CS 683 Emerging Technologies
Fall Semester, 2008
Doc 3 Erlang Exceptions & Concurrency
Sept 9 2008

References

Programming Erlang: Software for a Concurrent World, Armstrong, Chapter 4, 8, 10.

Erlang Documentation

Reading

Programming Erlang: Software for a Concurrent World, Armstrong, Chapter 4, 8.

Chapter 6 contains useful information about running and debugging Erlang code.
You will find it very useful

Section 5.4 contains some useful information. The rest of chapter 5 you can skip for now.

Raising Exceptions

exit(Why)
throw(Why)
erlang:error(Why)

```
factorial(1) -> 1;
factorial(N) when N < 1 ->
      erlang:error({factorialNonPositiveArgument, N});
factorial(N) ->
      N * factorial(N - 1).
```

```
4> stuff:factorial(-2).
** exception error: {factorialNonPositiveArgument,-2}
     in function  stuff:factorial/1
```


There are three different functions we can use to raise an exception

Catching error

```
testThrow (N) ->
    try factorial(N) of
        Result -> {normal, Result}
    catch
        error:Exception -> {thrown,N, Exception}
    after
        io:format("Option like Java's finally")
        %no return values in after
    end.
```

```
factorial(1) -> 1;
factorial(N) when N < 1 ->
    erlang:error({factorialNonPositiveArgument, N});
factorial(N) ->
    N * factorial(N - 1).
```

Catching throw

```
testThrow (N) ->
    try factorial(N) of
        Result -> {normal, Result}
    catch
        throw:Exception -> {thrown,N, Exception}
    after
        io:format("Option like Java's finally\n")
        %no return values in after
    end.

factorial(1) -> 1;
factorial(N) when N < 1 ->
    throw({factorialNonPositiveArgument, N});
factorial(N) ->
    N * factorial(N - 1).
```

```
8> stuff:testThrow(-31).
Option like Java's finally
{thrown,-31,{factorialNonPositiveArgument,-31}}
```

Basic Concurrency

Primitives

Pid = spawn(Fun)	create a process
Pid ! Message	send a message to process with Pid
receive ... end	receive a message

Used in Several Examples

```
-module(stuff).
-export([factorial/1,safeFactorial/1).

factorial(1) -> 1;
factorial(N) when N < 1 ->
      throw({factorialNonPositiveArgument, N});
factorial(N) ->
      N * factorial(N - 1).

safeFactorial(N) ->
      try factorial(N) of
            Result ->  {ok,Result}
      catch
            throw:Exception -> Exception
      end.
```

Server

```
-module (factorialServer).
-export ([start/0]).
-import (stuff, [safeFactorial/1, factorial/1]).

start() -> spawn(fun loop/0).

loop () ->
        receive
            {ClientPid, factorial, N} ->
                ClientPid ! {self(), stuff:safeFactorial(N)},
                loop()
        end.
```


We are sending server Pid back to the client in the response. We do not need to do this, but it allows the client to filter messages based on the server Pid. It also makes it harder for someone to spoof the server.

Client

```
-module (factorialClient).
-export ([factorialRpc/2]).

factorialRpc (ServerPid, N) ->
      ServerPid ! {self(), factorial, N},
      receive
            {ServerPid, Response} ->
                  Response
      end.
```

```
1> Pid = factorialServer:start().
<0.33.0>
2> factorialClient:factorialRpc(Pid,4).
{ok,24}
3> factorialClient:factorialRpc(Pid, -4).
{factorialNonPositiveArgument,-4}
```


Of course one has to compile all the code first. The code "factorialServer:start()." creates a new process in the same VM that is running the Erlang shell. The book would say that the server and client are running on the same node.

Hiding the Pid

```
-module (factorialServerNoPidNeeded).
-export ([start/0, rpc/1]).
-import (stuff, [safeFactorial/1, factorial/1]).

start() -> register(fac, spawn(fun() -> loop() end)).

rpc ( N) ->
      fac ! {self(), factorial, N},
      receive
            {fac, Response} -> Response
      end.

loop () ->
      receive
            {ClientPid, factorial, N} ->
                  ClientPid ! {fac, stuff:safeFactorial(N)},
                  loop()
      end.
```


Here I am using the convention from the book of putting client and server code in one file. The register function associates the atom "fac" with the server pid.

Issues

Server Exceptions
Message Mailbox
Timeouts
Remote Machines/Nodes

Server With Exceptions

```
-module (factorialServer).
-export ([start/0]).
-import (stuff, [safeFactorial/1, factorial/1]).

start() -> spawn(fun loop/0).

loop () ->
      receive
            {ClientPid, factorial, N} ->
                  ClientPid ! {self(), stuff:factorial(N)},
                  loop()
end.
```


When N is negative we will get a throw that the server does not catch.

Uncaught Server Side Throw

```
1> Pid = factorialServer:start().
<0.33.0>
2> factorialClient:factorialRpc(Pid, 4).
24
3> factorialClient:factorialRpc(Pid, -4).

=ERROR REPORT==== 8-Sep-2008::13:12:01 ===
Error in process <0.33.0> with exit value: {{nocatch,
{factorialNonPositiveArgument,-4}},[{stuff,factorial,1}]}
```

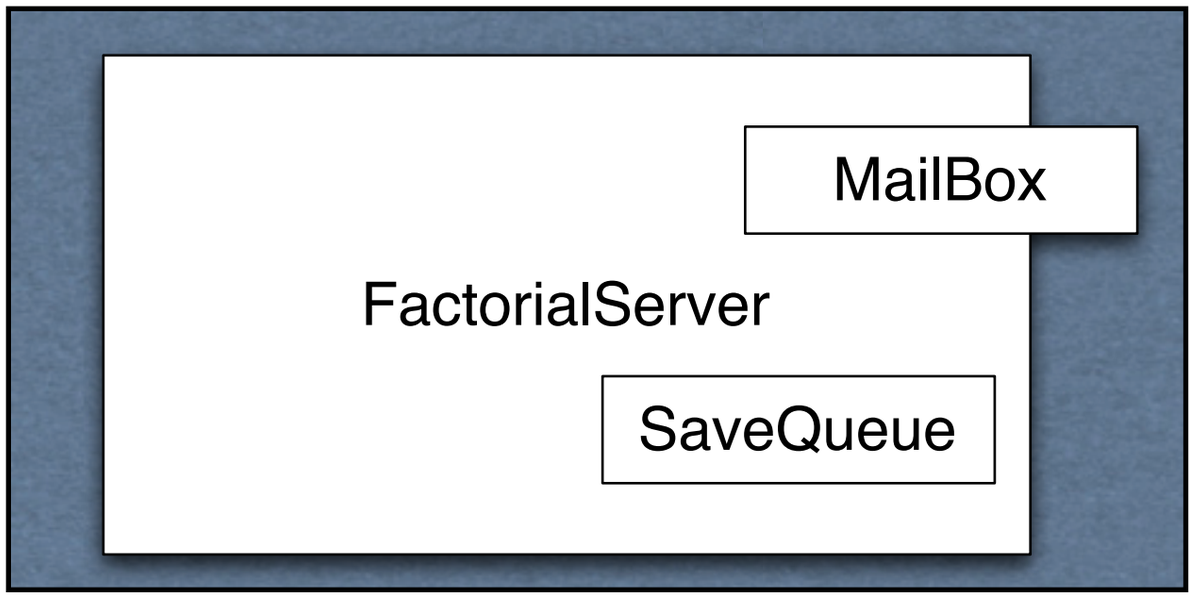

The erlang shell no longer responds. We will see more about this later.

Message Mailbox

```
1> Pid = factorialServer:start().
<0.33.0>
2> factorialClient:factorialRpc(Pid, 5).
{ok,120}
3> Pid ! {foo, 5}.
{foo,5}
4> Pid ! {bar}.
{bar}
5> factorialClient:factorialRpc(Pid, 6).
{ok,720}
```


What happens to those messages?

Some Message Details



Incoming messages

Added to mailbox

receive

Wait until message arrives

Repeat until find match

Inspect first message

If match

remove and process

Copy SaveQueue back

else move to SaveQueue

Timeouts

```
receive
    Pattern1 [when Guard1] -> Expression1;
    Pattern2 [when Guard2] -> Expression2;
    ...
    PatternN [when GuardN] -> ExpressionN
after
    TimeAmount -> ExpessionTimeout
end
```

Timeout Example

```
-module (factorialClient).
-export ([factorialRpc/1]).

factorialRpc ( N) ->
        fac ! {self(), factorial, N},
        receive
                {fac, Response} ->
                        Response
        after 1000 ->
                io:format("time out\n")
end.
```

Timer

```
-module (bookTimer).
-export ([start/2,cancel/0]).

start (Time,Fun) ->
      register(timer, spawn(fun() -> timer(Time, Fun) end)).

cancel() -> timer ! cancel.

timer (Time, Fun) ->
      receive
            cancel ->
                  void
      after Time ->
            Fun(),
            timer(Time, Fun)
      end.
```

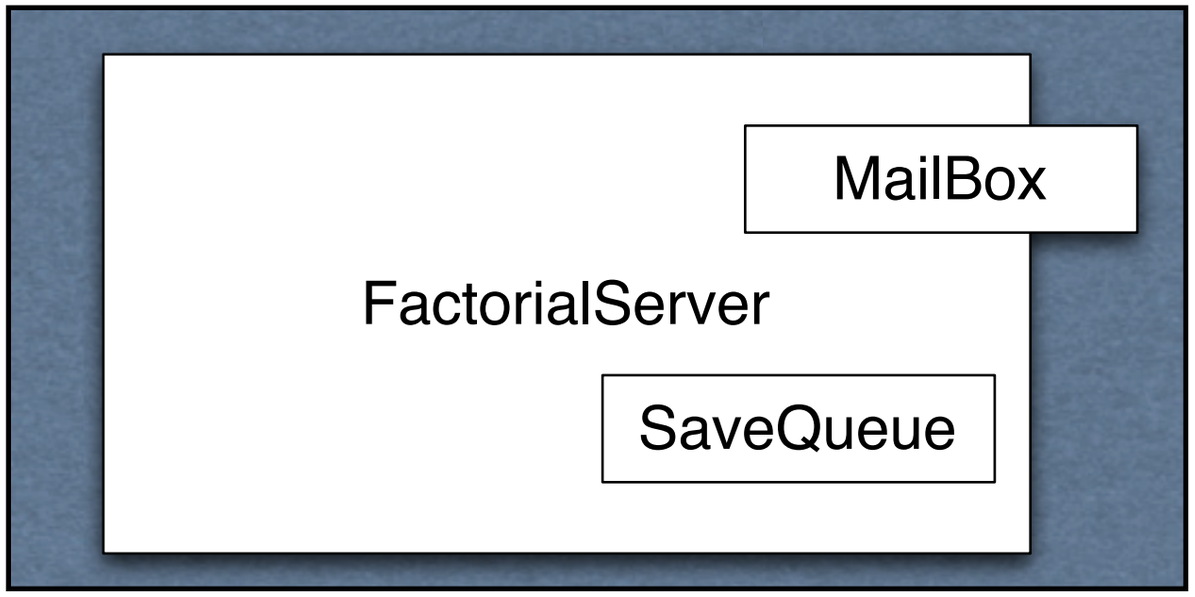

Modified version of the timer code from the text.

Urgent Messages

```
priority_receive() ->
    receive
        {urgent, Message} ->
            handle the message here,
            priority_receive()
    after 0 ->
        receive
            Any ->
                handle messge here,
                priority_receive()
        end
end.
```


This allows the sender to send an urgent (or out-of-bounds) message that will be read before regular messages that have been sent earlier but are still pending.

Message Details with Timeout



"after" section is only done after checking all messages in the Mailbox

If timeout occurs while waiting for a message evaluate the after code and move SaveQueue back in Mailbox

Incoming messages
Added to mailbox

receive
Wait until message arrives
Repeat until find match
Inspect first message
If match
remove and process
move SaveQueue back in Mailbox
else move to SaveQueue

One Machine, Two Nodes

Terminal One

```
Al pro 42->erl -sname localServer
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.3  (abort with ^G)
(localServer@AlPro)1> factorialServer:start().
true
(localServer@AlPro)2>
```

Terminal Two

```
Al pro 19->erl -sname clientTest
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.3  (abort with ^G)
(clientTest@AlPro)1> rpc:call(localServer@AlPro, factorialClient, factorialRpc,[8]).
{ok,40320}
(clientTest@AlPro)2>
```


sname stands for short name. Here we start up two different VMs on the same machine, one for the server and one for the client. We need to use rpc:call in the client to send a message to the server process in the other VM.

Using Two Machines

Machine 1

```
Al pro 43->erl -name server -setcookie test
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.3  (abort with ^G)
(server@AlPro.sd.cox.net)1> factorialServer:start().
true
```

Machine 2

```
Air 15->erl -name client -setcookie test
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.3  (abort with ^G)
(client@Air.sd.cox.net)1> rpc:call(server@AlPro.sd.cox.net, factorialClient,factorialRpc,[10]).
{ok,3628800}
(client@Air.sd.cox.net)2>
```


Code has to be same version on both machines. Machines need to allow incoming connections (port 4369 and others). Server needs to be DNS resolvable.(If on same LAN can use -sname without DNS.) Each machine needs same cookie. If not on the same LAN make sure firewalls permit connections. Need more security setup than this if not on same LAN. This example needs more work before we use it to deploy an Erlang program.