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.

```
State:
  // We only want to decide on one value, so remember if we have already
    decided
  Done ← false

Upon propose(Msg) do
  trigger to-Broadcast(Msg)

Upon to-Deliver(Msg) do
  if not Done
    Done ← true
    trigger decide(Msg)
```

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.

Recall: Properties of Consensus

- Uniform Agreement: Every correct process must decide on the same value.
- Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- Termination: All processes eventually reach a decision.

Example:

- 3 Processes
- P1: to-broadcast(M)
- be broadcast delivers M on P1 and P2
- P1 crashes, be-broadcast fails to deliver M to P3
- P3 never participates in consensus and does not learn about M
- (Still, it might catch up if other messages trigger participation in 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}
```

A message could be delivered twice (once via propose from P1, once proposed from P2 if P2 rb-delivers the message after the decide from P1)

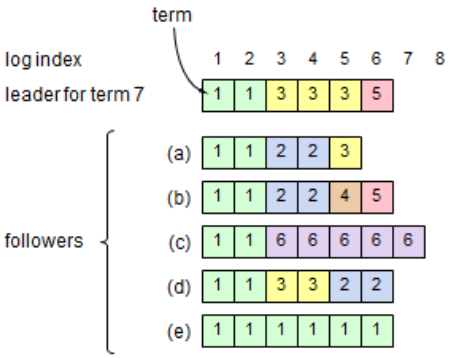
Compare this to the 2-phase CRDT set.

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`.

The answers are available on the same Raft page: `https://ramcloud.stanford.edu/~ongaro/userstudy/rubric.pdf`

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.



a) No. Entry $< 5, 3 >$ uniquely identifies a log prefix (by the AppendEntries consistency check), but this follower has entry $< 5, 3 >$ and a different log prefix before it than the leader.

b) No. Entry $< 6, 5 >$ uniquely identifies a log prefix (by the AppendEntries consistency check), but this follower has entry $< 6, 5 >$ and a different log prefix before it than the leader.

c) Yes. Since we can't say much about the other servers in the cluster, this server could have been leader in term 6 with a starting log of $< 1, 1 >, < 2, 1 >$ and could have written a bunch of entries to its log and not communicated with our current leader of term 7. This assumes that entries $< 3, 3 >, < 4, 3 >, < 5, 3 >$, and $< 6, 5 >$ were not committed in term 5, which is possible.

More concretely, the following execution with 3 participants could lead to this state:

1. Logs after Term 1:

Node 1:	1	1					
Node 2:	1	1					
Node 3:	1	1					

2. Because of a timeout, a leader election starts, but no one gets a majority for term 2.

3. In the next election, node 1 is elected for term 3. As the leader Node 1 appends 3 entries for term 3.

Node 1:	1	1	3	3	3		
Node 2:	1	1					
Node 3:	1	1					

4. Because of a timeout, a leader election starts, but no one gets a majority for term 4.

5. In the next election, node 1 is elected for term 5. As the leader it appends one entry for term 5.

Node 1:	1	1	3	3	3	5	
Node 2:	1	1					
Node 3:	1	1					

6. Because of a timeout, a leader election starts and node 3 gets elected as leader for term 6 with the votes from itself and node 2. As the leader it appends 5 entries for term 6.

Node 1:	1	1	3	3	3	5	
Node 2:	1	1					
Node 3:	1	1	6	6	6	6	6

7. Because of a timeout, a leader election starts and node 1 gets elected as leader for term 7 with the votes from itself and node 2.

d) No. Terms increase monotonically in a log. Specifically, the leader that created entry $< 5, 2 >$ could only have received $< 4, 3 >$ from a leader with current term ≥ 3, so its current term would also be ≥ 3. Then it could not create $< 5, 2 >$.

e) Yes. For example, (e) is the leader for term 1 and commits entries $< 1, 1 >$ and $< 2, 1 >$, then becomes partitioned from the other servers but continues processing client requests.

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.

No.

If the nextIndex value is too small, the leader will send extra AppendEntries requests. Each will have no effect on the follower's log (they will pass the consistency check but not conflict with any entries in the follower's log or provide any entries to the follower that the follower didn't already have), and the successful response will indicate to the leader that it should increase its nextIndex.

If the nextIndex value is too large, the leader will also send extra AppendEntries requests. The consistency check will fail on these, causing the follower to reject the request and the leader to decrement nextIndex and retry.

Either way, this is safe behavior, as no critical state is modified in either case.

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?

We'd need to set the election timeouts higher: A server's expected broadcast time is higher, and the election timeout should be much higher than the broadcast time so that candidates have a chance to complete an election before timing out again. The rest of the algorithm does not require any changes, since it does not depend on timing.

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.

- 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.

  No. This would allow a server to vote twice in the same term, which would then allow multiple leaders per term, which breaks just about everything.

- 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.

> No. This would allow a committed entry to not be stored on a quorum, which would then allow some other entry to be committed for the same index.

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.