PCPP **IT University of Copenhagen, F2024**

Exercises week 12

Last update: 2024/11/17

Goal of the exercises

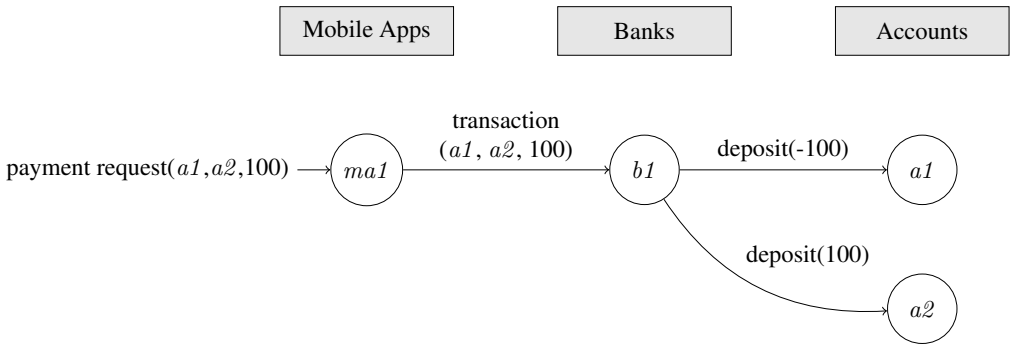
The goals of this week's exercises are:

- Design an Actor system.
- Design the communication protocol between actors.
- Implement the design in Erlang.

Exercise 12.1 Your task in this exercise is to implement a Mobile Payment system using Erlang. The system consists of three types of actors:

- *Mobile App*: These actors send transactions to the bank corresponding to mobile payments. These transactions are started by a user of the mobile app by making a payment request.
- *Bank*: These actors are responsible for executing the transactions received from the mobile app. That is, substracting the money from the payer account and adding it to the payee account.
- *Account*: This actor type models a single bank account. It contains the balance of the account. Also, banks should be able to send positive deposits and negative deposits (withdrawals) in the account.

The directory `code-exercises/week12exercises` contains a source code skeleton for the system that you might find helpful.



The figure above shows an example of the behavior. In this example, there is a mobile app, $ma1$, a bank, $b1$, and two accounts, $a1$ and $a2$. The arrow from $ma1$ to $b1$ models $ma1$ sending a transaction to $b1$ indicating to transfer 100 DKK from $a1$ to $a2$. Likewise, the arrows from $b1$ to $a1$ and $a2$ model the sending of deposits to the corresponding accounts to realise the transaction—the negative deposit can be seen as a withdrawal.

The description above sets high level requirements for the behavior of this actor system. It (purposely) does not mention concrete implementation details. You have freedom in choosing how to design/implement the state, behavior and communication protocols for these actors; as long as they comply with the high level requirements.

Mandatory

1. Design and implement the Mobile App actor (see the file `mobile_app.erl` in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state (if any), how it is initialized, purpose of messages, etc.

Note: Recall that you may interact with actors (processes) directly from the shell (`erl`). You can spawn an actor from the shell—e.g., a mobile app actor—and send it messages to observe its behavior. This note applies to exercises below where you are asked to design and implement an actor.

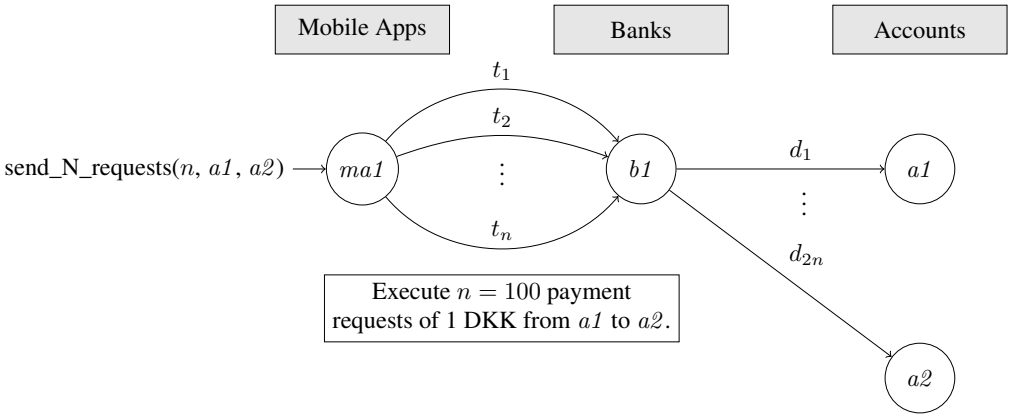
2. Design and implement the Bank actor (see the file `bank.erl` in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state (if any), how it is initialized, purpose of messages, etc.

3. Design and implement the Account actor (see the file `account.erl` in the skeleton), including the messages it must handle. Explain your design choices, e.g., elements of the state (if any), how it is initialized, purpose of messages, etc.

4. Write a function, `system_start2`, that starts 2 mobile apps, 2 banks, and 2 accounts. After executing the function, it must be possible to make payment requests via the mobile app actors. You may directly provide the code of the function so that it can be executed from the shell (`erl`), or you can create a special module containing the function; you may name this module `system_starter.erl`.

   Additionally, write another function that performs two payment requests: 1) from *a1* to *a2* via *ma1*, and 2) from *a2* to *a1* via *ma2*. The amount in these payments is a constant of your choice.

5. Extend the mobile app actor to include a function `send_N_requests` that sends n payment requests of 1 DKK between two accounts. It must be possible to call this function from the shell (`erl`) by issuing the command `> mobile_app:send_N_requests(...)` where `...` denotes is a list of parameters of your choice that specifies the number of requests n and the two accounts. The figure below illustrates the computation.



6. Extend the Account actor so that it can handle a message `print_balance`. Upon receiving this message the actor should print the current balance in the account.

   Additionally, write a function that makes 100 payment requests of 1 DKK between two accounts (using the `send_N_requests` you wrote in the previous exercise), and *afterwards* requests the two accounts involved in the requests to print their balance. As before, you may directly provide this function as code to be executed in the shell (`erl`), or you can include it in a module `system_utilities.erl`.

   Now run the function and answer the following questions:

   (a) What balance was printed?

   (b) Is that the only possible balance that could be printed?

   (c) What are the possible balances that could be printed after executing this function?

   Explain your answers.

7. Consider a case where two different bank actors send two deposits exactly at the same time to the same account actor. Can data races occur when updating the balance? Explain your answer.

Challenging

8. Modify the system above so that Account actors are associated with a bank, and ensure that negative deposits are only executed if they are sent by that bank. In other words, only the account's bank is allowed to withdraw money. Note that positive deposits may be received from any bank.
9. Modify the system so that an account's balance cannot be below 0. If an account actor receives a negative deposit that reduces the balance below 0, the deposit should be rejected. In such a case, the bank should be informed and the positive deposit for the payee should not be performed.