Prof. Dr. Annette Bieniusa
Philipp Lersch, M. Sc.

RPTU Kaiserslautern-Landau
Fachbereich Informatik
AG Softwaretechnik

Exercise 5: Programming Distributed Systems (Summer 2025)

Submission

Deadline: 10.6.2025 AOE

- You need a team and a Gitlab repository for this exercise sheet.
- In your Git repository, create a branch for this exercise sheet, for example with

```
git checkout -b ex5
```

- Create a folder named “ex5” in your repository and add your solutions to this folder.
- Create a merge request in Gitlab and assign Philipp Lersch as assignee. If you do not want to get feedback on your solution, you can merge it by yourself.

1 Total-order broadcast and consensus

In the lecture you have seen, that Total-order broadcast can be implemented using consensus.

a) Show that it is also possible to do the reverse: Describe an algorithm that implements consensus by using total-order broadcast. Your algorithm should provide a *propose* method to clients and trigger a *decide*-event when a proposed value is decided.

b) Assume we change the total-order broadcast algorithm from the lecture and only use best-effort broadcast instead of reliable broadcast. Show that this would make the algorithm incorrect by giving an execution that violates one of the properties of consensus.

c) Consider the following modification to the total-order broadcast algorithm from the lecture. Instead of two sets `delivered` and `received`, we only use a single set `pending`, which should contain the messages that have been received but not yet delivered.

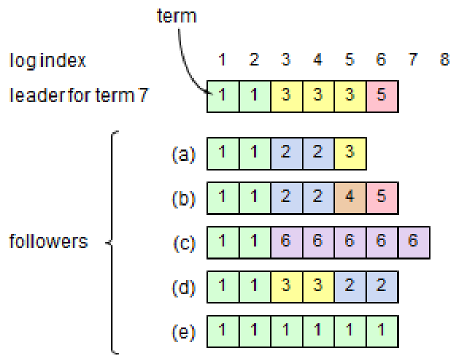
Why does this “optimization” not work correctly?

```
State:
  $k_p$           // consensus number
  pending    // messages received but not a-delivered by process

Upon Init do:
  $k_p$ ← 0;
  pending ← ∅;
Upon to-Broadcast(m) do
  trigger rb-Broadcast(m);
Upon rb-Deliver(m) do
  pending ← pending ∪ {m};

Upon pending ≠ ∅ do
  $k_p$ ← $k_p$ + 1;
  propose($k_p$, pending);
  wait until decide($k_p$, msg$^{k_p}$)
  ∀ m in msg$^{k_p}$ in deterministic order do trigger to-Deliver(m)
  pending ← pending \ msg$^{k_p}$
```

2 Raft

To learn about the details of the Raft algorithm, read the Raft paper "In Search of an Understandable Consensus Algorithm (Extended Version)" by Diego Ongaro and John Ousterhout (`https://raft.github.io/raft.pdf`). You can also watch the video at `https://youtu.be/YbZ3zDzDnrw`.

a) Consider the figure below, which displays the logs in a cluster of 6 servers just after a new leader has just been elected for term 7 (the contents of log entries are not shown; just their indexes and terms). For each of the followers in the figure, could the given log configuration occur in a properly functioning Raft system? If yes, describe how this could happen; if no, explain why it could not happen.



b) Suppose that a hardware or software error corrupts the nextIndex value stored by the leader for a particular follower. Could this compromise the safety of the system? Explain your answer briefly.

c) Suppose that you implemented Raft and deployed it with all servers in the same datacenter. Now suppose that you were going to deploy the system with each server in a different datacenter, spread over the world. What changes would you need to make, if any, in the wide-area version of Raft compared to the single-datacenter version, and why?

d) Each follower stores 3 pieces of information on its disk: its current term, its most recent vote, and all of the log entries it has accepted.

1) Suppose that the follower crashes, and when it restarts, its most recent vote has been lost. Is it safe for the follower to rejoin the cluster (assuming no modifications to the algorithm)? Explain your answer.

2) Now suppose that the follower's log is truncated during a crash, losing some of the entries at the end. Is it safe for the follower to rejoin the cluster (assuming no modifications to the algorithm)? Explain your answer.

3 Implementing State Machine Replication

In this exercise you will implement an online seat plan reservation service "Repseat", where users can reserve a space for an upcoming event.

Since users can get pretty disappointed, when their favorite event is not available, we want to use replication for high availability. We also want high consistency for dealing with money and demanding users and therefore will employ replicated state machines using the Raft algorithm.

The module `Repseat.Raft.Machine` contains the implementation of the replicated state machine and stubs for the functions you need to implement to solve this task. The comments in this file explain how to use and adapt the replicated state machine. We use the library "ra"[1] as Raft library.

[1] Source code is available at `https://github.com/rabbitmq/ra`. You can find some documentation at `https://rabbitmq.github.io/ra/`. The relevant parts are in the description of the `ra` and `ra_machine` modules.