Assigments week 6

Joe Armstrong

January 31, 2014

Contents

1 Introduction

These problems are for lectures F6 and F7, held in week 6 of 2014.

Additional material can be found in Chapters 11 to 14 of the course book *Programming Erlang, 2'nd edition*.

2 Problems

This week I've put the problems into three categories:

- **normal** solve these to get *Godkänd*.
- **advanced** solve these to get *Väl Godkänd*.
- **optional** solve these to get a deeper understanding of the problem. You don't have to hand in your solutions to these, but solving them, or attempting them will deepen your understanding of Erlang.

The sections and subsections in this paper are marked appropriately.

3 Simple Processes

Write the following routines:

3.1 normal: `double:start()`

`double:start()` creates a registered process called double. If you send it a message containing an integer it should double the integer and print the result.

If it is sent a non-integer it should print an error message and crash.

Start the double process and test that it works correctly.

3.2 normal: `monitor:start()`

`monitor:start()` should create a processes that starts and then monitors the double process that you made in the last section.

If the monitor process detects that the double process has crashed it should print an error message and start a new double process.

Start the monitor process. Send the double process something that causes it to crash. Then check that a new double process has been created and that the system has recovered from the error.

4 normal: Reliable Client-Server

Here we assume that the server might crash. If a client sends a message to and does not get a reply (this is detected by a timeout) the client waits a random time and tries again. It tries this strategy several times and gives up if the cannot get a reply from the server after 10 tries.

- Create a registered server called double.
- If you send it an integer it doubles it and sends back the reply.
- It crashes if you send it an atom.
- Make a process that sleeps for a random time and sends a message to the double server that causes it to crash.
- Make a monitor process that detects that the server has crashed. It restarts the server after a random delay.
- Make a client function that sends a request to the server and times out if the request is not satisfied. We can assume the server has crashed. The client should wait a second and then try again.
- Abort the client if it has tried more than ten times.

Hints: `crypto:rand_uniform(Lo, Hi) -> N` Generates a random number `N`, where `Lo =< N < Hi`.

A sleep function can be defined as follows:

```
sleep(T) ->
    %% sleep for T milliseconds
    receive
      after T ->
        true
    end.
```

5 advanced: Replicated Data

We're going to create two registered processes. Call them master and replica. The idea is that the master process will have a data store and that the replica process will have a copy of the store.

Normally we will update and request data from the master process, but if the master process crashes we'll use the replica process.

Write the following routines:

5.1 advanced: `store:start()`

.

starts the master and replica processes.

5.2 advanced: `store:store(Key, Value)` and `store:fetch(Key)`

`store(Key, Value)` stores a Key-Value pair in the data store. A master copy is kept in the master process, and a replica in the replica process. `fetch(Key)` retrieves the data from the store.

Both `store` and `fetch` are interfacing functions which send and receive messages between the client and the master and replica processes.

The store should function correctly if the master processes crashes.

Making the program work for arbitrary crashes

Note: In general distributed data replication is very difficult to achieve. If both the master and replica can crash at arbitrary times, and if arbitrary

messages can be lost the problem is impossibly difficult to solve[1].

6 Sending messages in a ring

6.1 normal: `ring:start(N,M)`

`ring:start(N,M)` should create a ring of `N` processes, then send an integer `M` times round the ring. The message should start as the integer 0 and each process in the ring should increment the integer by 1. After the message has been round the ring `M` times its final value should be `N*M`.

`ring:start(N,M)` should return the value of the last message (which should be `N*M`) and make sure that all the processes in the ring have terminated.

6.2 advanced: `ring:time(N,M)`

Time how long it takes to run `ring:start(N,M)`
(Hint, use `timer:tc(Mod, Func, Args)`[2]).

Measure the time it takes to send a message round the ring for reasonably large values of `N` and `M`. The product of `N*M` should be a few million. Try this for tens of thousands to millions of processes. Experiment with the `erl -smp disable`.

7 optional: A Mail Server

Write a simple mail server. The server should respond to 4 messages:

`{message,From,Pwd,To,Subject,Data}`
: send a message to the server, all the arguments are binaries. When the message arrives at the server the server should check that the user `From` has password `Pwd` and that the user `To` exists.

  The possible responses are:

[1] There are some theorems about this, this is why I've only made a simple error case where only the master crashes.

[2] Consult the documentation for the timer module.

- `eNoSuchUser` – if the person `To` does not exist.
- `eBadPassword` – if the person `From` does not have the password `Pwd`.
- `{ok, MailId}` – if the mail was accepted. `MailId` is a unique identifier identifying this mail. The mail should be stored in a mailbox associated with the user `To` together with a timestamp.

`{list_my_mail,User,Pwd}`

This message is used to check for mail.

The possible responses are:

- `eNoSuchUser` – if the person `To` does not exist.
- `eBadPassword` – if the person `From` does not have the password `Pwd`.
- `{ok, [{MailId,From,Header}]}` – returns a list of the mail waiting for the the user.

`{get_mail,MailId,Pwd}`

This message is used to fetch a specific mail. `Pwd` is the password of the user to whom the mail with Id `mailId` was sent.

The possible responses are:

- `error` – the mail Id does not exist or the password is incorrect.
- `{ok, [{MailId,From,Header,Content,Timestamp}]}` – returns the mail

`{delete_mail,MailId,Pwd}`

This message is used to delete a specific mail. The arguments have the same meaning as for the `get_mail` command.

The possible responses are:

- `error` – the mail Id does not exist or the password is incorrect+
- `{deleted, MailId}` the mail was deleted.

7.1 A sample session with the mail server

Assume Alice has password `ecila` and Bob `blurb`

1. *Alice sends a message to Bob*. She sends the message:

   ```
   {mail,<<"alice">>,<<"ecila">>,<<"bob">>,
                   <<"hello">>,<<"This is ..">>)
   ```

   The mail server check that Alice's password is correct. It is correct so the mail is added to Jim's mailbox. The server responds `{ok,43}`. This is mail number 43. The server adds a time stamp to the mail to record when it received the mail.

2. *Bob want to check his mail*. He sends the message:

   `{check_my_mail,<<"bob">>,<<"blurb">>>}`

   To the server. Bob has supplied the correct password, so the server responds:

   `{mails, [..., {mail,43,<<"alice">>,<<"hello">>}, ...]}`

   So Bob can see that he has got mail from Alice.

3. Bob retrieves the mail by sending the message:

   `{get_mail,43,<<"blurb">>}`

   to the server. He wants to read mail 43 which the server knows is in Bob's mailbox. To do this Bob supplies his password (`blurb`). The password is correct so the mail server replies:

   ```
   {msg,43,<<"alice">>,
           <<"hello">>,<<"This is ...">>,
           <<"2013-02-12 12:12:34">>}
   ```

   Bob reads the mail.

4. Bob deletes the mail by sending the message:

   `{delete_mail,43,<<"blurb">>}` to the server. The server responds `{deleted,43}`