Dr. Annette Bieniusa
Albert Schimpf, M. Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

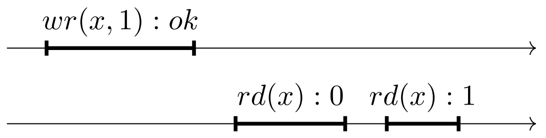
Exercise 9: Programming Distributed Systems (Summer 2020)

1 Consistency Models

For the following executions, decide which of the consistency models Linearizability (L), SequentialConsistency (SC), CausalConsistency (CC), and BasicEventualConsistency (EC) would allow the execution. Multiple or no options might apply. Also state how confident you are in each answer by giving a value between 0% and 100%.

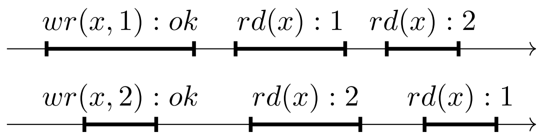
Each execution uses read (rd) and write (wr) operations on registers x and y. The initial value of registers is 0.

a)



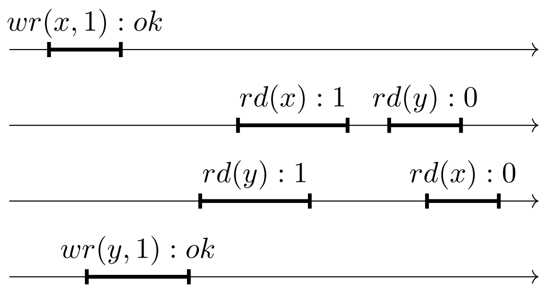
L	SC	CC	EC	Confidence (%)
☐	☐	☐	☐	

b)



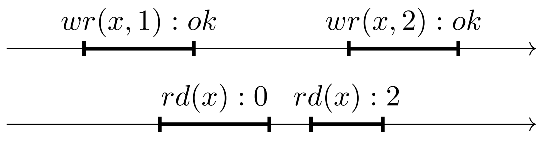
L	SC	CC	EC	Confidence (%)
☐	☐	☐	☐	

c)



L	SC	CC	EC	Confidence (%)
☐	☐	☐	☐	

d)



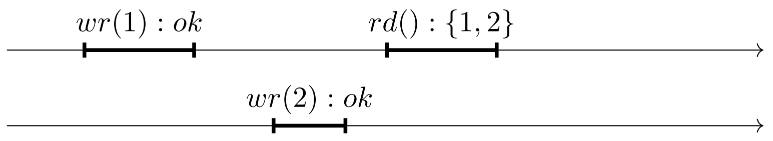
L	SC	CC	EC	Confidence (%)
☐	☐	☐	☐	

2 Concrete and abstract executions

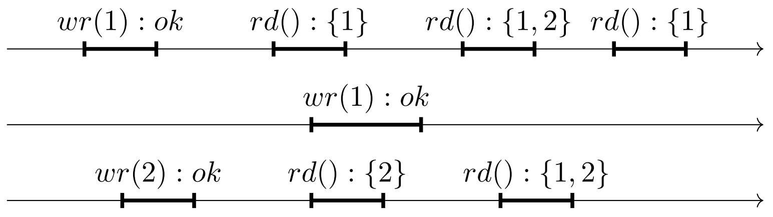
Consider the following concrete executions with operations on a Multi-Value Register object and provide an abstract execution that explains, why the execution is allowed under causal consistency.

Hint: To give an abstract execution, you need to specify the relation vis, which you can visualize in the pictures below by drawing arrows between operations.

a)



b)



3 CRDTs

Consider the following definition of an Add-wins Set (Observed-remove Set, non-optimized version):

```
payload set $E$, set $T$                    ▷ $E$: elements, $T$: tombstones
                    ▷ sets of pairs { (element e, unique-tag n), ... }
initial $\varnothing, \varnothing$
query contains (element $e$): boolean $b$
    let $b = (\exists n : (e, n) \in E)$
query elements (): set $S$
    let $S = \{e \mid \exists n : (e, n) \in E\}$
compare ($A$, $B$): boolean $b$
    let $b = ((A.E \cup A.T) \subseteq (B.E \cup B.T)) \wedge (A.T \subseteq B.T)$
merge ($B$)
    $E := (E \setminus B.T) \cup (B.E \setminus T)$
    $T := T \cup B.T$
```

Show that **merge** of an Add-wins Set is commutative, associative, and idempotent.

4 Registers

To better understand the differences between the registers discussed in the lecture, try to find example executions that show the differences:

a) Give an execution that is not allowed for a safe register.

b) Give an execution that is allowed for a safe register but not for a regular register.

c) Give an execution that is allowed for a regular register but not for an atomic register.

5 Final Project: A causally consistent CRDT database

For the final project you will develop a replicated data store named Microdote [1] . The database should be able to run replicated on multiple (2 - 10) machines. Each replica is a full replica (eventually) storing all the data. The database must be highly available and provide low latency, so every replica should be able to handle requests, even if it is temporarily disconnected from others.

Data model: Microdote is a key-CRDT store: Each replicated data object is stored under a key. The store provides an API to read the current state of an object given a key and to update objects. The supported update operations depend on the data type of the object. For example a counter supports increment- and decrement operations, while a set supports add- and remove-operations.

We will use the Antidote CRDT library[2] to support a variety of replicated data types. Check the readme file of the library and the lecture on CRDTs for information on how to use the library.

API: We use a Protocol Buffer[3] interface to let clients written in a variety of languages interact with our database. We will reuse the protocol buffer interface of the Antidote database, so that we can reuse existing clients and benchmarks. The code to handle protocol buffer requests is already provided in the template. It will call the `read_objects` and `update_objects` functions in the `microdote` module, which you have to implement. The details of this API are explained below in 5.1.

Consistency model: The data-store must provide the following consistency guarantees:

Eventual visibility: Every event eventually becomes visible at all replicas.

Causality: If $e_1 \xrightarrow{vis} e_2$ and $e_2 \xrightarrow{vis} e_3$, then $e_1 \xrightarrow{vis} e_3$

Correct return values: Each CRDT has a specification, which maps an abstract execution to a return value (Hint: the CRDT library ensures correct return values, if you use is correctly).

For example using a multi-value register guarantees:

$$v \in rval(e) \leftrightarrow \Big(\exists e_1 \in E.\ op(e_1) = assign(v) \\ \wedge \Big(\nexists e_2 \in E.\ e_1 \xrightarrow{vis} e_2 \wedge \exists v'.\ op(e_2) = assign(v')\Big)\Big)$$

Atomic operations: When several objects are updated with one call to `update_objects`, then it should not be possible to observe a state, where some of the updates are visible and others are not.

Session guarantees: Each call e_1 to `update_objects` and `read_objects` returns a clock which identifies the database version after the operation was completed. This clock can be passed to a succeeding API call e_2. In this case, it must be guaranteed that $e_1 \xrightarrow{vis} e_2$.

[1] Named after "Antidote", a planet-scale, available, transactional database with strong semantics. You are free to change the name of your project.

[2] https://github.com/AntidoteDB/antidote_crdt

[3] https://developers.google.com/protocol-buffers/

Testing In the initial template you will find only some very basic system tests. As failing system tests are often hard to debug, we recommend that you also write smaller component tests for the code you write.

Documentation and Code style We expect that you document your code with comments and write clean and readable code.

5.1 The Microdote API

Module name: `microdote`

The Microdote is the API of a key value database, where a key is a 3-tuple consisting of a main identifier (`Key`), the CRDT type (`Type`), and a namespace (`Bucket`).

```
-type key() :: {Key :: binary(), Type :: antidote_crdt:typ(), Bucket :: binary()}.
```

The API consists of 2 functions: `read_objects` and `update_objects`. You can choose an arbitrary representation for the type `clock()` used below.

The `read_objects` function takes a list of keys and returns the value of the corresponding objects. If there are no updates for a key, the initial value for the given Type is returned. The function takes a clock value, which can be `ignore` or come from the result of another call of `read_objects` or `update_objects`. If the clock comes from another call, it is guaranteed that this read observes a state that is not older than the state after the previous call.

```
-spec read_objects([key()], clock() | ignore) ->
{ok, [any()], clock()}
| {error, any()}.
read_objects(Objects, Clock) -> ...
```

The `update_objects` function takes a list of `{Key, Operation, Args}` tuples and executes the given updates atomically. If several updates are given for the same key, the updates are performed sequentially from left to right. The function takes a clock value, which can be ignore or comes from the result of another call of `read_objects` or `update_objects`. If the clock comes from another call, it is guaranteed that this update operation is applied on a state that is not older than the state after the previous call.

```
-spec update_objects([{key(), Op :: atom(), Args :: any()}], clock()) ->
{ok, clock()}
| {error, any()}.
update_objects(Updates, Clock) -> ...
```

Example Usage:

To test the API functions you can start a single node or a cluster of nodes with the Makefile targets (see `README.md` file in the template). On the shell you can then call the API methods:

```
(microdote1@127.0.0.1)1> {ok, Clock} = microdote:update_objects([{{<<"K">>,
    antidote_crdt_counter_pn, <<"V">>}, increment, 42}], ignore).
{ok,#{'microdote1@127.0.0.1' => 1}}
(microdote1@127.0.0.1)2> microdote:read_objects([{<<"K">>, antidote_crdt_counter_pn,
     <<"V">>}], Clock).
{ok,[{{<<"K">>,antidote_crdt_counter_pn,<<"V">>},42}],#{'microdote1@127.0.0.1' => 1}}
```

5.2 Architecture

Our template for this project already includes a few components for the final system:

1. A protocol buffer interface, which clients can use to interact with the database.

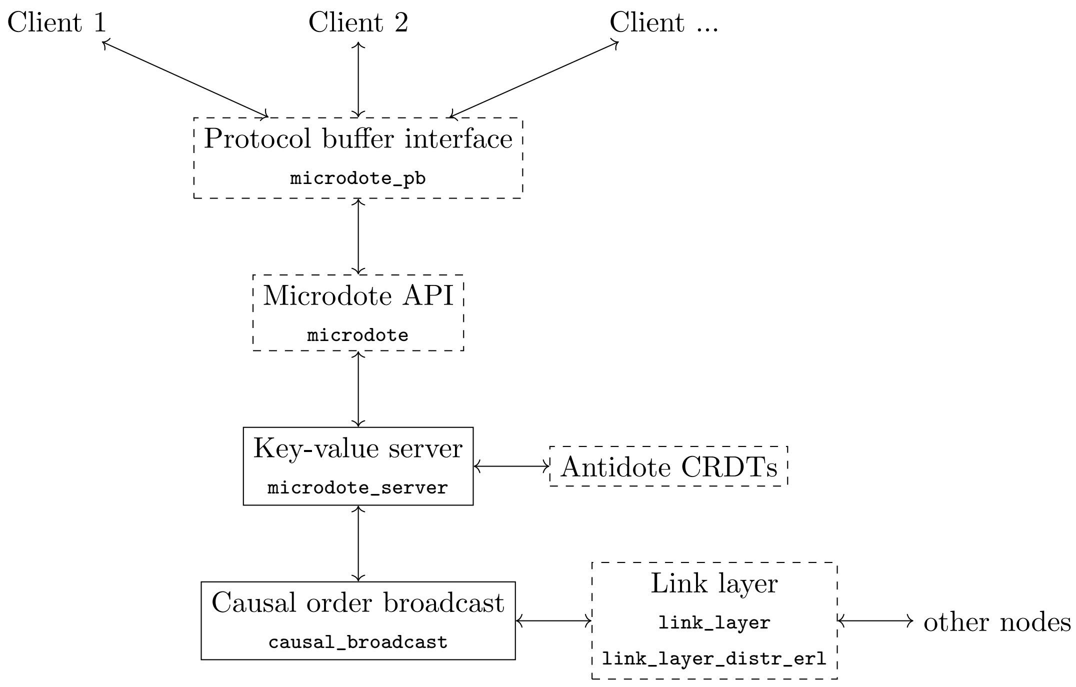
   The protocol buffer (PB) interface manages a set of sockets (using the ranch library). Clients connect through these sockets and send requests as PB messages. The PB module translates these messages to Erlang terms, and calls the key-value server. The result from the key-value server is again encoded into a PB response and sent back to the client.

2. The Antidote CRDT library.

3. The link layer you know from previous exercises.

You have to implement the API in the `microdote` module (see 5.1), for which you will probably need to implement other modules.

You are free to come up with your own architecture to implement the system. One possible architecture is sketched below:



To implement Microdote with this architecture you can follow these steps:

1. Add the causal broadcast algorithm from the previous exercise sheet to the project. When starting the broadcast, make it use the distributed Erlang link layer, which you can start with `{ok, LL} = link_layer_distr_erl:start_link(microdote)`. The `link_layer_distr_erl` module uses the environment variable `MICRODOTE_NODES` to find the other nodes in the cluster.

2. Create a module `microdote_server`, which implements a `gen_server`. This process keeps the state for each database entry in memory and handles requests for reading and updating objects.

   - To update an object, the server needs to retrieve the current state of the object. If the object does not exist, the initial state is created with `antidote_crdt:new`. Next, the downstream effect for the update is calculated with `antidote_crdt:downstream`. This downstream effect must be applied locally and at all other servers with `antidote_crdt:update`. To transfer the downstream effect to all servers, the causal broadcast algorithm is used.
   - For reading an object, the `antidote_crdt:value` function is used.
   - To handle the session guarantees, each server keeps a vector clock to sum-

marize its causal history. This clock has to be updated for each update-operation and when updates from remote nodes are received.

If a request with a clock that is not lower than the current local clock comes in, the server has to wait until the necessary updates arrive. To implement the waiting, put the request into a set of waiting requests and use the option of the `gen_server` module to return `{noreply, NewState}`. The server can then reply at a later point in time (when the local clock has advanced) using `gen_server:reply`.

Note that you cannot use `timer/sleep` to implement the waiting, since this would block the server and prevent it from receiving other messages – in particular the messages that would bring in the operations it is waiting for.

3. Add the `microdote_server` as a child of the `microdote_sup` supervisor to make it start when the application starts.

   Register the server using a local name (this is an option of `gen_server:start_link`).

4. In the `microdote` module, you can now implement the API functions by forwarding the requests to the `microdote_server`. Note that `gen_server:call` can directly use a locally registered name as the receiver of the message.