Distributed Systems – Assignment 3

Frederik Rothenberger
frothenb
frothenb@student.ethz.ch

Simon Wehrli
siwehrli
siwehrli@student.ethz.ch

ABSTRACT

This report describes the two Android apps programmed by Simon Wehrli and Frederik Rothenberger. Both apps provide a chat environment, which allows the user to register on the chat server and broadcast messages and receive broadcasted messages sent by other registered clients. The only difference between the apps is the way the timing information delivered with each message is used to give the most consistent message order possible to the client. The first app uses the Lamport Time and the second the vector time to achieve this goal.

INTRODUCTION

Both apps are divided into the following components. A graphical user interface let the user register for the chat room, send messages and displays chat and status messages. Furthermore, there are several asynchronous background tasks and threads implementing the actions chosen by the user while keeping the GUI responsive.

We first present the components common to both apps and then highlight important design decisions specifically to the Lamport resp. the vector time. Finally we discuss the problems encountered during the development process.

GENERAL COMPONENTS

Since both apps provide the same usability to the user and only differ in the implementation of the time logic, the structure of the apps is very similar. The apps can be divided into the following components: The *AsyncTask* for registering and deregistering the user, the *AsyncTask* for sending messages, the Thread for receiving messages and the Thread(s) which wait in order to give delayed messages a chance to arrive.

The *AsyncTask* for (de)registering is called *RegisterTask* and informs the chat server of the user joining or leaving depending on how this task was created. This task is executed whenever the user presses the register toggle button. Additionally we also deregister the user if, upon leaving the app via the back button, he is still logged in, so the server can free the resources taken by handling our user. If the user successfully registers him, the time vector or Lamport Time is set according to the "reg_ok" response of the server so that sending and receiving messages is in sync with the server from the beginning.

Sending messages is also done in an *AsyncTask* (the *SendTask*) so that the user can receive other messages or already type the next message while the other message is being sent. Of course, in advance to actually sending the message, the time vector and Lamport Time are increased by one. If, for any reason, the send is not successful a toast is displayed informing the user of this fact.

Receiving messages is, as opposed to the previously described components, not implemented using *AsyncTask* but uses the normal Thread class instead. This is because in Android *AsyncTasks* are not really executed asynchronously in parallel but rather sequentially without a guarantee of any order. Since we have to keep listening until the user closes the app or choses to deregister and implementation using *AsyncTasks* would block the other *AsyncTasks* like sending or (de)registering, which would be really bad. To allow this thread to exit gracefully, if the app is closed, we chose to only wait for five seconds before the blocking receive on the UDP Socket times out and allowing the receiving thread to check the current status of the app and either exit or continue listening.

Finally we needed some way of knowing when a message which arrived with a too high timestamp (either due to message loss or reordering) would be shown to the user anyway expecting that the messages with the lower expected timestamps were lost. Here we also used threads because it allowed for an easy and fast implementation. These threads do nothing more than sleep for the given timeout amount and then poll the message with the lowest time out of a time sorted *PriorityQueue* and displaying it to the user and updating the current time if still necessary.

The only differences between the two apps are actually a *boolean* flag which switches the way time is handled at a few key points where this happens.

TIME LOGIC

VS_siwehrli_A3_2 (Task 2, App 1)

Lamport Time

We learned that the only conclusion we can made out of Lamport timestamps is:

$$m1 \rightarrow m2 \Rightarrow T(m1) < T(m2)$$

(where $\rightarrow$ is casually dependence and $T(m)$denotes the timestamp of message m)

Since we have no measurement of causality in our application, only the inverse of the equation is useful:

$$T(m1) \geq T(m2) \Rightarrow \neg(m1 \rightarrow m2)$$

This gives us the insight, that if we have locally the information, that $m2$is not causally dependent on $m1$, we should look that the timestamp of $m1$is really greater than $m2$.

Message Receive
Messages are received in the receiver thread and then processed in parallel. The time logic (discussed below) then decides on the status of the received message, which determines if the message is delivered and displayed to the user or temporarily stored in a waiting queue (a synchronized blocking priority queue), trying to deliver it later when it might be possible to give a consistent view of the order of the messages. The possible statuses are:

- ***On time, in sequence***: The message is directly following the last delivered message (even though we do not know they are causally dependent). We deliver the message to the message list which makes it visible to the user. We know may deliver waiting messages from the queue too.
- ***On time, out of order***: The message has a timestamp that let us assume that some messages are missing. We do not deliver the message yet and add it to the waiting queue, start timeout, hoping the missing messages are arriving within it. If the timeout expires, we deliver the messages anyway, with the risk that the missing messages get *delayed*.
- ***Delayed***: The message was assumed to be lost and now arrives delayed. Anyway it's is inserted where the timestamp proposes, but it get marked as delayed (yellow colour) to notify the user of the inconsistence.

When deliver a message, we update the local time to the maximum of the local time and the stamp on the message. For both logics, we do not increase the local time yet, because then we would not be able to detect missing messages anywhere, since the timestamp differences seen by the chat users would be greater than one and especially dependent on the number of current chat users!

Note that this significantly varies from the version found in general literature. We only update the local time when we deliver a message to the user and not when receiving it in the first place. To understand the reason for this design decision, consider the following example (FIGURE):

Chat participant A sends (broadcasts) three messages $m1$, $m2$, $m3$, where $m2$ get lost somewhere and doesn't arrive at participant B. This makes B pushing $m3$ to his waiting queue and starting a timeout. If now the user B tries to send a new message $m4$ (point t in FIGURE), we could imagine different behaviours:

1. Do not let the user B send any messages at all until the timeout expires and the waiting message is delivered.
2. As soon as the user B starts typing, deliver the waiting message $m3$ anyway without respect to the timeout.
3. Let the user sending messages independently of waiting messages.

We chose the third behaviour because we don't like prevent the user from sending messages and the second behaviour seems not user friendly to us.

To improve the causality of the local order of the messages, we decided to update the local time not until a message is delivered. Hence on t, we still put timestamp 2 to the new message $m4$. Assume we would update the local time as soon as receiving a message. Then $m4$ would already have timestamp 4, which induce other participants, which haven't received $m2$ or $m3$, pushing $m4$ to their waiting queues, even though B knows that $m4$ neither causally depends on $m2$ nor on $m3$, because the user had not seen $m2$ and $m3$ when he has written $m4$.

This change to the original proposed protocol does not solve all problems of inconsistencies, but let increase the local Lamport Time as late as possible, which reduces the suggested (but maybe wrong) causal dependencies between messages.

Message Send
Before sending a message we always increase the local time and give the increased time along with the message, because we know locally that new message can depend on already displayed messages.

VS_siwehrli_A3_3 (Task 3, App 2)

Vector Time
By using vector time we can now correctly detect concurrent events. This is actually very useful in a scenario like this because many users can, and probably will, type messages concurrently and vector time gives us a better understanding of how these messages have to be ordered. Or rather, which messages were actually typed concurrently and which were typed sequentially.

The actual logic and reasoning on whether a message is delayed, early or on time is the same as with Lamport Time *[see VS_siwehrli_A3_2 (Task 2, App 1), Message Receive]*. The only difference is in the implementation of

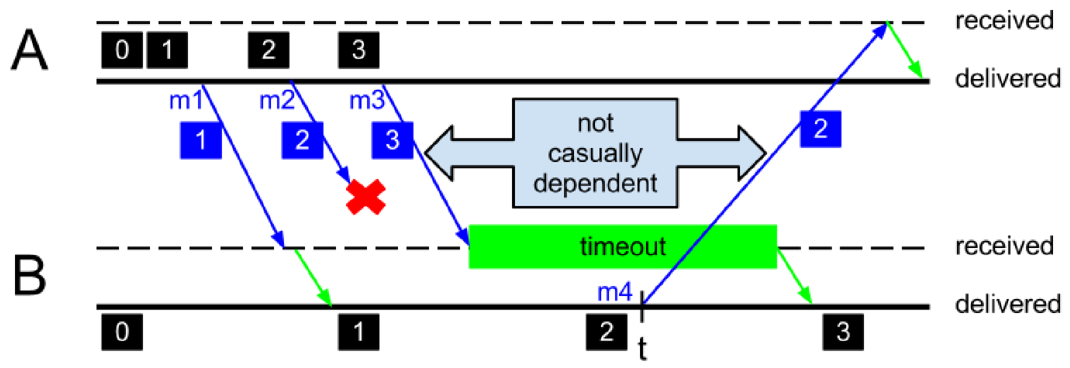

Figure 1: There are different behaviours possible if the user on B tries to send a message on t. The diagram shows a scenario where on t the new message gets the timestamp of the last delivered (not received) message.

the compare methods of message objects and that if two messages are compared as equal (regarding time), it is still necessary to check every component of the time vectors to update to a current local time vector of all the maxima of the different components.

REPORT QUESTIONS

What are the main advantages of using Vector Clocks instead of Lamport Time?

In addition to the weak clock condition (if an event $e1$ happened before event $e2$, $e1$ has a lower timestamp than $e2$) we now also have the strong clock condition: if $e1$ has a lower timestamp than $e2$ then $e2$ is causally dependant from $e1$ ($e1$ was (possibly) the cause of $e2$). Another advantage is that we now also are able to tell whether two events happened concurrent and are not causally dependent.

When exactly are two Vector Clocks causally dependent?

Two vector clocks are causally dependent if all timestamps of one clock are bigger than all the timestamps of the other one.

For example $\{A{:}\,2, B{:}\,3, C{:}\,4\}$ is causally dependent on $\{A{:}\,1, B{:}\,2, C{:}\,2\}$ but $\{A{:}\,2, B{:}\,7, C{:}\,4\}$ and $\{A{:}\,3, B{:}\,5, C{:}\,9\}$ are concurrent and not causally dependent since the orderings $B{:}\,7 > B{:}\,5$ and $A{:}\,2 < A{:}\,3$, $C{:}\,4 < C{:}\,9$ contradict each other.

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?

One implication of ticking on receive are unique timestamps. Even if several messages with exactly the same timestamp would be sent to a node, the receive event would differ in vector time for said node because of ticking on receive.

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

A clock tick happens before the sending of a message. If we change this the clock conditions are no longer valid:

Assume we have nodes A and B starting each with a time of 0. Now if A sends a message to B and ticks afterwards, B will have a time of $\{A{:}\,0, B{:}\,0\}$ while A now has $C{:}\,4 < C{:}\,9$. So B sends a response to A, also ticking after the send event, resulting in A receiving a message with timestamp $\{A{:}\,0, B{:}\,0\}$ which is bad for several reasons. First, we can not tell whether this was before or after the start (which also happens to have $\{A{:}\,0, B{:}\,0\}$ as a timestamp) and second, more importantly, the $\{A{:}\,1, B{:}\,0\}$ timestamp is now causally dependent on the received message which means that receiving the answer to a message happened before sending the original message. This is obviously nonsense.

Read and assess the paper Tobias Landes - Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications that gives a good overview on the discussed methods. In particular, which problem of vector clocks is solved in the paper?

This paper addresses the issue that distributed systems tend to change dynamically during run time and explores the implications thereof. In particular, the paper shows that having the clock vector grow dynamically is much easier than getting rid of obsolete and outdated entries.

GENERAL DIFFICULTIES

The chat application is not only distributed over the network, but also on the local host highly parallel which called for some synchronization between threads, and in contrast to the predefined client-server protocol, we had to come up with our own ideas here.

Most difficulties occurred during testing, since on a distributed and shared test setting it’s hard to reproduce a failure, especially if we have a server that may accidentally fail and other students spamming the server with their own tests.

Also, we once again struggled with certain peculiarities of the Android Framework. One point, as already mentioned earlier, was that *AsyncTasks* are executed sequentially (as in, by a thread pool having only one worker). Another difficulty arose from the different Android version we were developing on. Simon uses Android 2.3.3 and Frederik version 4.0.4. Apparently Google decided to change the way Socket timeouts are handled between these versions. In Android 2.3.3, if a *Socket.receive* hits the timeout a *SocketIOException* is raised (which has to be handled). Not so in Android 4.0.4: if this happens there, a *SocketTimeoutException* is raised, but you do not have to handle it (so you do not get told to either catch it or throw it to the caller), so our app just crashed. After we found out about this, we caught it in a catch block and printed the *exception.message* to the *LogCat*, only to have our app crash once again. As it turns out, the *SocketTimoutException* is actually *null* when entering the catch block and therefore it just raised a *NullPointerException* when we tried to read the exception message.

Additionally, we had some problems with non GUI threads “leaking views”. This happens when a thread, which is not the GUI thread, tries to manipulate views directly or if any threads try to access views after the app has already exited. Luckily, the Handler class is very easy to use and we did not lose much time chasing these bugs.

WHO DID WHAT

Simon built up the GUI and the general framework with lot of helper functionality for parsing JSON objects, sorting and classifying messages. Frederik then implemented the core time logic.

CONCLUSION

A non-reliable basis protocol (UDP in our case) might not be the choice for a chat app in real life. But it highlights some interesting problems of causality, which can’t be perfectly solved, but this makes it interesting to find a solution which gets as far as possible. And as an everyday user of chat applications there’s the final insight: it’s all but trivial!