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 5, 2014

Abstract

For this assignment we are to write a person server in Erlang that implements a certain API for manipulating and interacting with other persons in a networked graph.

Tasks

1 API methods

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

```
6  -module(facein).
7  -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.

```
11 rpc(Pid, Request) ->
12     Pid ! {self(), Request},
13     receive
14         {Pid, Response} -> Response
15     end.
```

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

1.1 `start`

```
9  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`

```
17 add_friend(P, F) ->
18     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`

```
20 friends(P) ->
21     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.

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

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

1.5 `recieved_messages`

```
26 received_messages(P) ->
27     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 21.

`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.

```
39          % b) adds a friend
40          {From, {add, P}} ->
41              P ! {self(), {name, N}},
42              receive
43                  {P, ok}                 -> From ! {self(), ok};
44                  {P, {error, Reason}}    -> From ! {self(), {error, Reason}}
45              end,
46              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 its 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.

```
48          {From, {name, F}} ->
49              case lists:member({F, From}, L) of
50                  true    -> From ! {self(), {error, 'Already on friend list'}},
51                             loop({N, L, MSG});
52                  false   -> From ! {self(), ok},
53                             loop({N, [{F, From}|L], MSG})
54              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

```
56          % c) retrives the friend list
57          {From, friends} ->
58              From ! {self(), L},
59              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

The implementation of broadcasting out a message from P to all friends within radius R must be non-blocking. Therefore, we do not wait for it to receive any feedback from the message passing on lines 63 and 66. Our solution is based on decrementing the radius R as we propagate the message out to all immediate friends, using `pass_msg` (see figure 12) and then recurse with a decremented radius.

```
61          % d) broadcast a message M from person P within radius R
62          {_, {broadcast, UID, P, M, 0}} ->
63              self() ! {P, {message, UID, M}},
64              loop({N, L, MSG});
65          {_, {broadcast, UID, P, M, R}} ->
66              self() ! {P, {message, UID, M}},
67              case L of
68                  []  -> loop({N, L, MSG});
69                  L   -> pass_msg(UID, L, P, M, R-1),
70                         loop({N, L, MSG})
71              end;
```

Figure 11: Receiving broadcasts (../assignment/facein.erl)

As evident of the code above in figure 11, we have the zero radius base case on lines 62–64 upon which we simply message ourselves, as was required by the assignment text. If it is the case that $R > 0$ then we message ourselves and `pass_msg` is called with a decremented radius. This rule recurses on the given friendlist FS, sending out the broadcast signal for each one. Note at this time the radius has been decremented before the call, and so the terminates on the base case of $R = 0$.

```
29 pass_msg(UID, FS, P, M, R) ->
30     case FS of
31         [{_,F}|[]]  -> F ! {self(), {broadcast, UID, P, M, R}};
32         [{_,F}|T]   -> F ! {self(), {broadcast, UID, P, M, R}},
33                        pass_msg(UID, T, P, M, R)
34     end.
```

Figure 12: Propagates the received message to all in list FS (../assignment/facein.erl)

2.4 Retrieving messages

Upon receiving the `messages` signal we filter out the unique identifier associated with the messages in `MSG` on line 82, using the built-in `lists:map` function, taking an anonymous function that simply builds a list of tuples containing the sender and message, instead of the triple which contains the UID as well.

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

Figure 13: Retreives the messages (../assignment/facein.erl)

The rule then responds to the calling thread with this filtered list on line 83, and simply continues the loop execution.

2.5 Invalid message

Any message we do not have an explicit handler for is treated as an error, and is simply propagated backward to the request with an error token and what the message contains. Such occurrences do not stop the process, however, as we simply ignore it, and continue executing the loop on line 89.

```
86         % handle any other occurrences
87         {From, Other} ->
88             From ! {self(), {error, Other}},
89             loop({N, L, MSG})
```

Figure 14: Invalid message handling (../assignment/facein.erl)

3 Testing

3.1 Starting processes

In testing `start` function, we are expecting the Erlang shell to reply simply `ok`, indicating that nothing went wrong.

```
5  % start-up person processes
6  Andrzej = facein:start(andrzej).
7  Jen = facein:start(jen).
8  Jessica = facein:start(jessica).
9  Ken = facein:start(ken).
10 Reed = facein:start(reed).
11 Susan = facein:start(susan).
12 Tony = facein:start(tony).
```

Figure 15: Starting up all person processes in the graph G (../assignment/tests.erl)

Running `c(facein), file:eval('tests.erl')` on *only* the above section of the test code we get just that; `ok`.

3.2 Constructing the graph

Now that we have all person processes running, we can construct the network graph G.

```
14 % construct network graph
15 facein:add_friend(Andrzej, Ken).
16 facein:add_friend(Andrzej, Susan).
17
18 facein:add_friend(Jen, Jessica).
19 facein:add_friend(Jen, Susan).
20 facein:add_friend(Jen, Tony).
21
22 facein:add_friend(Jessica, Jen).
23
24 facein:add_friend(Ken, Andrzej).
25
26 facein:add_friend(Reed, Jessica).
27 facein:add_friend(Reed, Tony).
28
29 facein:add_friend(Susan, Andrzej).
30 facein:add_friend(Susan, Jen).
31 facein:add_friend(Susan, Jessica).
32 facein:add_friend(Susan, Reed).
```

Figure 16: Construction of the network graph G (../assignment/tests.erl)

We do so by using the `facein:add_friend` API (see section 1.2). Yet again, including the code from 15 in conjunction with the above, we are expecting to see just `ok` — and we do.

3.3 Friend lists

With a fully constructed network graph G, we can now begin testing for some meaningful output. The test code below queries every person process in the graph G for their friendlists, respectively, and formats the response using the built-in `io:format` API.

```
34 % friend list tests
35 io:format("Andrzej's friends: ~w~n", [facein:friends(Andrzej)]).
36 io:format("Jen's friends: ~w~n", [facein:friends(Jen)]).
37 io:format("Jessica's friends: ~w~n", [facein:friends(Jessica)]).
38 io:format("Ken's friends: ~w~n", [facein:friends(Ken)]).
39 io:format("Reed's friends: ~w~n", [facein:friends(Reed)]).
40 io:format("Susan's friends: ~w~n", [facein:friends(Susan)]).
41 io:format("Tony's friends: ~w~n", [facein:friends(Tony)]).
```

Figure 17: Querying the friendlist of person processes (../assignment/tests.erl)

We are expecting to see an output of a person name followed by the person's friendlist. And this is indeed what we get.

```
1 Andrzej's friends: [{susan,<0.71.0>},{ken,<0.69.0>}]
2 Jen's friends: [{tony,<0.72.0>},{susan,<0.71.0>},{jessica,<0.68.0>}]
3 Jessica's friends: [{jen,<0.67.0>}]
4 Ken's friends: [{andrzej,<0.66.0>}]
5 Reed's friends: [{tony,<0.72.0>},{jessica,<0.68.0>}]
6 Susan's friends: [{reed,<0.70.0>},{jessica,<0.68.0>},{jen,<0.67.0>},{andrzej
      ,<0.66.0>}]
7 Tony's friends: []
```

Figure 18: Output of running the code from figure 17 (friendlist.txt)

3.4 Broadcasting

In order to test the broadcasting system we wanted to depict a situation that allows us to show several different radii. For this, we created a rumour in graph G. The code of figure 19 broadcasts several messages and at many different radii, such we end up with a very diverse inbox environment. Instead of verifying each an every message we will highlight a few representative examples.

```
43 facein:broadcast(Ken, "Martin and Casper will probably get an A.", 1).
44 facein:broadcast(Andrzej, "Really? Do you think Martin and Casper should get
45 an A for the exam?", 1).
46 facein:broadcast(Ken, "Oh, maybe. But I meant for this assignment. It's good
       really good!", 1).
47
48 facein:broadcast(Susan, "I heard Martin and Casper are getting an A for the
49 exam, even though it's not even released yet!", 1).
50 facein:broadcast(Jen, "Say what!?", 1).
51 facein:broadcast(Jessica, "That's cheating!", 1).
52 facein:broadcast(Reed, "Are you kidding me?!", 1).
53
54 facein:broadcast(Andrzej, "Oh, man...", 1).
55 facein:broadcast(Ken, "What?", 1).
56 facein:broadcast(Andrzej, "Rumour has it you're giving them an A at the exam.",
       1).
57 facein:broadcast(Ken, "People of graph G! I have said no such thing!", 10).
58
59 facein:broadcast(Jessica, "Susan is a liar...", 1).
60 facein:broadcast(Jen, "Yeah, Susan cheated!", 2).
61
62 facein:broadcast(Andrzej, "I heard Susan has cheated!", 1).
63 facein:broadcast(Ken, "Really?", 1).
64 facein:broadcast(Andrzej, "Yeah!", 1).
65 facein:broadcast(Susan, "Aw man... :(", 0).
66
67 facein:broadcast(Tony, "Meh, I don't give a damn. Leave me be!", 10).
```

Figure 19: A rumour spreads throughout graph G (../assignment/tests.erl)

When we print out the resulting inboxes (see figure 20) we see that notibly no one gets Tony's message, even though its radius is very high. This shows that the broadcasting mechanism does indeed require the message to travel along the directed edges of the graph, as opposed to the outcry of Ken, which does propagate throughout the entire graph. Similarly, when Susan *whispers* to herself ($R = 0$) she does get her own message.

```
69 io:format("Andrzej's messages:~n~p~n", [facein:received_messages(Andrzej)]).
70 io:format("Jen's messages:~n~p~n", [facein:received_messages(Jen)]).
71 io:format("Jessica's messages:~n~p~n", [facein:received_messages(Jessica)]).
72 io:format("Ken's messages:~n~p~n", [facein:received_messages(Ken)]).
73 io:format("Reed's messages:~n~p~n", [facein:received_messages(Reed)]).
74 io:format("Susan's messages:~n~p~n", [facein:received_messages(Susan)]).
75 io:format("Tony's messages:~p~n", [facein:received_messages(Tony)]).
```

Figure 20: Prints every person's inbox (../assignment/tests.erl)

3.5 Test output

```
Erlang/OTP 17 [erts-6.2] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe]

Eshell V6.2  (abort with ^G)
1> c(facein), file:eval(tests).
{error,enoent}
2> c(facein), file:eval('tests.erl').
Andrzej's friends: [{susan,<0.51.0>},{ken,<0.49.0>}]
Jen's friends: [{tony,<0.52.0>},{susan,<0.51.0>},{jessica,<0.48.0>}]
Jessica's friends: [{jen,<0.47.0>}]
Ken's friends: [{andrzej,<0.46.0>}]
Reed's friends: [{tony,<0.52.0>},{jessica,<0.48.0>}]
Susan's friends: [{reed,<0.50.0>},{jessica,<0.48.0>},{jen,<0.47.0>},{andrzej,<0
Tony's friends: []
Andrzej's messages:
[{<0.46.0>,"Yeah!"},
 {<0.49.0>,"Really?"},
 {<0.46.0>,"I heard Susan has cheated!"},
 {<0.47.0>,"Yeah, Susan cheated!"},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.46.0>,"Rumour has it you're giving them an A at the exam."},
 {<0.49.0>,"What?"},
 {<0.46.0>,"Oh, man..."},
 {<0.51.0>,
  "I heard Martin and Casper are getting an A for the\nexam, even though it's n
 {<0.49.0>,
  "Oh, maybe. But I meant for this assignment. It's good really good!"},
 {<0.46.0>,
  "Really? Do you think Martin and Casper should get\nan A for the exam?"},
 {<0.49.0>,"Martin and Casper will probably get an A."}]
Jen's messages:
[{<0.47.0>,"Yeah, Susan cheated!"},
 {<0.48.0>,"Susan is a liar..."},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.48.0>,"That's cheating!"},
 {<0.47.0>,"Say what!?"},
 {<0.51.0>,
  "I heard Martin and Casper are getting an A for the\nexam, even though it's n
```

```
Jessica’s messages:
[{<0.47.0>,"Yeah, Susan cheated!"},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.48.0>,"Susan is a liar..."},
 {<0.50.0>,"Are you kidding me?!"},
 {<0.48.0>,"That’s cheating!"},
 {<0.47.0>,"Say what!?"},
 {<0.51.0>,
  "I heard Martin and Casper are getting an A for the\nexam, even though it’s n
Ken’s messages:
[{<0.46.0>,"Yeah!"},
 {<0.46.0>,"I heard Susan has cheated!"},
 {<0.49.0>,"Really?"},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.46.0>,"Rumour has it you’re giving them an A at the exam."},
 {<0.49.0>,"What?"},
 {<0.46.0>,"Oh, man..."},
 {<0.49.0>,
  "Oh, maybe. But I meant for this assignment. It’s good really good!"},
 {<0.46.0>,
  "Really? Do you think Martin and Casper should get\nan A for the exam?"},
 {<0.49.0>,"Martin and Casper will probably get an A."}]
Reed’s messages:
[{<0.47.0>,"Yeah, Susan cheated!"},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.50.0>,"Are you kidding me?!"},
 {<0.51.0>,
  "I heard Martin and Casper are getting an A for the\nexam, even though it’s n
Susan’s messages:
[{<0.46.0>,"Yeah!"},
 {<0.51.0>,"Aw man... :("},
 {<0.46.0>,"I heard Susan has cheated!"},
 {<0.47.0>,"Yeah, Susan cheated!"},
 {<0.49.0>,"People of graph G! I have said no such thing!"},
 {<0.46.0>,"Rumour has it you’re giving them an A at the exam."},
 {<0.46.0>,"Oh, man..."},
 {<0.47.0>,"Say what!?"},
 {<0.51.0>,
  "I heard Martin and Casper are getting an A for the\nexam, even though it’s n
```

```
 {<0.46.0>,
  "Really? Do you think Martin and Casper should get\nan A for the exam?"}]
Tony's messages:[{<0.47.0>,"Yeah, Susan cheated!"},
                 {<0.49.0>,"People of graph G! I have said no such thing!"},
                 {<0.50.0>,"Are you kidding me?!"},
                 {<0.47.0>,"Say what!?"}]
ok
```

A Full `loop` Implementation

Make it multipage?

```
36 loop({N, L, MSG}) ->
37     %io:format('Person: ~w~nFriends: ~w~nMessages: ~w~n', [N, L, MSG]),
38     receive
39         % b) adds a friend
40         {From, {add, P}} ->
41             P ! {self(), {name, N}},
42             receive
43                 {P, ok}                 -> From ! {self(), ok};
44                 {P, {error, Reason}}    -> From ! {self(), {error, Reason}}
45             end,
46             loop({N, L, MSG});
47
48         {From, {name, F}} ->
49             case lists:member({F, From}, L) of
50                 true    -> From ! {self(), {error, 'Already on friend list'}},
51                            loop({N, L, MSG});
52                 false   -> From ! {self(), ok},
53                            loop({N, [{F, From}|L], MSG})
54             end;
55
56         % c) retrives the friend list
57         {From, friends} ->
58             From ! {self(), L},
59             loop({N, L, MSG});
60
61         % d) broadcast a message M from person P within radius R
62         {_, {broadcast, UID, P, M, 0}} ->
63             self() ! {P, {message, UID, M}},
64             loop({N, L, MSG});
65         {_, {broadcast, UID, P, M, R}} ->
66             self() ! {P, {message, UID, M}},
67             case L of
68                 []  -> loop({N, L, MSG});
69                 L   -> pass_msg(UID, L, P, M, R-1),
70                        loop({N, L, MSG})
71             end;
72
73         % adds a message, if it's not already added
74         {From, {message, UID, M}} ->
75             case lists:member({UID, From, M}, MSG) of
76                 true  -> loop({N, L, MSG});
77                 false -> loop({N, L, [{UID, From, M}|MSG]})
78             end;
79
80         % e) retrieves the received messages
81         {From, messages} ->
82             Messages = lists:map ( fun({_, F, M}) -> {F, M} end, MSG),
83             From ! {self(), Messages},
84             loop({N, L, MSG});
85
86         % handle any other occurrences
87         {From, Other} ->
88             From ! {self(), {error, Other}},
89             loop({N, L, MSG})
90     end.
```