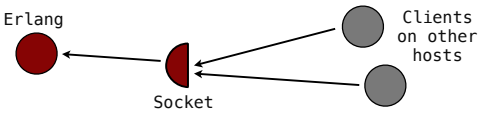
Erlang Solutions Ltd.

Ports and Sockets

Ports and Sockets

- UDP
- TCP
- Inet
- Ports

UDP



- The User Datagram Protocol is a connectionless protocol
- Provides no transmission error recovery
  - It is left up to the application to ensure packet reception and ordering
- Has very little overhead
  - It is ideal for transmissions where dropping a packet is more acceptable than waiting for it to be re-sent
- Implemented in the **gen_udp** module

UDP

```
gen_udp:open(Port) -> {ok, Port}
gen_udp:open(Port, Options) -> {ok, Port}
```

- Opens a UDP socket on a local host, used for sending and receiving
- Some options when opening sockets include
  - **list | binary** forwards messages either as lists or binaries
  - **{active, once | true | false}** for active and passive socket modes
  - **{header, Bytes}** splits the package in a list of length **Bytes** and a binary
  - **{ip, IpAddress}** specifies which network interface to use when the computer has more than one
  - **inet6** sets up the socket for IPv6

UDP

```
gen_udp:send(Socket, Port, Address, Packet)
   gen_udp:recv(Socket, Length, Timeout)
 gen_udp:controlling_process(Socket, Pid)
```

- **send(Socket, Port, Address, Packet)** allows to send a **Packet** through **Socket** to whatever is listening on **Port** at **Address**
- In passive mode, **recv/3** allows to receive a message from the socket.
- In active mode, messages are sent directly to the process in charge of the socket
- **gen_udp:controlling_process(Socket, Pid)** allows to change the process in charge of the socket.

UDP: example

```
1> {ok, Socket} = gen_udp:open(1234).
{ok,#Port<0.512>}
2> flush().
Shell got {udp,#Port<0.512>,{127,0,0,1},1235,"Hello world"}
Shell got {udp,#Port<0.512>,{127,0,0,1},1235,"Hello world"}
ok
3> gen_udp:close(Socket).
ok
```

```
1> {ok, Socket} = gen_udp:open(1235).
{ok,#Port<0.512>}
2> gen_udp:send(Socket,{127,0,0,1},1234,<<"Hello world">>).
ok
3> gen_udp:send(Socket, {127,0,0,1}, 1234, "Hello world").
ok
4> gen_udp:close(Socket).
ok
```

UDP: example

```
1> {ok,Socket}=gen_udp:open(1236,[binary,{active, false}]).
{ok,#Port<0.536>}
2> flush().
ok
3> gen_udp:recv(Socket, 0).
{ok,{{127,0,0,1},1235,<<"Hello world">>}}
4> gen_udp:recv(Socket, 0).
{ok,{{127,0,0,1},1235,<<"Hello world">>}}
5> gen_udp:recv(Socket, 0, 10).
{error,timeout}
```

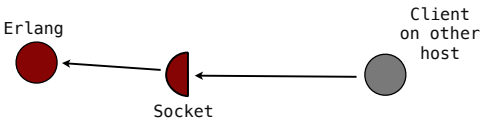
```
1> {ok, Socket} = gen_udp:open(1235).
{ok,#Port<0.539>}
2> gen_udp:send(Socket,{127,0,0,1},1236,"Hello world").
ok
3> gen_udp:send(Socket,{127,0,0,1},1236,<<"Hello world">>).
ok
```


UDP: example

```
1> {ok, Socket} = gen_udp:open(1234).
{ok,#Port<0.505>}
2> Pid = spawn(fun() ->
2>   receive Msg -> io:format("Received: ~p~n", [Msg]) end
2> end).
<0.34.0>
3> gen_udp:controlling_process(Socket, Pid).
ok
Received: {udp,#Port<0.505>,{127,0,0,1},1235,"Hello world"}
```

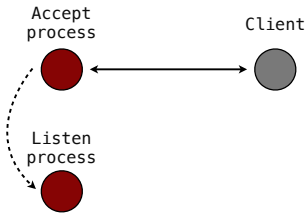
```
1> {ok, Socket} = gen_udp:open(1235).
{ok,#Port<0.539>}
2> gen_udp:send(Socket, {127,0,0,1}, 1234, "Hello world").
ok
3> gen_udp:close(Socket).
ok
```


TCP



- The Transmission Control Protocol is connection oriented
- The peers exchange streams of data
- Package reception and ordering is guaranteed
  - Used for HTTP, IM, peer-to-peer applications, mail, Erlang distribution, etc.
- Implemented in the **gen_tcp** module.

TCP



- Once a connection is set up, it remains open until either side closes it
- A TCP server then requires a process to listen to incoming requests and to handle the connection.
- The listener process can either spawn an acceptor and go back to listening, or accept the connection itself, then spawn a new listener.

TCP: clients

```
gen_tcp:connect(Address, Port, Options) -> {ok, Socket}
  gen_tcp:connect(Address, Port, Options, Timeout) ->
                        {ok,Socket}
```

- A client process is responsible for starting a connection towards the server with **connect/3-4**
- Some options when connecting to a socket include:
  - **list | binary** forwards messages either as lists or binaries.
  - **{active, once | true | false}** for active and passive socket modes
  - **{port, Port}** specifies which local port to use
  - **inet6** sets up the socket for IPv6

TCP: clients

```
client(Host, Data) ->
  {ok, Socket} =
  gen_tcp:connect(Host,1234,[]),
  send(Socket, Data),
  ok = gen_tcp:close(Socket).

send(Socket, <<Chunk:100/binary,
Rest/binary>>) ->
  gen_tcp:send(Socket, Chunk),
  send(Socket, Rest);
send(Socket, Rest) ->
    gen_tcp:send(Socket, Rest).
```

- A client process is responsible for starting a connection to the server
- A connection can be opened with **connect/3-4**
- It sends some data and closes the socket
- Data is sent with **send/2**
- The socket could have been closed by the other side.
- TCP connections can be closed with **close/1**

TCP: servers

```
gen_tcp:listen(Port, Options) -> {ok, ListenSocket}
    gen_tcp:accept(ListenSocket) -> {ok, Socket}
gen_tcp:accept(ListenSocket, Timeout) -> {ok, Socket}
```

- The server must set up a special socket to listen to incoming connections. It is opened by calling **gen_tcp:listen(Port, Opts)**.
- Some options when opening a listen socket include:
  - **list | binary** forwards messages either as lists or binaries.
  - **{active, once | true | false}** for active and passive socket modes
  - **{ip, IpAddress}** specifies which network interface to use when the computer has more than one.
  - **inet6** sets up the socket for IPv6
- The server can accept a connection request with **accept/1-2**. A regular socket is returned

TCP: servers

```
start() -> spawn(fun server/0).

server() ->
    {ok,ListenSocket} =
         gen_tcp:listen(1234,[binary,{active,false}]),
    spawn(?MODULE, wait_connect, [ListenSocket,0]),
    timer:sleep(infinity).
```

- The server creates a listen socket and passes it to a process in charge of listening
- The server's main loop sleeps to keep the socket alive
  - When the server terminates, the listen socket and all the related sockets close with it

TCP: servers

```
wait_connect(ListenSocket, Count) ->
  {ok, Socket} = gen_tcp:accept(ListenSocket),
  spawn(?MODULE, wait_connect, [ListenSocket, Count+1]),
  get_request(Socket, [], Count).

get_request(Socket, BinaryList, Count) ->
  case gen_tcp:recv(Socket, 0, 5000) of
    {ok, Bin} ->
      get_request(Socket, [Bin|BinaryList], Count);
    {error, closed} ->
   io:format("~p: ~p~n",[Count,lists:reverse(BinaryList)])
  end.
```

- The listening socket accepts the connection, then spawns a new process to become the listener in its place.
- The process handles the socket request.

TCP: example

```
1> tcp:start().
<0.35.0>
2> tcp:client({127,0,0,1},
<<"Hello, Concurrent World!">>).
ok
0: [<<"Hello, Concurrent
World!">>]
3> tcp:client({127,0,0,1},
<<"Another process handles
this">>).
ok
1: [<<"Another process handles
this">>]
```

- The TCP server is started and listens on port 1234
- The client establishes a connection with the listening process
- The listening process accepts the connection and becomes an accepting process
- A new listening process is spawned and waits for the next connection

Inet

- The **inet** module contains generic functions that will work on any sockets, whether you are using UDP or TCP.
- Allows to change socket options after they were started with **setopts(Socket, OptionList)**
- Lets you find what options were given to a socket with **getopts(Socket, Options)**
- Can retrieve statistics about a socket with **getstat(Socket)**
- Can list all currently active sockets with **i()**

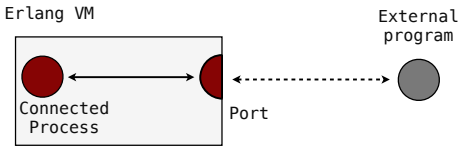
Inet: example

```
1> {ok,Sock} = gen_udp:open(1234).
{ok,#Port<0.528>}
2> inet:getopts(Sock,[active,header,broadcast,keepalive]).
{ok,[{active,true},{header,0},{broadcast,false},
{keepalive,false}]}
3> inet:setopts(Sock, [{active, once}]),
3> inet:getopts(Sock, [active]).
{ok,[{active,once}]
4> gen_udp:send(Sock, {127,0,0,1}, 1234, "Hi").
ok
5> inet:getopts(Sock, [active]).
{ok,[{active,false}]}
6> flush().
Shell got {udp,#Port<0.528>,{127,0,0,1},1234,"Hi"}
```

Inet: example

```
1> {ok, Socket1} = gen_udp:open(1234),
1> {ok, Socket2} = gen_udp:open(1235).
{ok,#Port<0.511>}
2> gen_udp:send(Socket1, {127,0,0,1}, 1235, <<"test">>),
2> gen_udp:send(Socket1, {127,0,0,1}, 1235, "test 2").
ok
3> inet:i().
Port Module    Recv Sent Owner    Local Address   Foreign Address State
505  inet_udp 0    10   <0.31.0> *:search-agent  *:*             BOUND
511  inet_udp 10   0    <0.31.0> *:mosaicsyssvc1 *:*             BOUND
4> inet:getstat(Socket1).
{ok,[{recv_oct,0},{recv_cnt,0},{recv_max,0},{recv_avg,0},
     {recv_dvi,0},{send_oct,10},{send_cnt,2},{send_max,6},
     {send_avg,5},{send_pend,0}]}
```


Ports



- Ports allow communication between an Erlang node and an external program
- The communication with the external program can be done through binary messages or standard input and output
- Ports act like Erlang processes that do not trap exits: they can be linked to, send and receive messages and receive exit signals
- Port functions are implemented in the **erlang** module

Ports

```
open_port({spawn, Cmd}, Options) -> Port
```

- **Cmd** will be run as an external program
- Some options when opening a port include:
  - **stream | {packet, N}** gives the size of binary packets to be used for this port. **N** can be 1, 2 or 4. If the packet size is variable, use stream.
  - **binary** all I/O from the port comprises data objects instead of bytes
  - **use_stdio** this uses the (Unix) standard input and output for communications. Use **nouse_stdio** to avoid this.
  - **exit_status** ensures that a message is sent to the port when the external program exits.

Ports

- The function **port_connect/2** allows to change the owner of the port. The previous owner remains linked to the port.
- **port_command(Port, Data)** is used to send **Data** to the **Port**
- Information about the port can be retrieved with **port_info/1**
- Ports can be closed with **port_close(Port)** when called from the controlling process.

Ports: example (C program)

```
/* echo.c */
#include <stdio.h>
#define BUFFER_LENGTH 80
int main() {
    char line[BUFFER_LENGTH];
    while (1) {
        if (fgets(line,
BUFFER_LENGTH, stdin) != NULL) {
            printf("%s", line);
            printf("ECHOED\n");
            fflush(stdout);
        } else {
            return 0;
        }
    }
}
```

- This program is an echo server that repeats what is sent on standard input, followed by "ECHOED".
- The Port has to make sure standard input is open (!= NULL), otherwise the port program will remain alive even after the Erlang VM terminates
- The output buffers need to be flushed, otherwise the program might not reply.

Ports: example (Erlang program)

```
-module(echo).
-export([start/0, stop/1,
repeat/2]).

start() -> open_port({spawn, "./
echo.o"}, [stream, {line, 80}]).

stop(Port) -> port_close(Port).

repeat(Port, Msg) ->
  case is_valid(Msg) of
    false->erlang:error(badarg);
    true -> ok
  end,
  port_command(Port, Msg),
  get_reply(Port).
```

- **start()** opens the port with the executable "echo.o"
- **stop(Port)** closes it
- **repeat(Port, Msg)** takes care of validating the message, sending it to the port and returning the port's reply.

Ports: example (Erlang program)

```
is_valid(Msg) when length(Msg) =< 80 ->
    [Last|_] = lists:reverse(Msg),
    Total = length([1 || $\n <- Msg]),
    if Last =:= $\n, Total =:= 1 -> true;
       Last =/= $\n; Total =/= 1 -> false
    end;

is_valid(_) -> false.
```

- The port program is line-based. All messages coming from Erlang need to end with a line break
- Only one line break is allowed per message
- A line is 80 characters long at most.

Ports: example (Erlang program)

```
get_reply(Port) -> get_reply(Port, []).
get_reply(Port, Acc) ->
 receive
  {Port, {data, {eol, "ECHOED"}}} ->
     {ok, lists:flatten(lists:reverse(Acc))};
  {Port, {data,{eol, Line}}} ->
     get_reply(Port,[Line|Acc]);
  {Port,{data,{noeol, Txt}}} ->
     get_reply(Port, [Txt|Acc])
 after 5000 ->
     {error, timeout, Acc}
 end.
```

- The port replies with messages of the form **{Port, {data, Data}}**
- Assumes the port is sending data until it ends a line (**eol**) with "ECHOED"

Ports: example (Erlang program)

```
1> os:cmd("gcc echo.c -o echo.o"),
1> Port = echo:start().
#Port<0.510>
2> echo:repeat(Port, "Hello, port!\n").
{ok,"Hello, port!"}
3> echo:repeat(Port, "How are you?\n").
{ok,"How are you?"}
4> erlang:port_info(Port).
[{name,"./echo.o"}, {links,[<0.31.0>]}, {id,510},
connected,<0.31.0>}, {input,40}, {output,26}]
5> echo:stop(Port).
true
6> erlang:port_info(Port).
undefined
```


Ports and Sockets

- UDP
- TCP
- Inet
- Ports