH VPN on Android, please follow the instructions on `https://www1.ethz.ch/id/servicedesk/guide/vpn/index_EN`, this should yield an IP address in 129.132.0.0/16[5].

You're provided with a code skeleton in the *vs-nethz-chat.zip*. Again, the files that are included in the archive contain useful hints and guidelines.

Your application should provide these essentials features: it should make sure that a network connection is available, then enable the user to choose a user name, register with the server and start chatting with other clients. It has to follow the guidelines defined in the protocol definition in Section 3 and in the accompanying slides that are provided at `http://vs.inf.ethz.ch/edu/VS/exercises/A3/HS2014_Android.Assignment3.Sheet.pdf`.

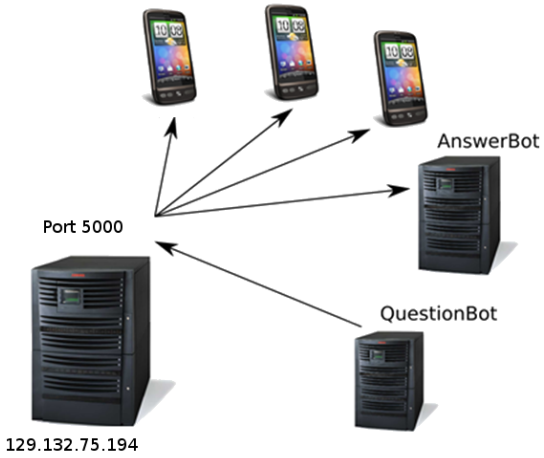

Figure 2: Overview of the chat system

[5]If you're using the WiFi at ETH (*eth* or *eth-5*, you would most likely get an IP address from Switch (195.176.111.1/24)

2.1 Basic Features (2 Points)

- Use the provided archive and modify it so that you have a project `vs-nethz-chat` with the package name `ch.ethz.inf.vs.android.nethz.chat`[6].
- Configure the UI to provide the user with the ability to enter a username and a button to register and deregister with the server. Note that we only accept usernames in the following format `nethz[0-9]*` where `nethz` is the ETH account identifier of the leader of the group (e.g., `caoh`). Please observe that the length of the username should be between 3 and 14 characters and that you have to be on the ETH subnet.
- Make sure that you're device is connected to the Internet before any interaction with the server.
- You should implement the option of selecting *Lamport timestamps* or *Vector Clocks* for the evaluation (check the XML files defining the GUI).
- After the registration, the UI needs to display incoming messages and let the user send chat messages. For displaying chat messages, you should reuse your implementation in Task 1.
- Before displaying the messages in the UI, you should use the `Logger` class that is included and log the message that you are displaying.
- Add communication functionality to allow sending messages to the server, a `DatagramSocket` sending to port `5000`. The registration and deregistration buttons should use this method for registration commands. Also make use of the `setSoTimeout(int timeout)` function. Again, the network communication should be implemented in `UDPCommunicator`. The server will return the assigned username, the assigned vector index and the current Lamport timestamp and time vector.
- Familiarize yourself with Java threads. The server will reply to the source port chosen by the client and track it based on its IP address/port choice, so implement a thread that listens for incoming messages in the background. For cross-thread information exchange, you will most probably want to use the `Handler` class. Remember to keep your `onCreate()` method as clean as possible and to use threads for delegating potentially long-running tasks. There are hints in the code provided.
- Test your listener thread. If you are using the emulator, remember to set port redirects. Depending on how fast you have progressed with the assignment, there will already be lots of communication going on. At the least `QuestionBot` and `AnswerBot` will be bringing life to the chatroom.
- Your application should also be able to handle that clients are dynamically joining and leaving the chat.

2.2 Incorporating Time Synchronization Measures

The methods we expect you develop are essentially the same for *Lamport timestamps* and *Vector Clocks*. You should enhance your listener thread to delay the display of incoming of messages if their timestamps show a gap. This means having the `isDeliverable(...)` method to explicitly inspect the timestamp of every incoming message and decides whether or not to delay its delivery. Delaying consists, in buffering the messages and deciding after enough time has elapsed that the message should be pushed to be displayed. Do not block on a single message, since you need to receive and process further messages for the re-ordering. Be careful with local concurrency, though!

[6]Reminder: When we refer to `nethz`, we refer to the group leader's ETH account identifier

To display the messages, pay attention to `DisplayMessage` and `DisplayMessageAdapter`. The latter should format the message to be displayed in the appropriate format (notice that the clients' usernames should be used). As soon as the messages are ready (successfully sent or successfully classified as deliverable, they should be logged using the `Logger` class included in the `ChatLogic` class. The log file will be used for the correction.

2.2.1 Lamport Timestamps (2 Points)

You obtain points for implementing the sorting based on Lamport Timestamps. Although, you're not using vector clocks to determine when the messages can be delivered to the UI, you should still transmit the correct vector clocks.

2.2.2 Vector Clocks (2 Points)

Here, you use Vector Clocks for determining the order of messages in the distributed system. Thus, you now have to parse and make use of the index number that the server assigns to your application during registration. Again, although you're not using Lamport timestamps for determining if the messages are deliverable, you should transmit it and keep its value up to date.

3 Mini-Test (2 Points)

As part of the assignment, you should answer these questions as briefly as possible. They relate 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 of the format **answers-`nethz`.pdf**[7]. Please make all group members' full names appear on the document.

1. What are the main advantages of using Vector Clocks over Lamport timestamps?
2. Give the conditions for two Vector Clocks to be causally dependent?
3. Draw the state machine for a successful registration. For this, identify the commands on top of arrows and link the states to your own naming in your implementation. (*Hint: What if there is no network connection?*)
4. Draw the state machine for sending a message.
5. We decided in the exercise that we would not let our applications trigger a tick when receiving a message. What would be the implications of ticking on receive?
6. Does a clock tick happen before or after the sending of a message. What are the implications of changing this?
7. Read the paper *Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications* by Tobias Landes [8] that gives a good overview on the discussed methods. In particular, which problem of vector clocks is solved in the paper?

[7] Again, the group leader's `nethz`
[8] http://vs.inf.ethz.ch/edu/vs/exercises/DVC_Landes.pdf

Deliverables

The following two deliverables have to be submitted by **09:00 A.M. (CET), November 3, 2014**:

- **code.zip** You should create a zip file containing the Eclipse projects created in this assignment. For this, right click on your project Export... > General > Archive File. We expect this exact structure since a code will extract and run your code:
  - /code/vs-`nethz`-capitalize/
  - /code/vs-`nethz`-chat/

  The projects should have been tested both on the mobile phone (compile for Android 4.1.2) 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 3 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

While questions are welcome, we request that you proceed to do you some research before contacting the TA (Google is a great source, reading this document, the slides and the comments in the code should also give you a lot of hints).
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`: hong-an.cao@inf.ethz.ch
- `SUBJECT`: [VS_2014] `group leader's nethz` - Topic
- `BODY`: Proper greetings and any relevant content.

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).
We expect that an unsuccessful attempt at registering with the server has to be handled and the user should be given the means to remedy this.
The logging has to be strictly compliant with the given instructions.
The server is monitored. This means that any attempt to launch an attack will be sanctioned by a fail for the group responsible for it. Then, the department will be made aware of such behavior.

Protocol specifications

The following commands *cmd* are accepted by the chat server. Please read the specifications carefully and pay attention to the **regular expressions**.

- **register** : the client must first send this JSON message to the server before any other interaction can be initiated and accepted by the server as in Listing 1. Notice that the username has to be in alphanumerical format in small caps ([a-z], [0-9]) and respect the following conventions :
  - `nethz`[0-9]*, where the `nethz` represents the `nethz` account of the team's leader, for example `caoh`
  - the length of the username should be between 3 and 14 characters

```
{
    "cmd" : "register",
    "user" : "caoh"
}
```

Listing 1: Client's *register* request in JSON

If the registration is successful, the server will return an index that identifies the client along with the initial time vectors and Lamport time as in Listing 2.

```
{
    "cmd": "register",
    "status": "success",
    "index" : "3",
    "init_time_vector" : {"2": 0, "1": 70, "0": 71},
    "init_lamport" : 74
}
```

Listing 2: Server's *register* reply in JSON: success

All other clients will receive the following notification: This means that a message will be broadcasted to all other participants and will notify them that the chat participant left (Listing 17).

```
{
    "cmd": "notification",
    "text": "caoh has joined (index 2)"
}
```

Listing 3: Broadcasted notification after successful *register* in JSON

In the case where the registration fails, for example when the username is already in use or not agreeing with the above mentioned conventions, a failure message will be returned to the client as in Listing 4.

```
{
    "cmd": "register",
    "status": "failure",
    "text": "Not registered: username already in use,
        nethz not recognized, invalid length
        (max 3-14 characters) or not on the ETH subnet."
}
```

Listing 4: Server's register reply in JSON: failure

If any other command is initiated without being registered, the reply shown in Listing 5 is returned by the server. This behavior can also happen if the server restarted. In this case, the client should proceed with registering again.

```
{
    "cmd": "register",
    "status": "failure",
    "text": "Not registered"
}
```

Listing 5: Server's *register* reply in JSON: not registered

If the client is already registered, an error message such as in Listing 6 will be returned

```
{
    "cmd": "register",
    "status": "failure",
    "text": "Already registered"
}
```

Listing 6: Server's *register* reply in JSON: already registered

- **get_clients** The list of chat participants can be retrieved by sending the request in Listing 7.

  The server then sends out a a mapping of the indices that identifies the clients and their usernames as in Listing 8. This needs to be stored by the client for further usage.

- **info** Information about the server can be requested using the code in Listing 9.

  The reply in Listing 10 is then send out to the client.

- **message** To broadcast message to all other chart participants the JSON request in Listing 11 should be sent.

  If the message could be delivered the client will receive the reply as in Listing 12.

  All other clients will receive the message as in Listing 13.

- **deregister** The client can decide to leave the chat by sending out the request as in Listing 15.

```
{
      "cmd": "get_clients"
}
```

Listing 7: Client's *get_clients* request in JSON

```
{
    "cmd": "get_clients",
    "clients": {
                        "0": "questionbot",
                        "1": "answerbot",
                        "2": "caoh"
                    }
}
```

Listing 8: Server's *get_clients* reply in JSON

The client receives the response displayed in Listing 16.

This means that a message will be broadcasted to all other participants and will notify them that the chat participant left (Listing 17).

The client receives the response displayed in Listing 18 in case of failure.

Important: if any other message is sent to server and doesn't comply to the above described commands, the server will the return the message shown in Listing 19. Also, the server is keeping track of inactive clients and will **deregister** them automatically after a timeout period of **5 minutes**.

```
{
     "cmd": "info"
}
```

Listing 9: Client's *info* request in JSON

```
{
    "cmd": "info",
    "text": "I am an advanced UDP server that is running
         at port 5000 to provide a desequencing service
         for Android UDP chatting programs..."
}
```

Listing 10: Server's *info* reply in JSON

```
{
     "cmd": "message",
     "text" : "hi there",
     "time_vector": {"2": 1, "1": 70, "0": 71},
     "lamport": 75
}
```

Listing 11: Client's *message* request in JSON

```
{
    "cmd": "message",
     "status": "success"
}
```

Listing 12: Server's *message* reply in JSON: success

```
{
    "cmd": "message",
    "text": "hi there",
    "sender": 2,
    "time_vector": {"2": 1, "1": 70, "0": 71},
    "lamport": 75
}
```

Listing 13: Broadcasted *message* in JSON

```
{
    "cmd": "message",
     "status": "failure"
}
```

Listing 14: Server's *message* reply in JSON: success

```
{
     "cmd": "deregister"
}
```

Listing 15: Client's *deregister* request in JSON

```
{
    "cmd": "deregister",
    "status": "success"
}
```

Listing 16: Server's *deregister* reply in JSON

```
{
    "cmd": "notification",
    "text": "caoh has left (index 2)"
}
```

Listing 17: Broadcasted notification after successful *deregister* in JSON

```
{
    "cmd": "deregister",
    "status": "failure"
}
```

Listing 18: Server's *deregister* reply in JSON in case of failure

```
{
    "cmd": "unknown",
    "status": "failure",
    "text": "Invalid JSON string. Make sure you use the
         right structure for this message."
}
```

Listing 19: Irregular JSON String