Distributed Systems – Assignment 3

Robin Guldener
ETH ID 11-930-369
robing@student.ethz.ch

Nico Previtali
ETH ID 11-926-433
pnico@student.ethz.ch

Lukas Bischofbergeryx-cvyxcvxyv
ETH ID 11-915-907
lukasbi@student.ethz.ch

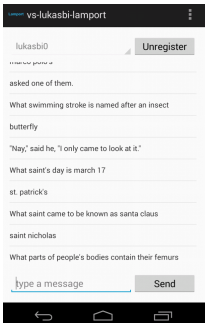

Figure 1: Lamport Chat overview

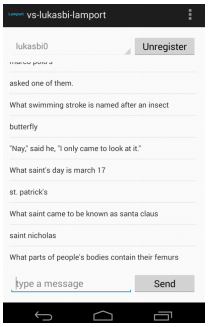

Figure 2: State transition diagram

ABSTRACT

For the implementation of this assignment we made use of the two lab provided HTC desire number 25 and 37 and a private Samsung Galaxy Tab 2. We were able to implement both assignments and to our best knowledge both chats are working flawlessly.

1. INTRODUCTION

- Lamport timestamps[3] and vector clocks[4] are used to order the messages we receive from the chat server. Both timestamps are stored for all messages implicitly since we always store the entire JSONObject returned from the server. This has the advantage that large parts of our chat implementation is shared between the lamport and the vector clock version, which eases maintenance and sped up our development process. For Lamport timestamps we explicitly ordered the messages as they were arriving whereas for vector clocks we only order the messages implicitly in the displaying ListView. Further details regarding these implementations can be found in the respective sections below.
- We used asynchronous tasks to register, unregister, get clients/server information from the server and sending chat messages. So these tasks can send and possibly also receive packets. For receiving information from the server, e.g chat messages and logout information about other clients we have implemented a background thread which has only the purpose to listen for incoming UDP packets.

2. LAMPORT TIMESTAMPS

- The UI thread has a field called `lamportTime` which stores the current Timestamps for this process. When recieving a new message the Lamport Timestamp is updated according to the algorithm presented in the lecture. When sending a new message the current Lamport Timestamp of the process is attached to the JSON object and then sent to the chat server.
- For all commands that are being sent to the chat server we provide a AsyncTask which then communicates in the background with the chat server. This AsyncTask [1] is reserved for all outgoing UDP packets. On the other side we implemented functionality to only recieve UDP packets. This class (providing the mentioned functionality) is running in a seperate thread and its main goal is to catch all incoming chat messages and the server status. The recieved messages are then sent to the UI thread for displaying them. This is done with the use of an Handler [2]. For better understanding see the state transition diagram (cf. Figure 2).
- The main problem was that we first tried to connected to the same UDP Port which is obviously not possible.

3. VECTOR CLOCKS

- To achieve efficiency we reused the whole code of Task 2. The main differences are that we now implemented Vector clocks and discarded Lamport Timestamps. For this we had do change the handler to parse the recieved JSON to extract and build the vector clock. Another main difference is that we introduced a new class called `VectorClockComparator` which is able to compare two vector clock timestamps.

```
1  public class VectorClockComparator implements
       Comparator<JSONObject> {
2    @Override
3    /**
4       * o1 is before o2 iff both conditions are
            met:
5       * - each process's clock is less-than-or-
            equal-to its own clock in other; and
6       * - there is at least one process's clock
            which is strictly less-than its
7       *   own clock in o2
8       */
9      public int compare(JSONObject o1,
           JSONObject o2) {
10     int isBefore = 0;
11
12     // Since we are not guaranteed that both
           objects have the same indexes compute
           intersection and compare based on that
```

```
13      HashMap<String, Integer> vector1 =
            MainActivity.vectorClockFromJSON(o1);
14      HashMap<String, Integer> vector2 =
            MainActivity.vectorClockFromJSON(o2);
15      HashMap<String, Integer> intersection = new
             HashMap<String, Integer>(vector1);
16      intersection.keySet().retainAll(vector2.
            keySet());
17
18      for (String key : intersection.keySet()) {
19        int val1 = vector1.get(key);
20        int val2 = vector2.get(key);
21
22        if (val1 > val2)
23          return 1;
24        else if (val1 < val2)
25          isBefore = -1;
26
27      }
28
29      return isBefore;
30    }
31  }
```

Listing 1: Using AsyncTask

- Since we cannot reliably compare two vector clocks with this chat implementation (this is due to the fact that the number of objects in the clock and the devices they represent may vary from message to message) we were unsure how to reliably order vector clock messages and guarantee direct consecutive ordering. This also represented our main challenge in this part of the application and since we could not come up with a satisfactory definition for consecutive ordering we finally agreed in ignoring the isDeliverable method and to provide at best-effor total ordering of the incoming messages. This is also reflected in our comparator listed above, which makes use of the intersection to order any two messages but this is not ideal from a theoretical standpoint and may not always yield good absolute orders. Further information regarding vector clocks from the lecture would have been beneficial.

4. DISCUSSION

Please reply to the following questions.

- **What are the main advantages of using Vector Clocks over Lamport Timestamps?** The system of vector clocks is strongly consistent. Thus, by comparing the vector timestamp of two events, we can determine if the events are causally related. This is called the *strong clock consistency condition*. Using only a Lamport Timestamps, only a partial causal ordering can be inferred from the clock (the implication goes only in one way, called the *clock consistency condition*).
- **When exactly are two Vector Clocks causally dependent?** The conditions are already listed in the code snippet but are repeated here. For two vector clocks to be causally dependent they need to be either each clock in the first is less than or equal to the clocks in the second OR there is at least one clock in process one that is strictly less than it's own clock in the second.
- **Does a clock tick happen before or after the sending of a message. What are the implications of changing this?** We decided that we would not let our applications trigger a tick when receiving a message. The implications of also ticking the clock on a message receive would be that consecutive ordering could no longer be guaranteed, since it is not known how many messages any client received before sending the next one.
- **Read and assess the paper Tobias Landes - Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications3 that gives a good overview on the discussed methods. In particular, which problem of vector clocks is solved in the paper?** The paper investigates the implications of dynamic arriving and leaving of processes on the vector clock logical time ordering system. Whilst the original vector clocks are not sufficient for such situation, the paper presents a workable solution for such dynamic environements that builds on the original vector clocks idea and extends the known concept.

5. CONCLUSION

One main challenge we encountered was to make two threads listening to the same UDP port. We first tried to handle this by sending and retrieving all commands and messages through the same thread. But then we decided to handle this challenge by binding one of the two threads to the current used port rather than just using one thread. We did it this way because then we can seperate between just listening to a UDP port for incoming messages (the chat and status messages) and sending commands to the chat server (such as deregister or sending a message). We think this is a much better design style especially when someone tries to read and understand our code.

To achieve good efficiency we divided the work into three seperate parts. Each team member worked on one part. Because the parts may overlapped we worked sometimes in parallel and sometimes not. The first part consisted of the following tasks

- Setup up the whole UI and provide each UI element with the related functionality.
- Setup up an AsyncTask for register / deregister commands as well as for retrieving the server status and a client list.
- Design the basic layout and functionality of the UDP listener thread. The UDP listener is design to be able to communicate with the UI thread because message have to be passed from the UDP listener thread to the UI.

The second part consisted of

- 1
- 2
- 3

Last we had to

- Implement the `ChatAdapter` for dynamicly populating the ListView
- Implement the isDeliverable method for lamport timestamps and cache all incoming requests until all consecutive messages arrived
- Implement the vector clocks comparator and figure out how to order two messages with different vector clock entries

6. REFERENCES

[1] Android AsyncTask. `http://vslab.inf.ethz.ch:8081/sunspots/Spot1/sensors/temperature`. Accessed on 10 Nov 2013.

[2] Handler object in Android. http://vslab.inf.ethz.ch:8081/sunspots/Spot1/sensors/temperature. Accessed on 10 Nov 2013.

[3] Lamport timestamps. http://en.wikipedia.org/wiki/Lamport_timestamps. Accessed on 10 Nov 2013.

[4] Vector clocks. http://en.wikipedia.org/wiki/Vector_clock. Accessed on 10 Nov 2013.