Joe's spitting in the sawdust Erlang tutorials
Tutorial number 1
Last edited 2003-02-13
A Fault-tolerant server

This tutorial shows you how to build a simple fault-tolerant server. All the code is here.

The example chosen is a "bank" server - the "server" is actually a pair of servers - if both servers crash - then the system is unavailable. If one server crashes, the other server takes over. Data is fully replicated on both servers.

Please, report all errors, omissions or improvements to the author.

1. A simple server
   1.1. A simple client
   1.2. A simple socket based server
   1.3. Initializing the data-base
   1.4. Data base access routines
   1.5. Running the program
2. A fault tolerant server
   2.1. A robust client
   2.2. A robust server
   2.3. Initializing the data base
   2.4. Now we are ready to run everything
3. Summing up
4. Notes

The problem

We want to make a *bank* server. The bank server models the behaviour of a real bank. You can deposit money with the bank, query the status of your account and take money out of your account.

Banks, being suspicious by nature do not let you take out more money than you have in your account.

Banks are worried about security and like to offer round-the-clock services. If one of their computer fails they still like to serve their customers. In the event of a failure they like everything to work as if the failure had not occurred. In particular even of a failure they do not want you to remove more money from you account than you have deposited ...

A fault-tolerant server

In order to make a fault-tolerant system you need at least two computers. No matter how good your software is you cannot make a fault-tolerant system using only one computer - if you have only one computer and it crashes you application will break.

We will use two servers, all data will be replicated on both servers.

There are two choices for how we structure our applications:

1. The client only knows about one server, or,
2. The client knows about *both* servers

In the first case the client knows one IP address (or domain name) and if the server at this IP address fails, some combination of hardware or software at the server site has to arrange that packets sent to the IP address of the server always arrive at a functioning machine.

There are various techniques for this, none of them are simple - most of them involve deep trickery, these solutions are highly non-portable.

In the second case the client has two addresses for the server. When the client wishes to do something it uses the first address. If the machine at this address is broken, it uses the second address. This method is easy to implement and needs no special purpose hardware. This method is well proven, and is, for example used by DNS. We will use this method in our server.

In the following example we will show how to program the fault-tolerant server in a number of small steps, firstly, we develop a simple non-fault tolerant server, then extend it to make a fault tolerant server.

1. A simple server

We start with a very simple server. It is non fault-tolerant. If the server crashes, hopefully no data will be lost. But the service will not operate.

This solution does not involve distributed Erlang. Their are several reasons for this:

- Distributed Erlang has *all or nothing* security [1] (so this is just too dangerous)
- Distributed Erlang was not designed for thousands of clients

The client-server communication is based on simple socket communication.

1.1. A simple client

We start with the bank_client.erl

```
-module(bank_client).

-export([deposit/2, withdraw/2, balance/1]).

deposit(Who, X)  -> simple_rpc({deposit, Who, X}).
withdraw(Who, X) -> simple_rpc({withdraw, Who, X}).
balance(Who)     -> simple_rpc({balance, Who}).

simple_rpc(X) ->
    case gen_tcp:connect("localhost", 3010,
                         [binary, {packet, 4}]) of
        {ok, Socket} ->
            gen_tcp:send(Socket, [term_to_binary(X)]),
            wait_reply(Socket);
        E ->
            E
    end.

wait_reply(Socket) ->
    receive
        {tcp, Socket, Bin} ->
```

```
            Term = binary_to_term(Bin),
            gen_tcp:close(Socket),
            Term;
        {tcp_closed, Socket} ->
            true
      end.
```

This is a simple "no frills" client, that accesses a bank server.

The address of the bank server is "hard wired" into the program at address **`localhost`** and port **`3010`**.

Since we are not using distributed Erlang we have to do all encoding and decoding of Erlang terms ourselves. This is achieved by using **`term_to_binary`** to encode the term and **`binary_to_term`** to decode the term.

Note also that the socket was opened with the argument **`{packet, 4}`** - this must match up with a corresponding argument in the server code. **`{packet, 4}`** means that all packets will be preceded by a 4 byte length count and that the library routines in **`gen_tcp`** will correctly assemble fragmented packets, in other words, the user won't have to worry about TCP IP packet fragmentation since only competed packets will be delivered to the Erlang processes involved.

1.2. A simple socket based server

bank_server.erl which communicates with the client is:

```
-module(bank_server).

-export([start/0, stop/0]).

start() ->
    mnesia:start(),
    spawn_link(fun() -> server(3010) end).

stop() ->
    mnesia:stop(),
    tcp_server:stop(3010).

server(Port) ->
    tcp_server:
        start_raw_server(Port,
                         fun(Socket) ->
                                 input_handler(Socket)
                         end,
                         15,
                         4).

input_handler(Socket) ->
    receive
        {tcp, Socket, Bin} ->
```

```
            Term = binary_to_term(Bin),
            Reply = do_call(Term),
            send_term(Socket, Reply),
            input_handler(Socket);
        {tcp_closed, Socket} ->
            true
     end.

send_term(Socket, Term) ->
    gen_tcp:send(Socket, [term_to_binary(Term)]).

do_call(C) ->
    Fun = the_func(C),
    mnesia:transaction(Fun).

the_func({deposit, Who, X})  ->  bank:deposit(Who, X);
the_func({withdraw, Who, X}) ->  bank:withdraw(Who, X);
the_func({balance, Who})     ->  bank:balance(Who).
```

Again this is a "no frills" sever. The server port is hard-wired to `3010`

Most of the work is done in tcp_server.erl. `tcp_server` keeps track of the number of socket sessions etc.

Note also that we use mnesia to keep track of the state of our bank balance.

1.3. Initializing the data-base

Our server stores all data in a disk based data-base - bank_manager.erl is used to initialize the data base before the program can be run.

```
-module(bank_manager).

-export([init_bank/0]).

-include("bank.hrl").

init_bank() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
    mnesia:create_table(account,
                        [{disc_copies,[node()]},
                         {attributes,
                          record_info(fields, account)}]),
    mnesia:stop().
```

To initialize the bank, we have to start Erlang and run the command `bank_manager:init_bank()`. This command must be done once only.

1.4. Data base access routines

Finally the code to access the data base:

The data base consists of a number of **account** records stored in the header file bank.hrl.

```
-record(account, {name,balance}).
```

These are manipulated with code in bank.erl:

```
-module(bank).

-export([deposit/2, withdraw/2, balance/1]).

-include("bank.hrl").

deposit(Who, X) ->
    fun() ->
            case mnesia:read({account, Who}) of
                [] ->
                    %% no accout so we make one
                    Entry = #account{name=Who,balance=X},
                    mnesia:write(Entry),
                    X;
                [E] ->
                    Old = E#account.balance,
                    New = Old + X,
                    E1 = E#account{balance=New},
                    mnesia:write(E1),
                    New
            end
    end.

balance(Who) ->
    fun() ->
            case mnesia:read({account, Who}) of
                [] ->
                    %% no account
                    {error, no_such_account};
                [E] ->
                    B = E#account.balance,
                    {ok, B}
            end
    end.

withdraw(Who, X) ->
    fun() ->
            case mnesia:read({account, Who}) of
                [] ->
                    %% no account
                    {error, no_such_user};
                [E] ->
```

```
                    Old = E#account.balance,
                    if
                        Old >= X ->
                            New = Old - X,
                            E1 = E#account{balance=New},
                            mnesia:write(E1),
                            ok;
                        Old < X ->
                            {error, not_enough_money}
                    end
            end
    end.
```

1.5. Running the program

We'll done go through all the steps necessary to run the program.

We start two shells. In shell one we compile the program and start the server:

```
$ erlc *.erl
[joe@enfield simple_socket_server]$ erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1> bank_manager:init_bank().
stopped

=INFO REPORT==== 18-Dec-2002::11:39:43 ===
    application: mnesia
    exited: stopped
    type: temporary
2> bank_server:start().
<0.107.0>
Starting a port server on 3010...
3>
```

First we compiled all the Erlang code `erlc *.erl`, started Erlang, initialized the data base `bank_manager:init_bank()` and finally started the server `bank_server:start()`.

Now we can move to a second window, start Erlang and access the server:

```
 erl
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
1> bank_client:balance("joe").
{atomic,{error,no_such_account}}
2> bank_client:deposit("joe", 10).
{atomic,10}
3> bank_client:deposit("joe", 15).
{atomic,25}
```

```
4> bank_client:balance("joe").
{atomic,{ok,25}}
5> bank_client:withdraw("joe", 1234).
{atomic,{error,not_enough_money}}
6> bank_client:withdraw("joe", 12).
{atomic,ok}
7> bank_client:balance("joe").
{atomic,{ok,13}}
8>
```

Everything works fine, but if the server crashes all is lost.

The next section makes a fault-tolerant version of the program using two servers. If one of the severs crashes, the other server will be used. The user will not notice that the server has crashed, read on ...

2. A fault tolerant server

To make our fault-tolerant server we use not one machine but two.

The two servers both run distributed Erlang and trust each other.

The clients know the hostnames and ports that are used to access the server.

2.1. A robust client

As in our simple example we start with the client code robust_bank_client.erl :

```
-module(robust_bank_client).

-export([deposit/2, withdraw/2, balance/1]).

deposit(Who, X)  -> robust_rpc({deposit, Who, X}).
withdraw(Who, X) -> robust_rpc({withdraw, Who, X}).
balance(Who)     -> robust_rpc({balance, Who}).

robust_rpc(X) ->
    Id = make_ref(),
    X1 = {call, Id, X},
    io:format("trying to connect to server 1~n"),
    case gen_tcp:connect("localhost", 3020,
                         [binary, {packet, 4}]) of
        {ok, Socket} ->
            io:format("sending to server 1~n"),
            gen_tcp:send(Socket, [term_to_binary(X1)]),
            wait_reply1(Socket, Id, X);
        {error, _} ->
            io:format("cannot connect to server 1~n"),
            robust_rpc_try_again(Id, X)
    end.

wait_reply1(Socket, Id, X) ->
    receive
        {tcp, Socket, Bin} ->
            case binary_to_term(Bin) of
                {ack, Id, Reply} ->
                    io:format("server 1 replied~n"),
                    B = term_to_binary({delete_tag, Id}),
                    gen_tcp:send(Socket, B),
                    gen_tcp:close(Socket),
                    {ok, {server1, Reply}};
                _ ->
                    robust_rpc_try_again(Id, X)
            end;
        {tcp_closed, Socket} ->
            robust_rpc_try_again(Id, X)
    after 10000 ->
            io:format("timeout from server 1~n"),
            gen_tcp:close(Socket),
            robust_rpc_try_again(Id, X)
    end.

robust_rpc_try_again(Id, X) ->
    io:format("trying to connect to server 2~n"),
    case gen_tcp:connect("localhost", 3030,
                         [binary, {packet, 4}]) of
        {ok, Socket} ->
            X1 = {call, Id, X},
            io:format("sending to server 2~n"),
            gen_tcp:send(Socket, [term_to_binary(X1)]),
            wait_reply2(Socket, Id);
```

```
        {error, E} ->
            io:format("cannot connect to server 2~n"),
            {error, both_servers_down}
    end.

wait_reply2(Socket, Id) ->
    receive
        {tcp, Socket, Bin} ->
            case binary_to_term(Bin) of
                {ack, Id, Reply} ->
                    B =  term_to_binary({delete_tag, Id}),
                    gen_tcp:send(Socket, B),
                    gen_tcp:close(Socket),
                    {ok, {server2, Reply}};
                O ->
                    {error, {unexpect_reply, O}}
            end;
        {tcp_closed, Socket} ->
            {error, server2}
    after 10000 ->
            io:format("timeout from server 2~n"),
            gen_tcp:close(Socket),
            {error, no_reply_sever2}
    end.
```

This bank client knows that there are two servers. For each transaction it generates a unique transaction Id (by calling `new_ref()`).

The client first tries to contact the first server sending with it the query and a unique tag.

If the client cannot connect to the server it tries the second server, and if this attempt fails the entire transaction fails since both servers are broken.

If the server manages to connect to the first server and send a message to the server, it will never know if the server managed to process the request, thus is the server does not reply or if a timeout occurs then the client does not know if the sever received the message and performed the transaction, or if the message was never received.

In the case of a failed transaction the client contacts the second server and repeats the request it made to the first server - each request is tagged with a unique tag. Every time a server replies the value of the reply is stored in the data base together with the tag. Before performing a computation the server checks to see if there is a return value associated with a particular tag - if so this value is send and the computation is not repeated. Since all data is reliably replicated across both server this strategy ensures that computations will be idempotent - i.e. carried out either once or not at all.

When the client has received a return value from one of the servers it responds with a `{delete_tag, Id}` message which causes the server to erase the cached value of the return value to the client. If this message never arrives, no errors will occur (since a new unique tag is generated for each new transaction), but only cached return values might accumulate in the data base. For this reason cached return values should probably be date stamped and

given a certain "time to live" - they can be garbage collected at a later date, if this is a problem. This is not done in this version of the program.

2.2. A robust server

The code for robust_bank_server.erl is very similar to that of a non-robust server:

```
-module(robust_bank_server).

-export([bstart/1, start/1, stop/1]).

%% make two windows
%%  1) In window one
%%     $ erl -sname one -mnesia dir '"one"'
%%     robust_bank_server:start(3020).
%%  2) In window two
%%     $ erl -sname two -mnesia dir '"two"'
%%     robust_bank_server:start(3030).

-include("reply.hrl").

bstart([APort]) ->
    Port = list_to_integer(atom_to_list(APort)),
    start(Port).

start(Port) ->
    mnesia:start(),
    spawn_link(fun() -> server(Port) end).

stop(Port) ->
    mnesia:stop(),
    tcp_server:stop(Port).

server(Port) ->
    tcp_server:
        start_raw_server(Port,
                         fun(Socket) ->
                                 input_handler(Socket, Port)
                         end,
                         15,
                         4).

input_handler(Socket, Port) ->
    receive
        {tcp, Socket, Bin} ->
            case binary_to_term(Bin) of
                {call, Tag, Term} ->
                    io:format("Server port:~p Tag:~p
call:~p~n",
                              [Port,Tag,Term]),
                    Reply = do_call(Tag, Term),
                    send_term(Socket, {ack, Tag, Reply}),
                    input_handler(Socket, Port);
```

```
                {delete_tag, Tag} ->
                    io:format("Server port:~p
deleteTag:~p~n",
                              [Port,Tag]),
                    delete_tag(Tag)
            end;
        {tcp_closed, Socket} ->
            true
     end.

send_term(Socket, Term) ->
    gen_tcp:send(Socket, [term_to_binary(Term)]).

do_call(Tag, C) ->
    Fun = the_func(C),
    F = fun() ->
           case mnesia:read({reply, Tag}) of
               [] ->
                   %% no cached value
                   Val = Fun(),
                   mnesia:write(#reply{tag=Tag,val=Val}),
                   Val;
               [C] ->
                   %% yes - return cached value
                   C#reply.val
           end
        end,
    mnesia:transaction(F).

the_func({deposit, Who, X})  ->  bank:deposit(Who, X);
the_func({withdraw, Who, X}) ->  bank:withdraw(Who, X);
the_func({balance, Who})     ->  bank:balance(Who).

delete_tag(Tag) ->
    F = fun() -> mnesia:delete({reply, Tag}) end,
    V = mnesia:transaction(F),
    io:format("delete tag=~p~n",[V]).
```

There are only a small number of minor changes to the code, when compared to the not robust code.

2.3. Initializing the data base

Now things become slightly more complicated. Instead of one data base we need two - also we need to get our node names correct. This is done in robust_bank_manager.erl

```
-module(robust_bank_manager).
```

```
-export([create_schema/0, create_table/0]).

-include("bank.hrl").
-include("reply.hrl").

db_nodes() ->
    [one@enfield, two@enfield].

create_schema() ->
    mnesia:create_schema(db_nodes()).

create_table() ->
    mnesia:create_table(account,
                        [{disc_copies,db_nodes()},
                         {attributes,
                          record_info(fields, account)}]),
    mnesia:create_table(reply,
                        [{disc_copies,db_nodes()},
                         {attributes,
                          record_info(fields, reply)}
                        ]).
```

This assumes that there are two nodes called **one@enfield** and **two@enfield**.

To initialize the system we open two terminal windows (on enfield) and proceed as follows:

In xterm 1.

```
$ erl -sname one -mnesia dir '"one"'
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
```

In xterm 2:

```
$ erl -sname two -mnesia dir '"two"'
Erlang (BEAM) emulator version 5.2 [source] [hipe]

Eshell V5.2  (abort with ^G)
(two@enfield)1> robust_bank_manager:create_schema().
ok
(two@enfield)2> mnesia:start().
ok
```

In xterm 1:

```
(one@enfield)1> mnesia:start().
ok
(one@enfield)2> robust_bank_manager:create_table().
```

The ctrl-c in *both* xterms.

2.4. Now we are ready to run everything

At this point you need to open *three* terminal windows.

Open window 1 and run the shell script start_one , open window 2 and run the shell script start_two . This will start two servers, one listening to port 3020 the other listening to 3030.

Open a third window and start Erlang. In what what follows the output show is what happens in the third Erlang window.

To kill a server, you move to the appropriate window and type Control+C. To restart a server you type **`start_one`** (or **`start_two`**), in the appropriate window:

Query the server

```
> robust_bank_client:balance("joe").
trying to connect to server 1
sending to server 1
server 1 replied
{ok,{server1,{atomic,{ok,7}}}}
```

Both servers are running - server one replies:

Kill server one, and re-do the query

```
4> robust_bank_client:balance("joe").
trying to connect to server 1
cannot connect to server 1
trying to connect to server 2
sending to server 2
{ok,{server2,{atomic,{ok,7}}}}
```

This time server 2 replies (we killed server one, remember).

Make a deposit

We make a deposit of 10 units,

```
5> robust_bank_client:deposit("joe", 10).
trying to connect to server 1
cannot connect to server 1
trying to connect to server 2
sending to server 2
{ok,{server2,{atomic,17}}}
```

Only server two is running - so the transaction takes place on server two.

Restart server one and query the balance

```
6> robust_bank_client:balance("joe").
trying to connect to server 1
sending to server 1
server 1 replied
{ok,{server1,{atomic,{ok,17}}}}
```

Server one replies with 17 units well done server one.

When server one was restarted - the two servers synced their data and the changes made to server two were propagated to server one.

Kill server two and query the balance

```
> robust_bank_client:balance("joe").
trying to connect to server 1
sending to server 1
server 1 replied
{ok,{server1,{atomic,{ok,17}}}}
```

Server one replies ...

3. Summing up

The intention here was not to make the ultimate fault-tolerant server, but rather to illustrate how to make a simple functioning server, with no detail omitted. A production server could be based on this simple design, but would involve a slightly less simplistic approach. The following improvements, might be considered:

- To make queries idempotent I cache the last return value and deleted it when the client has received the return value. Certain errors might cause cached return values not to be deleted, this could occur, for example, if the **`delete_tag`** message from the client to the server is never received. These cached values could be garbage collected at some suitable interval.
- I use "disk replicated" mnesia tables - a more sophisticated design using combinations of disk replicated tables (for persistent data) and memory replicated tables (for reply caching) might be better.
- *Suggested by Dominic Williams* When server1 is down, each transaction will have degraded performance, because the client still starts by trying one. An improved client would remember the last server that worked and use it first before trying the other server.

4. Notes

Security

Dominic Williams asked what "all or nothing security" meant.

Once you get distributed Erlang running a remote node can do anything to a remote node that is allowed by the user privileges of the remote node.

All nodes in a distributed Erlang node can do anything they want on other nodes in the system - this is *very* dangerous. I could easily do a `os:cmd("rm -rf *")` on a remote machine and blow away the entire remote user's filesystem.

If you want to write distributed applications, you can choose to either not use distributed Erlang and use an explicit socket for everything - in which case you strictly check everything coming into you system, or, set up a special user with restricted privileges, and run the Erlang node from within this user. This tutorial showed how to do things with an explicit socket.