Distributed Systems - Exercise
Summer term 2018

Tim Jungnickel
Andreas Kliem
FG CIT

Exercise sheet 2

In this second exercise sheet we explore how nodes in a distributed system can agree on a particular order of messages without relying on physical clocks. The challenges that are addressed with this exercise are based on fundamental problems with distributed systems and the here proposed solutions are extensively used in real-world applications. Hints and further details are provided in the corresponding lecture and tutorial.

Exercise 1: [Message Sequencer]
Develop an application with a *programming language of your choice* where multiple threads can broadcast messages over a message sequencer. Here, each thread maintains an own queue which can be seen as *inbox* for new messages. One thread continuously polls the inbox-queue for new messages and if a message is read, the next steps are evaluated based on the message type. Generally, we distinguish between the following message types:

- **External Messages:** External messages can be seen as messages from a client. These messages contain a random integer value as payload and are sent to the inbox-queue of *only one* thread. Once an external message is received, the message's content must be broadcasted to the other threads over the message sequencer.
- **Internal Messages:** Internal messages are sent from one thread to the inbox-queue of the message sequencer or from the message sequencer to the inbox-queue of one thread. Those messages can contain additional information (thread-id, counters etc.).

The message sequencer acts as a broadcasting service. It is the responsibility of the message sequencer to forward each received internal message to every thread in the system; including the sender-thread. Because there exists only one message sequencer in the entire system, all internal messages *should* be received in the same order at all threads. Note that the message sequencer never receives external messages.
The implementation has to comply with following technical requirements:

- The number of threads that can receive external messages must be customizable over a command-line argument and each thread stores a history (e.g. an array) of received internal messages. When the program terminates, all threads must be stopped and the threads write their history to a thread-specific log file on the file system.
- The message sequencer runs in an additional thread.
- Simulate clients by generating a customizable number of external messages which are sent to random threads (except the message sequencer).

Additional notes:

- This exercise can be entirely realized without using network/socket communication. The message transmission is simulated by putting/polling from the inbox queues.
- The groups should be able to demonstrate the above described application with a *reasonable* number of threads (>4) and generated external messages (>999).

Exercise 2: [Lamport Timestamps]
In the previous exercise the message sequencer can easily become a bottleneck and can reduce the overall throughput of the system. One solution to this problem is to use Lamport Timestamps, which provide total ordering without relying on physical clocks or centralized architectures, e.g. the message sequencer.

Redesign and reimplement the application from the previous example so that the threads can directly exchange messages. Here, one thread that reads an external message from the inbox now distributes the content as internal message to the inbox of every other thread. The treads must agree on a total order of the messages by utilizing Lamport Timestamps. To this end, every thread maintains a sequence of messages that reflects the order according to the Lamport Timestamps. When the program is terminated, all threads write their sequence to a thread-specific log file. If everything works, all threads report the same sequence of messages.

Additional notes:
- Since multiple threads are now accessing the inbox queue of other threads, the access must be synchronized to avoid race conditions.
- Each thread must store additional information (e.g. maximum counter) and the threads must be initialized with ascending id's.

Exercise 3: [Vector Clocks]
For this exercise there is no further implementation necessary. The task is to discuss what needs to be changed in the previous implementation when using vector clocks instead of Lamport Timestamps. In particular, the following questions must be respected:
- What kind of order would be implied by the vector clocks (total order/partial order)?
- Would the history of all threads be identical? If not, how would you test that the exposed histories of the threads are *correct*.
- What are the advantages/disadvantages of using Lamport Timestamps or vector clocks?

Note that every group member must be able to answer these questions. However, it is okay to bring notes to the evaluation.

General Remarks
- The exercise will be evaluated in the week from the 20th-25th May. Every group member must be able to present the solution and answer upcoming questions.