Prof. Dr. Annette Bieniusa
Philipp Lersch, M. Sc.

RPTU Kaiserslautern-Landau
Fachbereich Informatik
AG Softwaretechnik

Exercise Project: Programming Distributed Systems (Summer 2025)

Deadline: Wednesday 06.08.25 18:00

This exercise sheet will be your final project for this course. Successfully implementing this project is a **requirement for being admitted to the exams**. The project must be completed individually by each participant. Teamwork is not allowed.

You have to implement the basic functionality for this project (Section 1) and one of the feature extensions in Section 2.

Submit your code through your project's Git repository before Wednesday on August 6th, 18:00. Your repository is localted under
`https://softech-git.informatik.uni-kl.de/students/pds/ss25/project/<Acc-handle>`
Assign Philipp Lersch to a merge request when your project is ready to be reviewed.

You can find a template for the project in the material repository under
`https://softech-git.informatik.uni-kl.de/students/pds/ss25/materials/`.
Feedback and answers to questions can be requested **until the week before the provided deadline**. Afterwards, messages might not be answered in time.

Homework policy Programming is a creative process. Individuals must reach their own understanding of problems and discover paths to their solutions. During this time, discussions with friends and colleagues are encouraged, and they must be acknowledged when you submit your work. When the time comes to write code, however, such discussions are no longer appropriate. Each handed in solution must be entirely your own work!

Do not, permit any student to copy any part of your solution, and do not copy any part of another person's solution. In particular, you may not test or debug another participant's code, nor may you have someone test or debug your code. If you can't get code to work, consult the teaching assistant! You may look in the library (including the internet, etc.) for ideas on how to solve problems, just as you may discuss problems with your classmates. All sources must be acknowledged. The standard penalty for violating these rules in the assignment is to not pass this project.

(The above policies were adapted from policies used by Norman Ramsey at Purdue University in Spring 1996.)

AI policy You may use any AI tool of your choice, but you must acknowledge the tool(s) used. The correct place for such acknowledgements is the provided `README.md` file within the template. Furthermore, you are responsible for anything you submit.

1 Final Project: A causally consistent CRDT database

For the final project, you will develop a replicated data store named "Minidote"[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 data store 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: Minidote is a key-CRDT store: Each replicated data object is stored under a key. The store provides an API to update objects and read the current state of an object assigned to a key. 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.

You will implement your own CRDTs as Elixir behaviours[2]. Reuse of existing solutions from exercise sheet 6 is possible, but the code has to be changed to work with the behaviour pattern. The Antidote CRDT library[3] can be referenced for additional clues.

API: The base project provides an Elixir API to connect clients. The details of this API are explained below in Section 1.1.

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.

  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 object 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 $e_1 \xrightarrow{vis} e_2$.

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 add smaller component tests for the code you write.

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

[2] Take a look at the documentation for behaviours `https://hexdocs.pm/elixir/1.18.4/typespecs.html#behaviours`

[3] `https://github.com/AntidoteDB/antidote/tree/master/apps/antidote_crdt`

Documentation and Code style We expect you to write clean and readable code with comments for documentation. Additionally, add type specifications with meaningful type definitions. Documentation that concerns the complete project is best placed within the provided `README.md` file.

1.1 The Minidote API

Module name: `Minidote`

The Minidote module provides 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`).

```
key() :: {
  Key :: binary(),
  Type :: CRDT.t(),
  Bucket :: binary()
}
```

Note that in Elixir, strings are binaries. Some valid examples of keys are:

```
Key1 = {<<"key">>, :counter_pn_ob, <<"counter_test_local">>}
Key2 = {"Some other key", :set_aw_op, "Some other bucket"}
```

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()}.
def read_objects(objects, clock) do ...
```

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(), atom(), any()}], clock()) ::
    {:ok, clock()}
  | {:error, any()}.
def update_objects(updates, clock) do ...
```

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). You can then call the API methods within the shell:

```
iex(minidote1@127.0.0.1)1> {:ok, clock} = Minidote.update_objects([{{"K", :
set_aw_op, "V"}, :increment, 42}], :ignore).
```

```
    {:ok,%{"minidote1@127.0.0.1":  1}}
    iex(minidote1@127.0.0.1)2> Minidote.read_objects([{"K", ::set_aw_op, "V"}],
clock).
    {:ok,[{{"K",:antidote_crdt_counter_pn,"V"},42}],%{"minidote1@127.0.0.1":  1}}
```

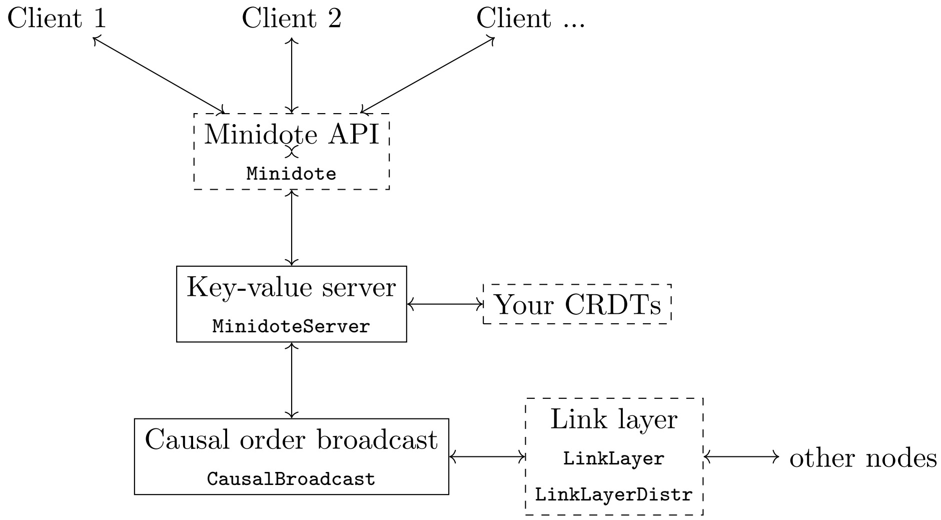
1.2 Architecture

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

1. The *CRDT* module class and one and a half behaviour.
2. The *LinkLayer* module to connect the individual nodes.
3. Some tests that are in dire need of being improved.

You have to implement the API in the `Minidote` module (see 1.1), for which you will probably need to implement further modules.

You are free to change as much of the template as you like and come up with your own (distributed) architecture to implement the system. One possible architecture is sketched below:



To implement Minidote with this architecture, you can follow these steps:

1. Add the causal broadcast algorithm from the lecture to the project. When starting the broadcast, make it use the distributed Elixir link layer. The `LinkLayer` module uses the provided group name to find other nodes in the cluster with the same group name.
2. Create a module `MinidoteServer`, which implements a `GenServer`. 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 `CRDT.new`. Next, the downstream effect for the update is calculated with `CRDT.downstream` . This downstream effect must be applied locally and at all other servers with `CRDT.update`. To transfer the downstream effect to all servers, the causal broadcast algorithm is used.
   - For reading an object, the `CRDT.value` function is used.
   - To handle the session guarantees, each server keeps a vector clock to summarize 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 `GenServer` module to return `{:noreply, new_state}`. The server can then reply at a later point in time (when the local clock has advanced) using `GenServer.reply`.

Note that you cannot use `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 `MinidoteServer` as a child of the `MinidoteSup` supervisor to make it start when the application starts.

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

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

1.3 CRDTs

Implement all five CRDTs from exercise 6 including the state-based counter as beheviours. Examples are provided with `Counter_PN_OB` and `Set_AW_OP`. Additionally, add at least one more CRDT. These additional CRDTs should provide new and meaningful semantics like add, delete or enable wins. Re-implementing the same type with similar semantics but slightly different internals is not an option.

If you are short on ideas you can take a look into the documentation in `crdt.ex`. Additionally, map and sequence CRDTs(e.g. for text or lists) might be more complex but also interesting. Please provide an additional explanation at the top of each implementation within the `@moduledoc` on the desired semantics of each CRDT, if it is state- or operation-based, how it is internally structured.

The CRDT data flow pattern should follow the same concept of antidote crdts, where each operation first creates an downstream effect without modifying the CRDTs state as described in Item 2. The created downstream effect is subsequently applied through an update function that updates the state of a local replica. Reading the CRDTs value must also be possible.

2 Additional Features

The in Section 1 described foundational data store leaves room for useful improvements. A few possible improvements are listed as features below. Choose one of them and integrate it into your solution.

2.1 Feature: Crash recovery and log pruning

Add support for efficiently restarting the system after a crash, without loosing any database state.

This can be done by writing all update operations to a persistent log. However, over time this log will grow and restarting can become very slow. To avoid this problem, you can write the state of objects to disk as well. Once all data store nodes know about an update and the corresponding state has been written to disk, the corresponding log entries can be pruned.

Hints: You can use the Erlangs disk-base term storage (DETS) and Disk Log to store data to disk. Alternatively, you can also use Elixir bindings for external libraries like LevelDB or RocksDB.

2.2 Feature: Dynamic Membership

Add support for dynamically adding and removing servers from the cluster.

Hint: The `link_layer_distr.ex` module uses Erlang process groups to handle cluster membership. You can adapt this module to support adding and removing new members.

You also need to make sure that a new server can get up to date with current state of the other servers. If you implemented robustness against lost messages (Task **??**), this should already work without any additional changes.

Your implementation should also handle the case, that a server is removed from the system and later a new server joins with the same node name.

2.3 Feature: Strong Consistency

Add support for strong consistency: It should be possible to perform read- and update operations with sequential consistency guarantees.

To do this you can adapt the API in `minidote.erl`.

If you want to keep the API unchanged, you can simply use the convention that all reads and updates are performed with strong consistency if the bucket name starts with `"sc_"`.

Hint: You can use a Raft library like rabbitmq/ra to implement strong consistency using replicated state machines.

2.4 Feature: HTTP API

Add support for accessing and managing the data store via an HTTP interface instead of using distributed Elixir. The link layer should be replaced with the appropriate functionality. *Hint:* You can use `ranch` for socket management and `JSON`, `Protocol Buffer` or something similar as the serialization interface between nodes.