Dr. Annette Bieniusa
M.Sc. Peter Zeller
M.Sc. Mathias Weber

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Exercise 2: Programming Distributed Systems (SS 2018)

- To get feedback to your solution: Create a new branch in your git repository (e.g. `git checkout -b ex2`). Submit your solution to the programming exercise via your group's repository to the new branch in a folder named "ex2". When your solution is ready, create a merge request in Gitlab and assign Peter Zeller to it. This will allow us to comment on your code and give you feedback.
- Test your submission with the provided test cases. Feel free to add more tests, but do not change the given test files, as we might update them later.
- Prepare task 1, 2 and 3 for the exercise on Thursday, April 26.
- Prepare task 4 and 5 for the exercise on Thursday, May 3.

1 Vector clocks and causal broadcast

A vector clock is a mapping from processes to positive integers. Implement a module named `vectorclock` with the following functions:

- `new()` creates a new vector clock, where all processes have value 0.
- `increment(VC, P)` increments the entry of process `P` by 1.
- `get(VC, P)` returns the value for process `P`.
- `leq(VC1, VC2)` checks, whether `VC1` is less than or equal to `VC2`. This is the case, iff $\forall P.\ get(VC_1, P) \leq get(VC_2, P)$.
- `merge(VC1, VC2)` merges two vector clocks by computing their least upper bound (the smallest vector clock `V`, such that $VC_1 \leq V$ and $VC_2 \leq V$).

2 Causal Broadcast

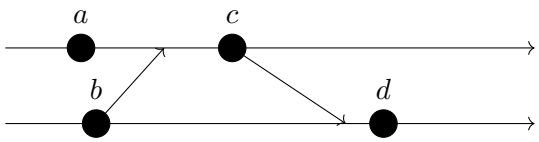
Give an example execution, which shows that the following algorithm does not correctly implement causal broadcast.

```
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
```

Only considering the last message is not sufficient. In the example below, when message b is delivered to node 1 it will be the last messsage, so message c will have b as its last messsage. When c is delivered to node 2 before message a is delivered it would therefore be accepted. This vioalates causality.



Link layer

The algorithms you will implement in the tasks below are based on a link-layer, which is provided by us (included in template for this exercise) and implements the communication network. You can assume that this layer implements the perfect-link model.

To use it, use the `link_layer` module, which 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: all future messages will be delivered
%% to the registered process (Receiver)
register(LL, Receiver)
%% get a list of all nodes (including own node)
all_nodes(LL)
%% get a list of all other nodes
other_nodes(LL)
%% get this node
this_node(LL)
```

3 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. On success the function returns a tuple `{ok, Beb}`, where `Beb` 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. When delivering a broadcast message `Msg`, the tuple `{beb_deliver, Msg}` should be sent to this process.
2. A function `broadcast(Beb, Messsage)`, which broadcasts a message to all participating processes. The first argument is the process-id returned by `start_link`, the second argument is the message to send. The return value should be the atom `ok`.

4 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(Beb, Messsage)` functions, similar to the `best_effort_broadcast` module. However, instead of sending a

message `{beb_deliver, Msg}` it should send a message `{rb_deliver, Msg}` to deliver the broadcast.

5 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(Beb, Messsage)` functions. To deliver a broadcast it should send a message `{rco_deliver, Msg}`.