# IN.5022 — Concurrent and Distributed Computing

Series 7 and 8

*Due dates: 9.11.2022 and 16.11.2021, 12:00, on Moodle*

***About these series***

The objective of these series is to develop a small library (module) for group communication (GC). Carefully read the course material and the algorithms presented during the lectures.

To reduce the complexity, we will make several simplifying assumptions:

1. Groups are closed and static, and we will only consider one group (but its size can be chosen at runtime).
2. All processes can run in the same virtual machine as part of a single Erlang/Elixir program. Deployment on different remote machines (or *Dockers*) is optional.
3. We do not care about memory usage (e.g., no need to garbage-collect message lists).
4. We assume reliable channels and we do not consider process failures (except for one optional scenario, see below). The primitives provided as part of the library will essentially focus on providing ordering guarantees.
5. The primitives will be tested by simulating communication patterns with delays (fixed or random) to trigger out-of-order delivery, as shown in the course material. These scenarios can be hardcoded but should demonstrate the correct execution of the algorithms.

**Exercise 1 (due 9.11.2022)**

Sketch a first version of the GC module with functionality for:

1. Creating a group with *N* members, where *N* is a parameter. This will lead to the creation of *N* Erlang/Elixir processes. The group structure can (but does not need to) be implemented as a list containing tuples with the number and process identifier of each group member. Depending on your design, each group member can spawn additional processes acting as endpoints for communication channels.
2. Closing the group, i.e., terminating all processes that have been created.
3. Multicasting a message to the group with no reliability or ordering guarantee, i.e., the *B-multicast* primitive presented in the lecture.
4. Executing at least one sample communication scenario to create the group, multicast a few messages (injecting delays as required) and exhibiting out-of-order reception at some group members.

Report the result of executing the scenario(s), e.g., with a dump or screenshot of the terminal output.

**Exercise 2 (due 16.11.2022)**

Extend the GC module to support the following primitives presented during the lecture:

1. *R-multicast* (reliable) — mandatory
2. *F-multicast* (FIFO) — mandatory
3. *C-multicast* (causal) — optional

For each of the latter two primitives, develop a communication scenario that shows that the order properties are not met when using *R-multicast*, but they are when using the ordered multicast algorithm.

Note that, while you are encouraged to stack the protocols as presented in the lecture, i.e., implement them using transformations, you are not obliged to do so: you can also implement each algorithm in a standalone manner. In any case, you might want to have separate Erlang/Elixir processes for the different algorithms.

Optionally, to demonstrate reliability, you can trigger a crash failure while a process is halfway through multicasting a message, i.e., after having it sent to only some but not all group members.

Report the result of the execution of these scenarios.

**Exercise 3 (optional)**

Add support for the *CO-multicast* algorithm from Isis that uses vector clocks, as presented in the lecture. To that end, you will need to develop some functions to create and compare vector timestamps. The algorithm can be implemented in a standalone manner, i.e., without dependencies with the other multicast algorithms of the GC module.

**Exercise 4 (optional)**

Deploy and test your programs on multiple machines.