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

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Exercise 3: Programming Distributed Systems (SS 2018)

- To get feedback to your solution: Create a new branch in your git repository (e.g. `git checkout -b ex3`). Submit your solution to the programming exercise via your group's repository to the new branch in a folder named "ex3". 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.
- Prepare this sheet for the exercise on Thursday, May 17.

1 Atomic broadcast and consensus

In the lecture you have seen, that atomic broadcast can be implemented using consensus (Atomic Broadcast Algorithm, Lecture 4, Slide 19).

a) Show that it is also possible to do the reverse: Describe an algorithm that implements consensus by using atomic broadcast. Your algorithm should provide a *propose* method to clients and trigger a *decide*-event when a proposed value is decided.

```
State:
  // We only want to decide on one value, so remember if we have already
    decided
  Done ← false

Upon propose(Msg) do
  trigger a-Broadcast(Msg)

Upon a-Deliver(Msg) do
  if not Done
    Done ← true
    trigger decide(Msg)
```

b) Assume we change the atomic broadcast algorithm from the lecture and only use best-effort broadcast instead of reliable broadcast. Show that this would make the algorithm incorrect.

- 3 Processes
- P1: a-broadcast(M)
- be broadcast delivers M on P1 and P2
- P1 crashes, be broadcast fails to deliver M to P3
- P3 never participates in consensus and does not learn about M
- (Still, it might catch up if other messages trigger participation in consensus)

c) Consider the following modification to the atomic broadcast algorithm from the lecture. Why does this "optimization" not work correctly?

```
State:
  k_p              // consensus number
  pending     // messages received but not a-delivered by process

Upon Init do:
  k_p ← 0;
```

```
  pending ← ∅;
Upon a-Broadcast(m) do
  trigger rb-Broadcast(m);
Upon rb-Deliver(m) do
  pending ← pending ∪ {m};

Upon pending ≠ ∅ do
  k_p ← k_p + 1;
  propose(k_p, pending);
  wait until decide(k_p, msg^{k_p})
  ∀ m in msg^{k_p} in deterministic order do trigger a-Deliver(m)
  pending ← pending \ msg^{k_p}
```

A message could be delivered twice (once via propose from P1, once proposed from P2 if P2 rb-delivers the message after the decide from P1)

2 Optimizing causal broadcast

The causal broadcast algorithm from the lecture, which you implemented for the last exercise sheet, reuses the reliable broadcast algorithm. This is a nice and modular solution, however it has some performance issues:

- Without crashes, the number of messages sent by our reliable broadcast implementation is N^2 if N is the number of processes.
- The reliable broadcast algorithm keeps a set, which grows with every delivered message.

Design an optimized algorithm, which does not have these performance issues.

You don't have to implement your optimized version of the algorithm. However, if you do implement it, be aware that the tests we provided for the last exercise sheet were specific for that algorithm and might not work for algorithms that use a different strategy.

Ideas:

- Use vectorclock instead of set to know which messages have already been delivered.
- Don't send received messages to all other nodes. Instead, store messages until they are received at all nodes and let nodes request missing messages once they detect that something is missing.
- To detect missing messages, periodic heartbeat messages including the current vectorclock can be used.

3 Practical exercise: state machine replication

In this exercise you will implement an online gambling service "Repbet", where users can bet money on the outcome of certain events.

Since gamblers can get pretty angry, when their favorite game is not available, we want to use replication for high availability. We also want high consistency for dealing with money and aggressive gamblers and therefore will use replicated state machines with the Raft algorithm.

The template you can download for this task already contains the basic setup. Check the Readme file for instructions on building the project. The module `repbet_machine` contains the implementation of the replicated state machine and stubs for the functions you need to implement to solve this task. The comments in this file explain how to use and adapt the replicated state machine. We use the library "ra"[1] as Raft library.

Implement the following functions:

```
%% Creates a new user with the given Name, Mail, and Password.
%% Returns 'ok' when the user was created and {error, name_taken} if the
%% user name is already taken.
create_user(Node, Name, Mail, Password) -> ...

%% Checks if there is a user with the given name and password combination
%% Returns ok if password matches and {error, authentication_failed} otherwise
check_password(Node, Username, Password) -> ...

%% Creates a new challenge with the given description that users can bet on.
```

[1] Source code is available at `https://github.com/rabbitmq/ra`. You can find some documentation at `https://rabbitmq.github.io/ra/`. The interesting parts are in the description of the `ra` and `ra_machine` modules.

```
%% Each challenge gets assigned a new unique id, which is returned
%% as {ok, ChallengeId}.
create_challenge(Node, Description) -> ...

%% Makes the given user bet on a given challenge.
%% Result is either true or false (true if the user bets, that the challenge
%% turns out to be true, false otherwise)
%% Money is the amount of money the user bets on this challenge.
%% This amount is directly subtracted from the users balance.
%% Returns ok if betting was successful
%% Returns {error, amount_must_be_positive}, if Money is <= 0
%% Returns {error, invalid_user} or {error, invalid_challenge} if user
%% of challenge does not exist
%% Returns {error, insufficient_funds} if the users account balance is
%% less than the Money he wants to bet
bet(Node, Username, ChallengeId, Result, Money) -> ...

%% Changes the balance of the given user by the given Amount
%% Amount can be positive or negative, but the user account must
%% always have positive value
%% Returns ok on success and {error, insufficient_funds}, if changing
%% the balance would go below 0
change_balance(Node, Username, Amount) -> ...

%% Completes the given challenge by setting a Result
%% Result is either true or false
%% All Money bet on this challenge is distributed among the users
%% that guesses the result correctly,
%% proportionally to the amount they have bet.
complete_challenge(Node, ChallengeId, Result) -> ...

%% Get a list of all challenges, that have not yet been completed
%% Returns a list of tuples: {Id, Description, TotalMoney, YesPercentage}
%% Id is the Id of the challenge
%% Description is its description
%% TotalMoney is the overall amount invested in this challenge
%% YesPercentage is the percentage of money that has been set on 'yes' (true)
list_active_challenges(Node) -> ...

%% Get a list of all challenges, that the given user has participated in
%% Returns a list of tuples:
%% {Id, Description, Result, UserGuess, UserMoney, TotalMoney, YesPercentage}
%% Id is the Id of the challenge
%% Description is its description
%% Result is either 'true' or 'false' for completed challenges
%% or 'not_completed' otherwise
%% TotalMoney is the overall amount invested in this challenge
%% YesPercentage is the percentage of money that has been set on 'yes' (true)
%% UserGuess is the result guessed by the user in his bet
%% UserMoney is the amount of money invested by the user
list_user_challenges(Node, Username) -> ...

%% Get the current balance of a user
%% Returns {ok, Balance} or {error, user_not_found}
get_balance(Node, Username) -> ...
```