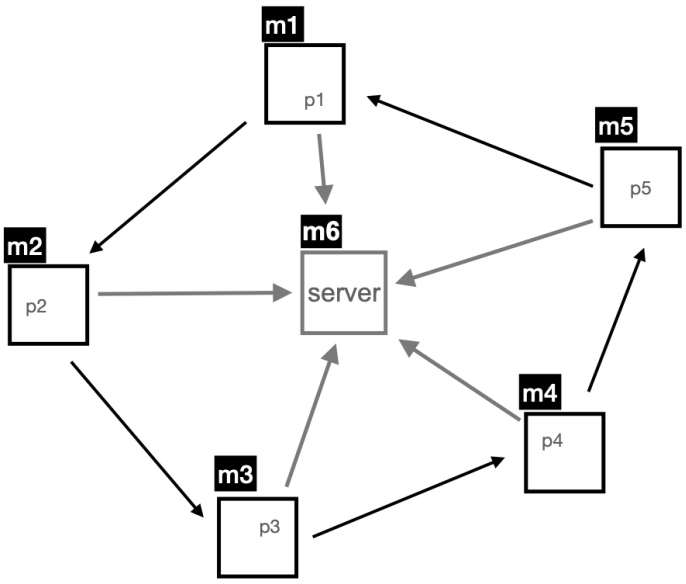
Distributed Systems
Practical Assignment
– 2025/26 –

1. Mutual Exclusion with the Token Ring Algorithm



⟹ Read van Steen & Tanenbaum (Chapter 6, "Token Ring Algorithm").

⟹ Create a ring network with 5 peers (call them `p1` to `p5`) such as the one represented in the figure above. Each peer must be in a different machine (`m1` to `m5`). Besides these 5 peers, create also a `calculatormulti` server called `server` that runs on machine `m6`.

⟹ Each peer only knows the IP address of the machine where the next peer is located: `p1` knows the IP of `m2`, `p2` knows the IP of `m3`, ..., `p5` has the IP of `m1`, thus closing the ring. All peers know the IP address of the machine where the `server` is located (`m6`). These can be passed to the peer via the command line, e.g., for peer `p2` you would run the following command in machine `m2`: `$ peer IP-of-m3 IP-of-m6`.

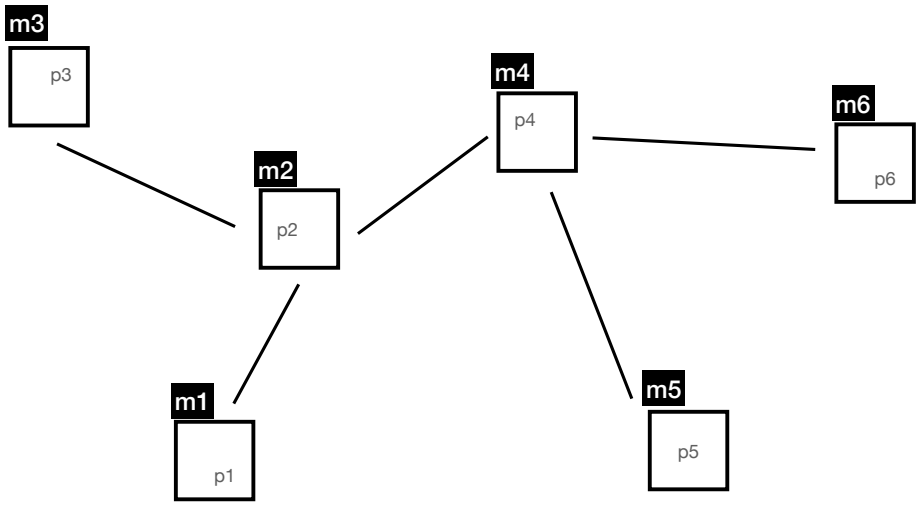
⟹ One of the threads in each peer generates requests for the `server` following a Poisson distribution with a frequency of 4 per minute. The operation and the arguments for each request are also random. These requests are placed in a local queue.

$\Longrightarrow$ Another thread in that peer runs in a loop waiting for a message (that can only come from the previous peer). This message is designated a *token*. The token can be a void message. The peer that holds it at any given moment has exclusive access to `server`, effectively implementing mutual exclusion.

$\Longrightarrow$ When the peer receives the token, it checks if it has requests for the server in its local queue. If it does, it holds the token until all requests are processed at the server; after the results are received and printed on the terminal, the peer restarts the forwarding of the token. If it does not, it forwards the token to the next peer in the ring.

$\Longrightarrow$ [**EXTRA MARKS:**] implement a scheme that enables the token to continue to move in the event of a failure in one of the peers.

2. Data Aggregation in a P2P Network



$\Longrightarrow$ Read van Steen & Tanenbaum (Chapter 6, on "Data Dissemination Algorithms").

$\Longrightarrow$ Create a network of 6 peers (`p1` to `p6`), running on different machines (`m1` to `m6`), with the topology shown in the figure above.

$\Longrightarrow$ Peers must have a thread waiting for others to connect and register themselves. This is how peers get to know each other, and the network is built.

$\Longrightarrow$ Each peer keeps a map data structure (e.g., Map or HashMap in Java) with the IPs/names of the machines of the peers that have registered themselves with it. For example, peer `p2` (in machine `m2`) keeps the entries `[m1, m3, m4]` whereas peer `p6` (in machine `m6`) keeps only `[m4]`.

$\Longrightarrow$ Each peer initially generates a random double value v in the range $(0, 1)$ and exchanges this value with its neighbours.

$\Longrightarrow$ This synchronization process uses the Anti-Entropy algorithm and follows a Poisson distribution with a frequency of 2 per minute. A peer chooses a random neighbour from its map and synchronizes the current value with it.

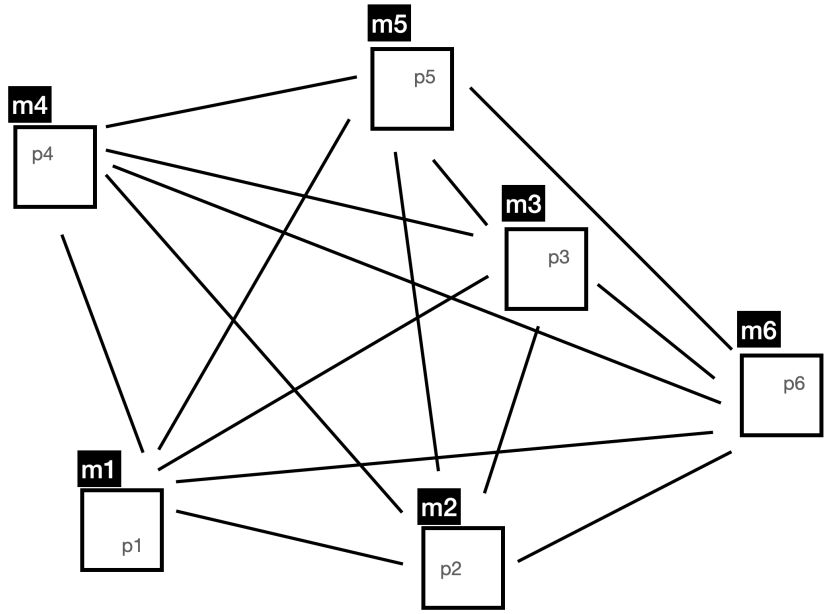
$\Longrightarrow$ In a synchronization, two peers P_i and P_j exchange their current values. Afterwards, the resulting value in both P_i and P_j will be $(v_i + v_j)/2$.

$\Longrightarrow$ Once this process starts, the value held by each peer will converge to the average of all values originally generated by the peers.

$\Longrightarrow$ Change your peers to allow them to receive a specific value when they start running. Try this with all peers set to 0 except for one that is initialized to 1. Observe the behavior of the system. Notice that this converges to the average of all values, which will be, in this case, $1/N$ where N is the number of peers in the network. Thus, after a while, the size of the network will be known to all peers.

$\Longrightarrow$ [**EXTRA MARKS:**] create a script that builds a *connected* network of N such peers with a random topology. As above, the peers should be initialized to 0 except for one, which is initialized to 1. Run simulations for different values of N. Create a plot that shows the convergence time towards $1/N$ as a function of N.

3. A Basic Chat Application Using Totally-Ordered Multicast



$\Longrightarrow$ Read van Steen & Tanenbaum (Chapter 6, "Lamport Clocks" and "Totally Ordered Multicast").

$\Longrightarrow$ Create a network of 6 peers (`p1` to `p6`) each in a different machine (`m1` to `m6`) with the topology shown in the figure above. Each peer has a table with the IP addresses of all the other peers/machines.

$\Longrightarrow$ Implement Lamport clocks in each peer so that messages can be adequately timestamped.

$\Longrightarrow$ Each peer sends a random word according to a Poisson distribution with a frequency of 1 per second. You can use a text format dictionary from the Internet and extract its keywords into a set. Then choose a random keyword from that set. Each word is sent in a message to all other peers (they are all in the IP table).

$\Longrightarrow$ The goal is that all peers print the same sequence of words. To do this, the peers must agree on a global order for the messages before processing them (print the words therein). This is not trivial, as factors such as communication latency and varying network topology affect message delivery even in small networks.

$\Longrightarrow$ To achieve this, you must implement the Totally-Ordered Multicast (TOM) Algorithm using Lamport clocks to timestamp the messages. Check Chapter 6 of van Steen and Tanenbaum for the details of the algorithm and here for another, detailed, description).

$\Longrightarrow$ Note that in TOM there is a difference between receiving and processing a message. Processes always receive incoming messages, but they are processed only when specific conditions are met. In this application, peers process messages by printing the words therein. If you correctly implemented the TOM algorithm the printed list of words must be the same for all peers.

$\Longrightarrow$ **[EXTRA MARKS:]** Design and implement a mechanism that allows the distributed system to detect and prevent malicious peers from trying to compromise the protocol by: (a) "rewriting history" using an already used timestamp value; (b) "writing in the future" using a timestamp larger than the one expected.

General Remarks

The practical assignment is individual.

We suggest using Java to implement the three scenarios. If using another programming language, please check with us first. You can use RPC frameworks such as gRPC. Assuming Java will be your choice, the software you will produce should be organized into 3 packages as follows:

$\Longrightarrow$ `ds.assignment.tring`

$\Longrightarrow$ `ds.assignment.p2p`

$\Longrightarrow$ `ds.assignment.tom`

The deadline for code submission is January 5th, 2026.

To submit your code, please send us both an email with the subject `DS2526` with a `.ZIP` attached. This file should contain the source code for the three packages and a text file named `README` that explains how to compile and run the examples. After the submission deadline, you must present your work. The presentations will take place during the first full

week of January (more details about the schedule will be given later). To speed things up, we suggest writing scripts that automatically set up the peers and networks. The exercises have the same weight in the practical assignment grade. [**EXTRA MARKS**] represent 20% of the grade of each exercise.

Not presenting your work results in a grade of 0 ("zero") in the practical assignment.

Enjoy your work,

Luís Lopes (`lmlopes@fc.up.pt`)
João Soares (`joao.soares@fc.up.pt`)