

Exercise Project: Programming Distributed Systems (Summer 2024)

Deadline: Friday 26.07.24 AoE

This exercise sheet will be your final project for this course. Successfully implementing this project is a **requirement for being admitted to the exams**.

You have to solve and submit this task individually or as a team, together with up to two other students. If you work in a group it must be visible for us, that every group member worked on the code (e.g. in the Git log).

You have to implement the basic functionality for this project (Task 1) and must choose at least N of the feature extensions (Tasks 3-6), where N is the number of people in your team.

Submit your code via your Git repository before Friday, July 26, timezone Anywhere on Earth. Either assign us to a merge request or notify us via mail when your project is ready to be reviewed and tested. We can **also give you feedback on work in progress**.

You can find a template for the project in the shared repository under <https://softech-git.informatik.uni-kl.de/students/pds/ss24/materials/>.

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 written work. When the time comes to write code, however, such discussions are no longer appropriate. Each program must be entirely your own group's work!

Do not, under any circumstances, permit any student from another group to see any part of your program, and do not permit yourself to see any part of another group's program. In particular, you may not test or debug another group's code, nor may you have someone from another group 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 homework 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 exercise.

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

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 database must be highly available


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

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 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 library² 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: The base projects provides an Elixir API to connecting clients. The details of this API are explained below in 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 (Hint: the CRDT library ensures correct return values, if you use it correctly).

For example using a multi-value register guarantees:

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

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 $e_1 \xrightarrow{vis} e_2$.

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.

1.1 The Minidote API

Module name:  `minidote`

²https://github.com/AntidoteDB/antidote/tree/master/apps/antidote_crdt

The Minidote 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`).

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

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

```
Key1 = {<<"key">>, :antidote_crdt_counter_pn, <<"counter_test_local">>}  
Key2 = {"Some other key", :antidote_crdt_register_mv, "counter_test_local"}
```

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). On the shell you can then call the API methods:

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

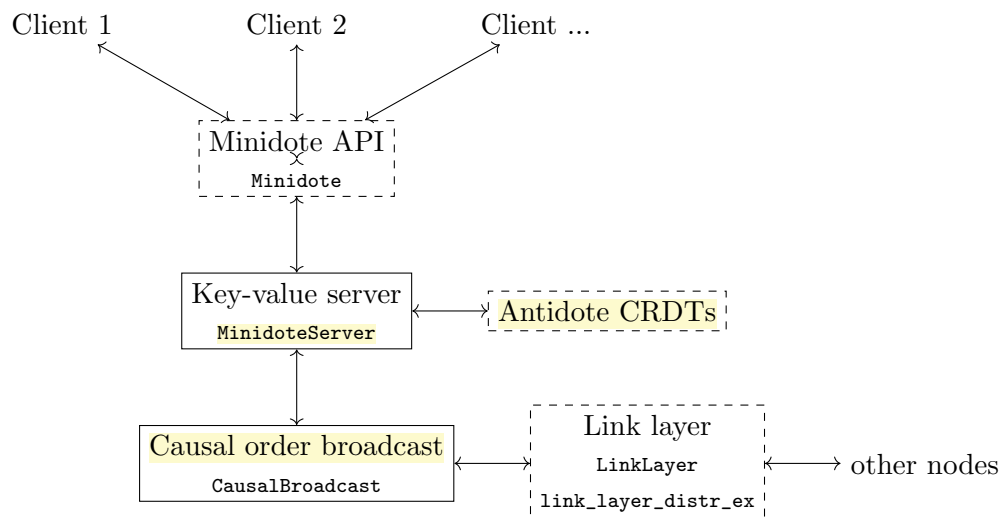
1.2 Architecture

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

1. The `Antidote` CRDT library.
2. The `link layer` you know from previous exercises.

You have to implement the API in the `Minidote` module (see 1.1), 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 Minidote with this architecture you can follow these steps:

1. Add the `causal broadcast algorithm` from `exercise sheet 3` to the project. When starting the broadcast, make it use the distributed Elixir link layer. The `link_layer_distr_erl` module uses the environment variable `MINIDOTE_NODES` to find the other nodes in the cluster.
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 `: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 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. 

2 Extension: HTTP Api



Add support for accessing and managing the database via a 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 data centers.

3 Extension: Robustness



The causal broadcast algorithm from the lecture assumed a perfect link model and the Crash-Fault model for failures. However in practice these guarantees are not met:

- Messages in `distributed Elixir` do not guarantee reliable delivery. From the Erlang language specification:

An implementation should ensure that whenever possible, a signal dispatched to a process should eventually arrive at it. There are situations when it is not reasonable to require that all signals arrive at their destination, in particular when a signal is sent to a process on a different node and communication between the nodes is temporarily lost.

- When a node crashes we want to be able to restart it and continue working.

Make your `Minidote` implementation robust against these kind of failures. More precisely your system should guarantee `eventual visibility` even when messages are lost and it should provide the following durability guarantee:

Durability: After an update operation returns, `the value must be guaranteed to be persistently stored on at least one machine`. The update must not be lost when the machine crashes and the database restarts later. `Read operations may not return results, which are not yet persistently stored`.

4 Extension: Crash recovery and log pruning

Add support for efficiently restarting the system after a crash, without losing 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 database servers 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.

5 Extension: 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_"`.

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

6 Extension: Dynamic Membership

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

Hints: 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 3), 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.