Files - Sockets

Lectures F8 + F9 (Chapters 15..18)

Joe Armstrong

Ericsson AB

W7/F8 - Frameworks1 - Files, Sockets - Chapters 15,16,17

- Interfacing.
- Files.
- Random access.
- Directory and file operations.
- Sockets.
- UDP.
- TCP.
- Parallel and Sequential servers.

W7/F9 - Frameworks2 - Websockets, - Chapters 17,18,

- Streaming music.
- Websockets.

Interfacing

- Ports.
- Sockets.
- Linked-in drivers.
- NIFS.

Ports

```
Port = open_port({spawn, "C Program"}, [{packet,2}]),
...
Port ! {self(), {command, Bin}
...
receive
   {Port, {data, Bin}} ->
      ...
   {Port, closed}
end
```

- Ports behave like linked processes.
- Message to the Port go to standard input/output of the external process.
- Message from the external process goes to the “connected process”.

Problems with interfacing

- Agreeing on “whats on the wire”.
- We could use JSON, XML, thrift, protocol buffers, ASN.1 ,...
- Both programs at either end of the wire must implement the same protocol.
- Difficult to describe protocols.

Safe or Unsafe interfaces

- Ports + Sockets - Safe (but watch out for DOS attacks and buffer overflows).
- Linked-in drivers (Ports on Steriods) - unsafe.
- NIFS - links object code into the erlang kernel. Extremely dangerous.

Files

- erl -man file
- `http://www.erlang.org/doc/man/file.html`

Reading binary data

- File at a time I/O is the most efficient

```
> file:read_file("f8-f9.org"}
{ok, <<"#+STARTUP: overview, hideblocks\n#+...
> file:read_file("missing").
{error, enoent}
> file:write_file("file", Bin)
```


Storing terms in files

```
term_to_file(File, X) ->
   file:write_file(File, term_to_binary(X)).

file_to_term(File) ->
   {ok, Bin} = file:read_file(File),
   binary_to_term(Bin).
```


Storing readable terms in files

```
consult(F) ->
    {ok, [L]} = file:consult(F),
    L.

unconsult(File, Term) ->
    {ok, S} = file:open(File, [write]),
    io:format(S, "~p.~n",[Term]),
    file:close(S).
```


Random access

```
test_random_io() ->
    file:write_file("abc",<<"0123456789">>),
    {ok, S} = file:open("abc", [read,raw,binary]),
    {ok,<<"0123">>} = file:pread(S, 0, 4),
    {ok, <<"678">>} = file:pread(S, 6, 3),
    {ok, <<"78">>} = file:pread(S, 7, 2),
    {ok, <<"789">>} = file:pread(S, 7, 3),
    {ok, <<"789">>} = file:pread(S, 7, 10),
    file:close(S).
```


Directory and file operations

```
1> file:list_dir(".").
{ok,[".git",".log","abc","big",
    "big.digest","blocks.tmp", ...]}
2> filelib:is_file("f8-f9.org").
true
3> filelib:file_size("f8-f9.org").
1726
4> filelib:is_dir("../paradis").
true
```

- erl -man file
- erl -man filelib

Sockets

- A connection endpoint “Network socket”.
- Provide mechanisms for processes on the same machine or on different machines to communicate.
- Come in different types (Raw, UDP, TCP, SCTP).
- RAW = applications see everything.
- UDP = User Datagram Protocol.
- TCP = Transmission Control Protocol.
- SCTP = Stream Control Transmission Protcol.

UDP

- User Datagram Protocol.
- No connection setup.
- Data can be lost, no retransmission.
- Data can be fragment so use small packets (less than 576 bytes should not be fragemented).

TCP

- Connection oriented.
- Flow Control.
- Packets can be (are) fragmented.
- *“Reliable”*.

UDP

```
server(Port) ->
   {ok, Socket} = gen_udp:open(Port, [binary]),
   loop(Socket).

loop(Socket) ->
   receive
      {udp, Socket, Host, Port, Bin} ->
         ...
         gen_udp:send(Socket, Host, Port, Reply),
         loop(Socket)
   end.
```

```
{ok, Socket} = gen_udp:open(0, [binary]),
ok = gen_udp:send(Socket, "localhost", 4000, Bin)
```


UDP Factorial Server

```
start_server(Port) ->
   spawn(fun() -> server(Port) end).

%% The server
server(Port) ->
   {ok, Socket} = gen_udp:open(Port, [binary]),
   io:format("server opened socket:~p~n",[Socket]),
   loop(Socket).

loop(Socket) ->
   receive
      {udp, Socket, Host, Port, Bin} = Msg ->
         io:format("server received:~p~n",[Msg]),
         N = binary_to_term(Bin),
         Fac = factorial(N),
         gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
         loop(Socket)
   end.

factorial(0) -> 1;
factorial(N) when N > 0 -> N * fac(N-1).
```


UDP Factorial Client

```
fac(Host, Port, N) ->
   {ok, Socket} = gen_udp:open(0, [binary]),
   io:format("client opened socket=~p~n",[Socket]),
   ok = gen_udp:send(Socket, Host, Port,
                 term_to_binary(N)),
   Value = receive
             {udp, Socket, _, _, Bin} = Msg ->
                io:format("client received:~p~n",[Msg]),
                binary_to_term(Bin)
          after 2000 ->
                0
          end,
   gen_udp:close(Socket),
   Value.
```

A sample session

```
$ erl
1> c(udp_test).
{ok,udp_test}
2> udp_test:start_server(4000).
<0.40.0>
server opened socket:#Port<0.2437>
3>
```

```
$ erl
> udp_test:fac("localhost", 4000, 123).
121463043670253296757662432418812958554542170884833823153289181618292358923621676688311569606126402021707358352212940477825910915704116514721860295199062616467307339074198149529600000000000000000000000000
```


Erlang TCP client

```
{ok,Socket} = gen_tcp:connect(Host,Port,
                        [binary, {packet, 0}]),
ok = gen_tcp:send(Socket, ...),
receive
   {tcp,Socket,Bin} ->
      receive_data(Socket, Bin),
      ...
   {tcp_closed,Socket} ->
      ...
end.
```

- `{packet,0}` data gets sent without any length count.
- `{packet, 2 | 4}` data is sent with a 2 or 4 byte length header. The receiving side will automaticaly defragment the data if it was opened with packet 2 of 4 option.

Nano web client

```
-module(nano_web_client).
-compile(export_all).

nano_get_url() ->
    nano_get_url("www.sics.se").

nano_get_url(Host) ->
    {ok,Socket} = gen_tcp:connect(Host,80,
                            [binary, {packet, 0}]),
    ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
    receive_data(Socket, []).

receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, [Bin|SoFar]);
        {tcp_closed,Socket} ->
            list_to_binary(lists:reverse(SoFar))
    end.
```

- Page 264 Erlang book

Running the client

```
> nano_web_client:nano_get_url("www.google.com").
<<"HTTP/1.0 302 Found\r\nLocation:
http://www.google.se/?gws_rd=cr&ei=mY70UqaPNoaoywPQ94CI
Cache-Control: private\r\nCon"...>>
```


Erlang TCP server

```
start_nano_server() ->
   {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                 {reuseaddr, true},
                                 {active, true}]),
   {ok, Socket} = gen_tcp:accept(Listen),
   gen_tcp:close(Listen),
   loop(Socket).

loop(Socket) ->
   receive
      {tcp, Socket, Bin} ->
         ...
         Reply = ...
         gen_tcp:send(Socket, Reply),
         loop(Socket);
      {tcp_closed, Socket} ->
         true
   end.
```

- Page 268 Erlang book

Sequential and Parallel TCP Servers

```
start_seq_server() ->
    {ok, Listen} = gen_tcp:listen(Port, ..),
    seq_loop(Listen).

seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:listen(Listen),
    loop(Socket),
    seq_loop(Listen).
```

```
start_par_server() ->
    {ok, Listen} = gen_tcp:listen(Port, ..),
    spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
    {ok, Socket} = gen_tcp:listen(Listen),
    spawn(fun() -> par_connect(Listen) end).
    loop(Socket).
```


TCP + UDP problems

- UDP - lost packets.
- TCP - fragemented packages.
- TCP - flow control.
- Both - DOS attacks.
- Both - Security.
- Both - Firewalls.

Security 1

- Change:

```
ok = gen_udp:send(Socket, "localhost", 4000,
                  term_to_binary(Term)),
...
receive
  {udp, Socket, _, _, Bin} ->
    binary_to_term(Bin)
end.
```

- To:

```
ok = gen_udp:send(Socket, "localhost", 4000,
                  encrypt(Key, term_to_binary(Term))),
...
receive
  {udp, Socket, _, _, Bin} ->
    binary_to_term(decrypt(Key, Bin))
end.
```


Security 2

```
1> c(elib2_aes).
{ok,elib2_aes}
2 > Password = "1234".
"1234"
3> C = elib2_aes:encrypt(Password, <<"hello joe">>).
<<199,113,224,181,20,198,47,18,178,39,128,253,35,143,81,
  185,95,3,250,249,1,185,72,136,214,182,198,28,221,...>>
4> elib2_aes:decrypt(Password, C).
<<"hello joe">>
```

- No guarantees.
- Side channel attacks.

Things to try at home

- Shoutcast.
- Streaming music.

Websockets

- You can use sockets in the browser.
- Low overheads.
- Stream data in and out of the browser.
- In the book and on my github account.

Philosophy

- Let's be "truly OO"
- To get a thing in the browser to do something you send it a message.
- When somthing interesting happens in the browser you get sent a message.
- This is NOT ajax, nor long-polling

Sending messages to DIVS

- Step 1) In HTML we can define a div:

```
<div id="clock"></div>
```

- Step 2) In the browser we call a Javascript function to connect to Erlang:

```
connect_to_erlang("localhost", 1456, "clock1");
```

- Step 3) in Elang the function `clock1:start` is spawned:

```
start(Browser) ->
    Browser ! [{cmd,fill_div}, {id,clock},
               {txt,current_time()}],
    running(Browser).
```


Extending the system:

- In Erlang:

```
Browser ! [{cmd,CName}, {tag1,va1},{tag2,val2,..}]
```

- In Javascript we evaluate:

```
Cname({cmd:CName, tag1:"va1", tag2:"val2", ..});
```

- So `fill_div` in JS is:

```
function fill_div(o){
    $('#'+o.id).html(o.txt);
}
```


Ezwebframe demos

- Download from `https://github.com/joearms/ezwebframe`.
- Unpack.
- Type `make`.
- Point Browser at `http://localhost:1456`.

