Exercise 2: Programming Distributed Systems (SS 2019)

- 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 these tasks 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 `∀P. get(VC1, P) ≤ get(VC2, P)`.
- `merge(VC1, VC2)` merges two vector clocks by computing their least upper bound (the smallest vector clock `V`, such that `VC1 ≤ V` and `VC2 ≤ 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, lastm, m) do**
- `if ( p ≠ self ) then`
  - `pending <- pending ∪ {(p, uid, lastm, m)};`
  - `while exists (q, uid, lastm, mq) ∈ pending such that lastm ∈ delivered`
    - `pending <- pending \ {(q, uid, lastm, mq)};`
    - `trigger rco-Deliver(q, mq);`
    - `delivered <- delivered ∪ {uid}`
    - `last <- uid;`
**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:

- `send(LL, Data, Node)` sends Data to other Node
- `register(LL, Receiver)` registers a receiver: all future messages will be delivered to the registered process (Receiver)
- `all_nodes(LL)` gets a list of all nodes (including own node)
- `other_nodes(LL)` gets a list of all other nodes
- `this_node(LL)` gets this node

### 3 Best-effort broadcast

Implement a module named `best_effortBroadcast`, 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 `{deliver, Msg}` should be sent to this process.

2. A function `broadcast(Beb, Message)`, 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 `reliableBroadcast`, which implements the reliable broadcast algorithm from the lecture.

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

### 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, Message)` functions. To deliver a broadcast it should send a message `{deliver, Msg}`.
Notes

1. In the literature it is often assumed that processes are numbered which allows to write down clocks like $[4,7,3]$ or \begin{pmatrix} 4 \\ 7 \\ 3 \end{pmatrix}$ instead of the longer \{$p_1 \mapsto 4, p_2 \mapsto 7, p_3 \mapsto 3$\}. However, in this exercise we do not assume that the number of processes is known and arbitrary terms can be used as process names.