Weekly Assignment 4

Advanced Programming 2014 @ DIKU

Martin Jørgensen
University of Copenhagen
Department of Computer Science
tzk173@alumni.ku.dk

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

October 4, 2014

Abstract

...

Tasks

1 API methods

The API visible functions and the module name is declared in the two lines in 1.

```
-module(facein).
-export([start/1,add_friend/2,friends/1,broadcast/3,received_messages/1]).
```

Figure 1: Module name and API function exports. (../assignment/facein.erl)

Some of the API functions uses a helper function called `rpc`, the method is defined in 2. This function simple sends a message to a specified process, and posts the response the target process sends back.

```
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} -> Response
    end.
```

Figure 2: The RPC function. (../assignment/facein.erl)

1.1 `start`

```
start(N) -> spawn(fun() -> loop({N,[],[]}) end).
```

Figure 3: The `start` function. (../assignment/facein.erl)

Figure 3 shows our implementation of `start(N)`, it quite simply takes a name and starts the main loop function in a new thread. The loop gets started with a name, no friends and no messages.

1.2 `add_friend`

```
add_friend(P, F) ->
    rpc(F, {add, P}).
```

Figure 4: Text (../assignment/facein.erl)

Figure 4 shows the `add_friends` function, this function takes 2 PIDs as arguments, since we want to add Fs name to P's friendslist we chose to send a signal (`{add, P}`) to F instructing it do send it's name to P. Please read the section about the main loop to see how this is implemented. Because we use the rpc function we will wait for a response and return it to the caller.

1.3 friends

```erlang
friends(P) ->
    rpc(P, friends).
```

Figure 5: The friends implementation. (../assignment/facein.erl)

friends will use rpc to send a request to P via RPC. P respond with its friend list which friends will then return. Figure 10 shows the implementation of the function.

1.4 broadcast

Figure 6 shows our implementation of the broadcast function. Since broadcast do not wait for a response, we didn not use the rpc function and chose instead to send the message directly. As hinted by the assignment we tag each broad cast with a unique reference number, identifying messages among each other.

```erlang
broadcast(P, M, R) ->
    P ! {self(), {broadcast, make_ref(), P, M, R}}.
```

Figure 6: The broadcast implementation. (../assignment/facein.erl)

1.5 recieved_messages

```erlang
received_messages(P) ->
    rpc(P, messages).
```

Figure 7: The recieved_messages implementation. (../assignment/facein.erl)

Figure 7 show the implementation of recieved_messages, it uses rpc to send a request to a process and then returns the response.

2 Main loop

This section covers the main loop. Since this function is big and clearly segmented, we will cover it case for case. For the full implementation either consult the *facein.erl* file or see Figure 11.

loop takes a triple as argument, the triple contains the name of the person, the list of their friends and a list of messages the person have recieved. The loop will wait to recieve a message and then depending on pattern matching will performs actions as descriped in the following subsections.

2.1 Adding friends

Adding a friends is a 2 step process, as descriped in add_friend the person we want to add (F) to a friendslist (Ps friendlist), recieves a message with a pattern as shown in 8.

```
        % b) adds a friend
        {From, {add, P}} ->
            P ! {self(), {name, N}},
            receive
                {P, ok}                 -> From ! {self(), ok};
                {P, {error, Reason}}    -> From ! {self(), {error, Reason}}
            end,
            loop({N, L, MSG});
```

Figure 8: The pattern that catches the first step of a friend request. (../assignment/facein.erl)

When the process recieves the proper message it will send a message to P with it's own PID and name and then await a reply from P. The reply will the be forwarded back to the caller. In the end it will call itself (loop) with it's own name, friends and messages.

```
        {From, {name, F}} ->
            case lists:member({F, From}, L) of
                true    -> From ! {self(), {error, 'Already on friend list'}},
                           loop({N, L, MSG});
                false   -> From ! {self(), ok},
                           loop({N, [{F, From}|L], MSG})
            end;
```

Figure 9: Adding a friend to a friendlist and responding. (../assignment/facein.erl)

The second step is in P which matches a message with the pattern shown in Figure 9 it will check if F is already on P's friendlist, if it is it will send an error back, otherwise it will send an ok back and then call loop with it's name, it's friendlist with F appended and the message list.

2.2 Retrieving friends

```
        % c) retrives the friend list
        {From, friends} ->
            From ! {self(), L},
            loop({N, L, MSG});
```

Figure 10: Retrieving the friendlist and sending it back. (../assignment/facein.erl)

When a message matches the pattern seen in Figure 10 it will respond with a message containing it's ID a friend list before it restarts the `loop` method with the same arguments.

2.3 Broadcasting a message

2.4 Retrieving messages

2.5 Invalid message

3 Testing

A Full `loop` Implementation

Make it multipage?

```
loop({N, L, MSG}) ->
    %io:format('Person: ~w~nFriends: ~w~nMessages: ~w~n', [N, L, MSG]),
    receive
        % b) adds a friend
        {From, {add, P}} ->
            P ! {self(), {name, N}},
            receive
                {P, ok}                 -> From ! {self(), ok};
                {P, {error, Reason}}    -> From ! {self(), {error, Reason}}
            end,
            loop({N, L, MSG});

        {From, {name, F}} ->
            case lists:member({F, From}, L) of
                true    -> From ! {self(), {error, 'Already on friend list'}},
                           loop({N, L, MSG});
                false   -> From ! {self(), ok},
                           loop({N, [{F, From}|L], MSG})
            end;

        % c) retrives the friend list
        {From, friends} ->
            From ! {self(), L},
            loop({N, L, MSG});

        % d) broadcast a message M from person P within radius R
        {_, {broadcast, UID, P, M, 0}} ->
            self() ! {P, {message, UID, M}},
            loop({N, L, MSG});
        {_, {broadcast, UID, P, M, R}} ->
            self() ! {P, {message, UID, M}},
            case L of
                []  -> loop({N, L, MSG});
                L   -> pass_msg(UID, L, P, M, R-1),
                       loop({N, L, MSG})
            end;


        % adds a message, if it's not already added
        {From, {message, UID, M}} ->
            case lists:member({UID, From, M}, MSG) of
                true  -> loop({N, L, MSG});
                false -> loop({N, L, [{UID, From, M}|MSG]})
            end;

        % e) retrieves the received messages
        {From, messages} ->
            Messages = lists:map ( fun({_, F, M}) -> {F, M} end, MSG),
            From ! {self(), Messages},
            loop({N, L, MSG});

        % handle any other occurrences
        {From, Other} ->
            From ! {self(), {error, Other}},
            loop({N, L, MSG})
    end.
```