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

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Exercise Project: Programming Distributed Systems (SS 2018)

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 one other student. 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).

Submit your code via your Git repository before Wednesday, July 11, 23:59. Either create 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/progdist18/progdist_material/tree/master/code/minidote_template`.

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 and provide low latency, so every replica should be able to handle requests, even if it is disconnected from others.

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

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 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 `minidote_server` module, which you have to implement. 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 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 $e_1 \xrightarrow{vis} e_2$.

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.

Fault model We assume a non-byzantine fault model. Messages might be lost or delayed. Nodes may crash and restart after some time.

In addition, the system should be able to handle unpredictable errors – if a single Erlang process crashes, supervisors should restart the relevant part of the system.

[2] `https://github.com/SyncFree/antidote_crdt`

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

Performance requirements We are not trying to implement the world's fastest data store here. It is more important to be correct than to be fast. However, you should try to achieve a decent performance and try to avoid strongly degrading performance if the system keeps running for a long time.

For the following performance requirements, we assume a system with 3 machines, each running one replica of the database. Each machine runs Linux, an Intel Core i5-4310M CPU, and has 8GB of RAM. The connection between nodes has an average latency below 50ms and allows for a throughput of at least 10 Mbit/s.

Staleness: Under normal operation, updates should be delivered in a timely manner: If all nodes are connected, latency between nodes is below 100ms, and there are less than 10 updates per second with less than 1kB of data per update, then other updates should become visible at other nodes within 5 seconds.

After a network problem (here: a node got disconnected for 10 seconds), updates should become visible within 30 seconds after the network connection has been restored to normal operation.

Latency: Assume a setup where 10 threads per node sequentially issue operations (50% read, 50% writes) on counter objects. The keys are chosen randomly with a power-law probability distribution[4].

The average latency for write-operations should be less than 100ms and 95% of write-operations should have a latency below 200ms.

For read-operations the average latency should be less than 50ms.

Throughput: For the same setup as above but with 100 threads, the system should be able to perform at least 500 operations per second.

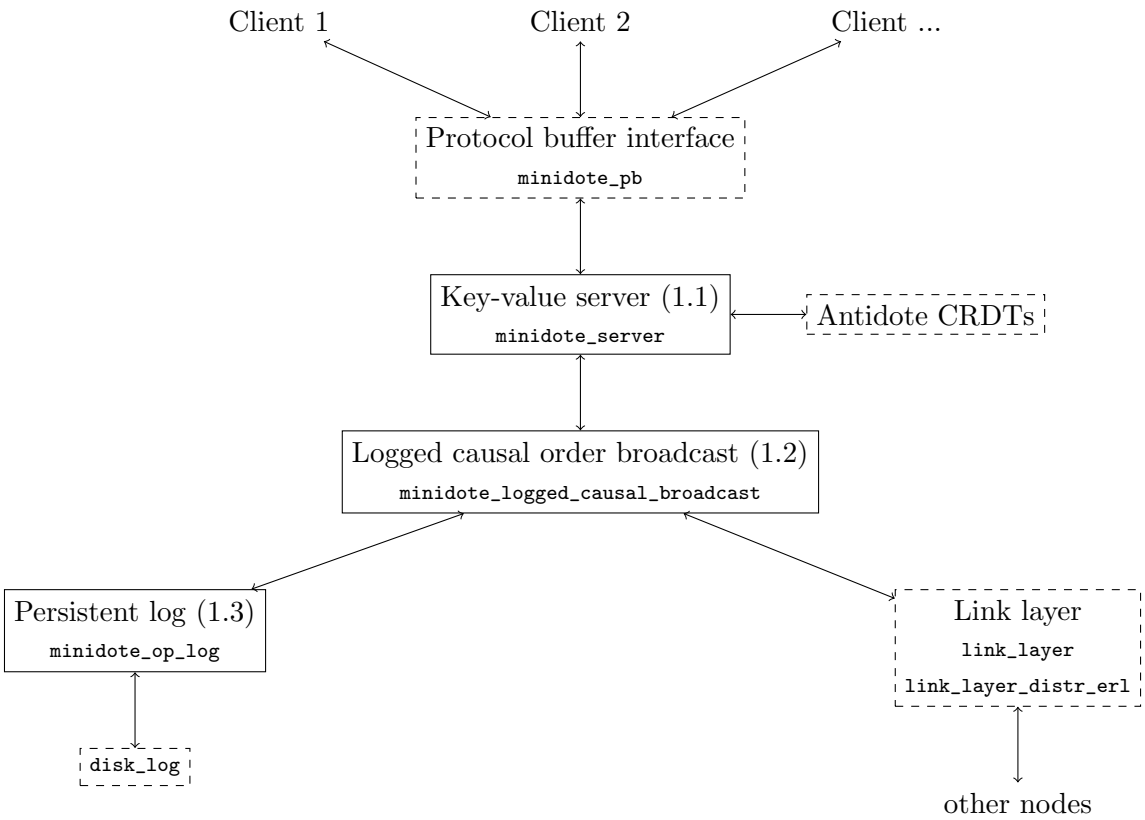
Testing In the initial template you will find only some very basic tests. Over time, we will make more tests and benchmarks available.

To use our unit tests, you need to stick to the architecture described below.

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

Architecture There are three main components, which you need to implement. In the overview below, these components are marked with solid borders. Components for which we provide libraries are marked with a dashed border.

[4] more precisely: a pareto distribution, where keys 0 to 1000 are chosen 80% of the time.



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.

1.1 Key-value server

Module name: `minidote_server`

The key-value server holds the state of all objects in memory under their respective key. A key is a 3-tuple consisting of a main identifier (`Key`), the type of the datatype (`Type`), and a namespace (`Bucket`).

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

The key-value server should provide the following API described below. You can choose an arbitrary representation for the type `clock()`.

```
-type server() :: pid() | atom().

% Starts the key-value server and returns the process identifier of the server.
% The process will be registered locally under the given ServerName.
% During initialization the minidote_logged_causal_broadcast is instantiated.
% Before the server can return any other requests, it recovers from the log.
% It will receive broadcast messages from the broadcast module and a
% log_recovery_done message once all broadcast messages from the log
% have been sent.
-spec start_link(atom()) -> {ok, server()} | ignore | {error, Error :: any()}.
start_link(ServerName) -> ...

% Takes a list of keys and returns the value of the corresponding objects.
% To get the value of a CRDT it uses the antidote_crdt:value function.
% 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 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(server(), [key()], clock() | ignore) ->
    {ok, [any()], clock()}
  | {error, any()}.
read_objects(Server, Objects, Clock) -> ...

% Takes a list of {key, update} pairs 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 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(server(), [{key(), Op :: atom(), Args :: any()}], clock()) ->
    {ok, clock()}
  | {error, any()}.
update_objects(Server, Updates, Clock) -> ...
```

1.2 Logged causal order broadcast

Module name: `minidote_logged_causal_broadcast`

This module provides functionality similar to the causal broadcast algorithm you implemented for Exercise 2. In addition to the features you implemented back then, the logged causal order broadcast module should be able to handle crashes. To this end, each message that is broadcast is stored in a log (using the persistent log module, see 1.3). On startup, this log is read and used to redeliver all messages from the log to all nodes (including the sender). When all messages from the log have been delivered, a `log_recovery_done` message is sent.

The module uses the `link_layer` and `link_layer_distr_erl` modules to communicate with other nodes. Another minor optimization is that the broadcast function only delivers the message to other nodes and not to the sender (the sender only receives the messages during recovery).

```
% Starts a process handling the broadcast 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 is a process-id.
% When delivering a broadcast message Msg, the tuple {rco_deliver, Msg}
% should be sent to this process.
% The second argument is a ServerName that is passed to the log modules
% start_link function.
% After initialization, recovery from the logs starts automatically.
% Each message in the log is sent to the RespondTo process.
% When recovery is complete, a message log_recovery_done is sent to the
% RespondTo process.
-spec start_link(pid(), atom()) -> {ok, pid()} | ignore | {error, Error :: any()
    }.
start_link(RespondTo, ServerName) -> ...

% broadcasts a message to all other nodes in the system.
% The first argument is the process-id returned by start_link.
% The second argument is the message to send.
% When the function returns 'ok' it is guaranteed, that the
% message has been persistently stored to disk.
-spec broadcast(pid(), any()) -> ok.
broadcast(B, Msg) -> ...

% Stops the broadcast process.
-spec stop(pid()) -> any().
stop(B) -> ...

% Returns the name of the current node.
-spec this_node(pid()) -> any().
this_node(B) -> ...
```

1.3 Persistent Log

Module name: `minidote_op_log`

This module implements persistent storage of a log. The files created by this module should be stored in a folder that can be configured with the environment variable `OP_LOG_DIR`, which should default to `"data/op_log/"`.

The log keeps a sequence of log entries for each node in the system. Each log entry consists of an index number and some data. For each node, the logged entries are indexed consecutively starting from 1.

```
-type log_entry() :: {Index :: pos_integer(), Data :: any()}.

% Starts the server
% The first argument is a name for the server.
% Any files opened by this module must include the ServerName in the file path
% to ensure that we can run local tests without name clashes.
% The second argument is the process, which receives recovery messages.
% After starting, log recovery starts automatically.
% For each entry in the log, a message {log_recovery, Node, {Index, Data}}
% is sent to the RecoveryReceiver and when recovery is finished a message
% 'log_recovery_done' is sent.
-spec start_link(atom(), pid()) -> {ok, pid()} | ignore | {error, Error :: any()}.
start_link(ServerName, RecoveryReceiver) -> ...

% Add a log entry to the end of the log.
% Server is the process returned by start_link.
% Node is the node() where the message originally came from.
% Entry is the log entry to store, consisting of index and data.
% When the function returns 'ok' the entry is guaranteed to be persistently stored.
% When an entry with the same index already exists for the given node,
% an error is returned. If the previous index does not exist, the
% call waits for it to be added.
-spec add_log_entry(server(), node(), log_entry()) -> ok  | {error, Reason :: term()}.
add_log_entry(Server, Node, Entry) -> ...

% Read all log entries belonging to a given node and in a certain range.
% The function works similar to lists:foldl for reading the entries.
% Server is the process returned by start_link.
% Node is the node() where the message originally came from.
% FirstIndex is the first index to read.
% LastIndex is the last index to read (or 'all' for reading all entries).
% F is the fold function, which takes a single log entry and the current
% accumulator and returns the new accumulator value.
% Acc is the initial accumulator value.
% Returns the accumulator value after reading all matching log entries.
-spec read_log_entries(server(), node(), integer(), integer() | all,
    fun((log_entry(), Acc) -> Acc), Acc) -> {ok, Acc}.
read_log_entries(Server, Node, FirstIndex, LastIndex, F, Acc) -> ...
```

Hint: You can use Erlang's `disk_log` module for implementing the persistent log. The `sync` function can be used to ensure data is persistently stored on disk. However, it can be slow, so you might not want to call it for every single call of `add_log_entry`. If there are concurrent requests you can use a single call to `sync` for all of them.