Assignment 3

Start:	28 October 2013
End:	11 November 2013

Objectives

In this assignment, you will develop an n-person mobile chat application that preserves the causality and temporal ordering of messages in spite of the unreliability of the underlying UDP protocol. You will implement two concepts to determine the order of events in a distributed system that you have learnt in the lecture.

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 two servers running:

- **`vslab.inf.ethz.ch:4000`** provides a capitalization service: it returns every incoming message changed to upper case. This server may be used to test UDP-based communication. It replies to the source port chosen by the client.
- **`vslab.inf.ethz.ch:5000`** provides the chat service, which distributes all messages received from a registered client to all other registered clients. The server randomly delays messages to simulate communication unreliability inherent in real systems. While this server is listening for incoming datagrams (registration commands or chat messages) on its port `Server:5000`, it responds to registration messages to the source port of the client, and it is distributing the chat messages to the registered clients to that same port.

With this assignment you can gain 10 points out of the total of 45. The exercises marked with a ⊙ are necessary to meet the minimum requirements ("save-point").

1 Getting Familiar with Datagrams

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 `vslab.inf.ethz.ch:4000`. This application has only demonstration purposes and does not need to be submitted, however, you can reuse parts of it later.

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

1. Create a new application in a project called `vs-nethz-capitalize` with the package name `ch.ethz.inf.vs.android.nethz.capitalize`[2].
2. Set up the UI to enable the user to enter and submit a text message and also display the response by our corresponding service.
3. 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.

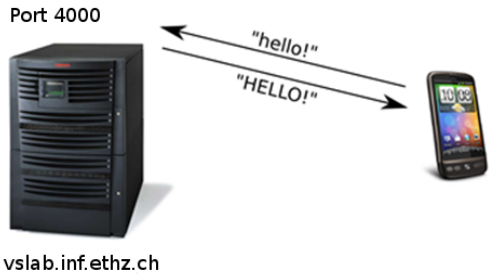

Figure 1: The setup of the capitalization service

2 Starting the Conversation: Lamport Timestamps (4 Points, ☉)

In this task, you will create a chat application that leverages the communication server at `vslab.inf.ethz.ch:5000`. The application will use Lamport Timestamps to delay the delivery of messages to the user to keep them in order. Create a program that enables the user to choose a user name, register with the server and start chatting with other clients using the guidelines defined in the accompanying slides that are provided at `http://vs.inf.ethz.ch/edu/vs/android/`.

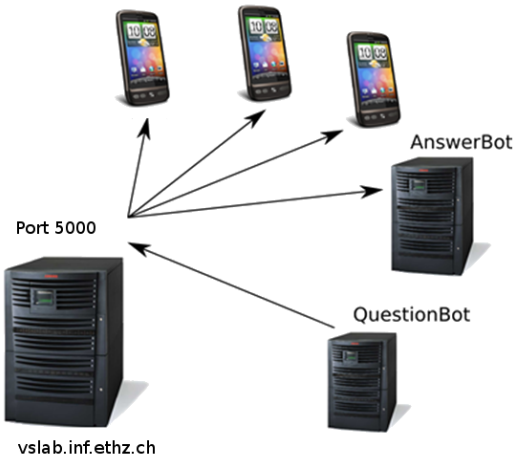

Figure 2: Overview of the chat system

[2] `nethz` should be the group's leader `nethz` login

1. Create a new project `vs-nethz-lamport` with the package name `ch.ethz.inf.vs.android.nethz.lamport`.

2. 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 one of the leader of the group (e.g., `caoh1`). Please observe that the length of the username should be between 3 and 14 characters.

3. After the registration, the UI needs to display incoming messages and let the user send chat messages. For displaying chat messages, you can, for instance, use the Android `ArrayAdapter` that provides an implementation of an `Adapter` and uses an array of arbitrary objects to manage a `ListView`.

4. 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. The server will return the assigned username, the assigned vector index and the current Lamport timestamp and time vector. Ignore the information on the assigned index and vector, which will be needed for Task 3.

5. 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.

6. 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.

7. Enhance your listener thread: use Lamport Timestamps (see Section 4 for more details about the protocol specifications) to delay the displaying of incoming messages if their timestamp shows a gap. To do this, create a method `isDeliverable(...)` that explicitly inspects the timestamp of every incoming message and decides whether or not to delay its delivery. As soon as this function returns `true` for a message, it shall be delivered to the user. We are providing you with a desired template for the `isDeliverable(...)` method to test your code based on a dump of messages to be sorted by your method.

8. 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!

3 Overcoming the Desequencer: Vector Clocks (4 Points)

For this task, create a new application that implements the same functionality as the one developed in Task 2, but uses Vector Clocks for determining the order of messages in the distributed system. Your application should also be able to handle that clients are dynamically joining and leaving the chat.

1. Create a new application in `vs-nethz-vector` using the package name `ch.ethz.inf.vs.android.nethz.vector`. Set up the UI and functionality as before.

2. Instead of Lamport Timestamps, use Vector Clocks to determine the order of messages and display them to the user accordingly. Thus, you now have to parse and make use of the index number that the server assigns to your application during registration.

3. Create the method `isDeliverable(...)` that explicitly inspects the vectorial timestamps of incoming messages and decides whether or not to delay the delivery of a message. As soon as this function returns `true` for a message, the message shall be delivered to the user.

4 Report (2 Points, ⊙)

As part of the assignments, you should produce a short report **(1-2 pages)** on the design and implementation issues of Tasks 2 and 3 and motivate any choice you made during the process. You can find a template for your report on the course Web site. Include a thorough discussion of issues and considerations related to Vector Clocks in your report. Specifically, answer these questions:

- What are the main advantages of using Vector Clocks in comparison with Lamport Time?
- When *exactly* are two Vector Clocks causally dependent?
- 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?
- Does a clock tick happen before or after the sending of a message. What are the implications of changing this?
- Read and assess the paper Tobias Landes - Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications[3] that gives a good overview on the discussed methods. In particular, which problem of vector clocks is solved in the paper?

Deliverables

The following two deliverables have to be submitted by **09:00 A.M. (CET), November 11, 2013**:

- **code.zip** You should create a zip file containing the Eclipse 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.
- **report.pdf** The report 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 that.

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

Protocol specifications

The following commands *cmd* are accepted by the chat server. Please read the specifications carefully.

- **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 `caoh1`
  - the length of the username should be between 3 and 14 characters

```
{
      "cmd" : "register",
      "user" : "caoh1"
}
```

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.

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

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

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 3.

```
{
    "error": "reg_fail",
    "text": "Already registered or username already in use,
        nethz not recognized or invalid length
        (max 3-14 characters)."
}
```

Listing 3: Server's register reply in JSON: failure

If any other command is initiated without being registered, the reply shown in Listing 4 is returned by the server. This behavior can also happen if the server restarted. In this case, the client should proceed with registering again.

```
{
    "error": "not_registered"
}
```

Listing 4: Server's *register* reply in JSON: not registered

If the client is already registered, an error message such as in Listing 5 will be returned

```
{
    "error": "already_registered"
}
```

Listing 5: Server's *register* reply in JSON: already registered

- **get_clients** The list of chat participants can be retrieved by sending the request in Listing 6.

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

Listing 6: Client's *get_clients* request in JSON

The server then sends out a a mapping of the indices that identifies the clients and their usernames as in Listing 7. This needs to be stored by the client for further usage.

- **info** Information about the server can be requested using the code in Listing 8.

  The reply in Listing 9 is then send out to the client.

- **message** To broadcast message to all other chart participants the JSON request in Listing 10 should be sent.

  If the message could be delivered the client will receive the reply as in Listing 11.

  All other clients will receive the message as in Listing 12.

- **deregister** The client can decide to leave the chat by sending out the request as in Listing 13.

  The client receives the response displayed in Listing 14.

  This means that a message will be broadcasted to all other participants and will notify them that the chat participant left (Listing 15).

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 16. Also, the server is keeping track of inactive clients and will **deregister** them automatically after a timeout period of **5 minutes**.

```
{
      "clients": {
                        "0": "questionbot",
                        "1": "answerbot",
                        "2": "caoh1"
                  }
}
```

Listing 7: Server's *get_clients* reply in JSON

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

Listing 8: Client's *info* request in JSON

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

Listing 9: Server's *info* reply in JSON

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

Listing 10: Client's *message* request in JSON

```
{
      "success": "msg_ok"
}
```

Listing 11: 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 12: Broadcasted *message* in JSON

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

Listing 13: Client's *deregister* request in JSON

```
{
    "success": "dreg_ok"
}
```

Listing 14: Server's *deregister* reply in JSON

```
{
    "cmd": "message",
    "text": "caoh1 has left (index 2)"
}
```

Listing 15: Broadcasted *deregister* message in JSON

```
{
    "error": "Invalid JSON string. Make sure you use the
        right structure for this message."
}
```

Listing 16: Irregular JSON String