

## Excercise Sheet 7

### Solution

## Lecture Distributed Systems

### Winter Term 2025/26

#### Exercise 1: Process Migration

There are several state components of a process that are stored in the OS, e.g.

- the table of open files (the process could not access them after migration)
- the signal mask (the process might not handle certain signals correctly anymore)
- alarm timer set (the process would not receive the alarm signal anymore)
- etc.

So at least this information of the OS kernel would have to be transferred. Alternatively, you can send system calls back to the source node (and forward signals etc. from it to the new node).

A migration in the described way is of course only possible between computers that use exactly the same CPU architecture and OS version.

#### Exercise 2: Policies of Dynamic Load Balancing Systems

- a) The **Transfer Policy** describes how a computer decides whether (or when) to move workload. Typically, this is determined by load thresholds. If a computer has a load that is greater than this threshold, it will surrender tasks (processes or jobs). Conversely, a node whose load is less than a threshold can request tasks from other nodes.

Once a node is set as a sender, it must choose which task to send to another node (**Selection Policy**). It must be ensured that the overhead caused by moving the task does not nullify the benefit of the move. One method would be to send new tasks that have just entered the queue. This ensures that a task that needs to be sent has not already been executed, i.e. has no state.

The **Location Policy** describes how to determine a receiver node for the tasks to be moved (or a sender node if the receiver initiates load balancing). A very simple way is to take a random node. It would be better to ask several nodes about their load and select the least loaded node as the receiver.

The **Information Policy** describes how/when the load information is determined and distributed. For example, the load can be determined regularly, or event-driven (if the load situation changes, e.g. when tasks are created or terminated). The load information can be distributed globally to all nodes or only locally to the nearest neighbors.

- b) Receiver-initiated methods can result in a slowdown of the system (with low load). The performance of the system decreases.

If the system never has a high load, then usually all nodes by the transfer policy choose to be a receiver. If we use receiver-initiated procedures, each node will constantly send requests for tasks, and there are no senders to fulfill them. These task requests fill the network with unnecessary traffic and can slow down actual task transmissions, thereby reducing performance.

The latency between sending a request and a task arriving can also cause idle times on nodes, although there may be enough tasks in total to keep all nodes busy.

### Exercise 3: Clock Drift

- a) A simple example is the distributed compilation of a program using *make*. Make is controlled by a so-called makefile, in which dependencies e.g. between source and object files are specified, as well as rules specifying how target files can be created from source files. For optimization reasons, such a rule is only applied if the source file has changed since the last time it was used, so that only really necessary compilation processes take place. Make is based on the time stamps of the files stored in the file system that indicate the last change. If the clocks on the individual computers involved deviate from each other, inconsistent states may occur, e.g. a source file may have an older date than the corresponding object file, even though the source code was changed after the last translation process. Erroneously, the recompilation of this file would not take place.
- b) With the additional leap seconds inserted, some days take exactly 86401 seconds instead of 86400. This is necessary because an earth rotation does not take exactly  $24 * 60 * 60$  seconds, but a small fraction of a second longer. So leap seconds have nothing to do with a possible inaccuracy of the atomic clock, but only serve to adapt to the calendar.

### Exercise 4: Clock Synchronisation

There are several ways to synchronize distributed clocks. One of these is Cristian's protocol. A client process  $P$  requests the time in a message  $m_r$  from a time server  $S$ , and receives the time  $t$  in a message  $m_t$  from  $S$ . The problem is that because of the signal propagation time  $m_t$ , the time  $t$  is out of date again when arriving at  $P$ . Therefore,  $P$  measures the round trip time  $T_{round}$  from sending  $m_r$  to receiving  $m_t$  and sets its clock to  $t + T_{round}/2$  ( $T_{round}/2$  is the estimated propagation time of the message  $m_t$ ).

The accuracy of the procedure is limited by:

- the assumption that the runtimes of  $m_r$  and  $m_t$  are the same (systematic error).
- the variations in the message durations (statistical error).

In a LAN, the variations (jitter) occurs due to competition for the medium (collisions, buffer times in switches) and due to the processing of messages in the operating systems of  $P$  and  $S$ . For a LAN, the accuracy is usually within  $1ms$ . In the Internet, additional inaccuracies occur because the messages are delayed in the routers. For WANs, the accuracy is probably within  $5 - 10ms$ .

Also with the help of GPS an exact synchronization of clocks is not possible. Due to the known position of satellites and the GPS receiver, the message transfer time can be calculated very accurately, but still varies, e.g., due to weather conditions. In addition, the positions are also afflicted with errors.