Dr. Annette Bieniusa
Albert Schimpf, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Exercise 5: Programming Distributed Systems (Summer 2020)

Submission

- 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 Albert Schimpf as assignee. If you do not want to get feedback on your solution, you can merge it by yourself.
- Test your submission with the provided test cases. Feel free to add more tests, but do not change the existing test cases.

1 Specifying Causal Order

Alice and Bob come up with two different variants of the causal-order property:

Causal-Broadcast CB_A If p_i delivers m, then p_i must first deliver every message m' with $m' \rightarrow m$.

Causal-Broadcast CB_B If p_i delivers m' and m and $m' \rightarrow m$, then p_i must deliver m' before m.

- Give an examplary execution to show that CB_A and CB_B are not equivalent.
- Which one is more general, i.e. does $CB_A \Rightarrow CB_B$ or $CB_B \Rightarrow CB_A$?

2 Causal Broadcast

Give an example execution, which shows that the following algorithm does not correctly implement causal broadcast.

```
State:
  pending   // set of messages that cannot be delivered yet
  delivered // set of delivered message-ids
  last      // message-id of last received message
Upon Init do:
  pending <- ∅;
  delivered <- {none};
  last <- none;

Upon rco-Broadcast(m) do
  trigger rco-Deliver(self, m);
  uid <- generateUniqueId(m);
  trigger rb-Broadcast(uid, last, m);
  delivered <- delivered ∪ {uid};
  last <- uid;

Upon rb-Deliver(p, uid, last_m, m) do
  if ( p ≠ self ) then
    pending <- pending ∪ {(p, uid, last_m, m)};
    while exists (q, uid, last_m, m_q) ∈ pending such that last_m ∈ delivered
      pending <- pending \ {(q, uid, last_m, m_q)};
      trigger rco-Deliver(q, m_q);
      delivered <- delivered ∪ {uid}
      last <- uid
```

3 Causal Reliable Broadcast - Improved

- Give a variant of the Causal Reliable Broadcast Algorithm with no waiting that builds on fifo-Broadcast.
- Assuming that *no process ever fails*, add some form of garbage collection on the local past.

4 Implementing the Broadcast Algorithms

The algorithms you will implement in the tasks below are based on a link layer, which is provided in the template for this exercise. This link layer implements an abstraction of the communication network that simplifies the testing of your implemenations.

The `link_layer` module provides the following functions, that all take the link-layer instance `LL` as their first argument:

```
%% sends Data to other Node
send(LL, Data, Node)
%% Registers a receiver for the broadcast
register(LL, Receiver)
%% get a list of all nodes/prcoesses (including itself)
all_nodes(LL)
%% get a list of all other nodes/processes (excluding itself)
other_nodes(LL)
%% get a descriptor for this node
this_node(LL)
```

You can assume that the link layer implements the perfect-link model as discussed in the lecture.

4.1 Best-effort broadcast

Implement a module named `best_effort_broadcast`, which implements the best-effort broadcast algorithm from the lecture.

The module should provide the following exported functions:

1. A function `start_link(LinkLayer, RespondTo)`, which starts a process handling the algorithm. If it succeeds, the function returns a tuple `{ok, Pid}`, where `Pid` is a process id used in later calls to `broadcast` (see below). The first argument of the function is a reference to the link-layer process, which is to be used for communicating with other nodes (see above). The second argument is a process id for the process representing the application/higher level. When delivering a broadcast message `Msg`, the tuple `{deliver, Msg}` should be sent to this process.
2. A function `broadcast(Pid, Messsage)`, which broadcasts a message to all processes participating in the broadcast. The first argument is the process id returned by `start_link`, the second argument is the message to be broadcast. The return value should be the atom `ok`.

4.2 Reliable broadcast

Implement a module named `reliable_broadcast`, which implements the reliable broadcast algorithm from the lecture.

The module should provide the `start_link(LinkLayer, RespondTo)` and `broadcast(Pid, Messsage)` functions, similar to the `best_effort_broadcast` module.

4.3 Causal broadcast

Implement a module named `causal_broadcast`, which implements the causal broadcast algorithm 2 (waiting) from the lecture.

Again, the module should provide the `start_link(LinkLayer, RespondTo)` and `broadcast(Pid, Messsage)` functions. To deliver a broadcast, it should send a message `{deliver, Msg}`.