

TCP Sockets in Erlang

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho



Client:

```
connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}
```

Server:

```
listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}
```

```
accept(ListenSocket) -> {ok, Socket} | {error, Reason}
```

Send / receive:

```
send(Socket, Packet) -> ok | {error, Reason}
```

```
recv(Socket, Length) -> {ok, Packet} | {error, Reason}
```

- data can be received as lists (default) or `binary`;
- *iodata* can be sent: lists of bytes, binaries, or list of *iodata*;
- `recv` only used in passive mode (below);



- Framing:

`{packet, PacketType}`

- `raw` | `0`: no framing
- `1` | `2` | `4`: header with 1, 2 or 4 bytes length, big-endian; generated at send, stripped at receive;
- `asn1` | `cdr` | `sunrm` | `fcgi` | `tpkt` | `line`: receive-side framing; e.g., `line` for line-oriented protocols;
- `http`: for http server;

- Delivery to processes:

`{active, true | false | once | N}`

- `true`: what arrives to socket delivered as Erlang messages; dangerous: no flow-control; default;
- `false`: to be used with explicit `recv`
- `once`: delivers one message; must be rearmed (`inet:setopts`); best of both worlds: allows flow-control and selection;



Example: chat server in Erlang

```
-module(chat) .  
-export([server/1]) .
```

server(Port) ->

```
Room = spawn(fun() -> room([]) end),  
{ok, LSock} = gen_tcp:listen(Port, [binary, {packet, line}]),  
acceptor(LSock, Room) .
```

acceptor(LSock, Room) ->

```
{ok, Sock} = gen_tcp:accept(LSock),  
Room ! {new_user, Sock},  
gen_tcp:controlling_process(Sock, Room),  
acceptor(LSock, Room) .
```

- Two processes: room and acceptor;
- Line based framing; lines converted to messages;
- Room designated target of socket messages
(controlling_process);



Example: chat server in Erlang (room)

```
room(Sockets) ->
  receive
    {new_user, Sock} ->
      io:format("new_user~n", []),
      room([Sock | Sockets]);
    {tcp, _, Data} ->
      io:format("received~p~n", [Data]),
      [gen_tcp:send(Socket, Data) || Socket <- Sockets],
      room(Sockets);
    {tcp_closed, Sock} ->
      io:format("user_disconnected~n", []),
      room(Sockets -- [Sock]);
    {tcp_error, Sock, _} ->
      io:format("tcp_error~n", []),
      room(Sockets -- [Sock])
  end.
```



Example: chat server in Erlang (V2)

```
-module(chatv2) .  
-export([server/1]) .
```

```
server(Port) ->
```

```
    Room = spawn(fun() -> room([]) end),  
    {ok, LSock} = gen_tcp:listen(Port, [binary, {packet, line}, {reuseaddr, true}]),  
    acceptor(LSock, Room) .
```

```
acceptor(LSock, Room) ->
```

```
    {ok, Sock} = gen_tcp:accept(LSock),  
    spawn(fun() -> acceptor(LSock, Room) end),  
    Room ! {enter, self()},  
    user(Sock, Room) .
```

- Processes: room, acceptor, and a process per client (user);
- User process manages client session;
- After accept, a new acceptor is created;
- Current acceptor starts managing client (becomes user);



Example: chat server in Erlang (V2, room)

```
room(Pids) ->
  receive
    {enter, Pid} ->
      io:format("user_entered~n", []),
      room([Pid | Pids]);
    {line, Data} = Msg ->
      io:format("received~p~n", [Data]),
      [Pid ! Msg || Pid <- Pids],
      room(Pids);
    {leave, Pid} ->
      io:format("user_left~n", []),
      room(Pids -- [Pid])
  end.
```

- Keeps list of processes that interact with clients;
- Does not make use of sockets;



Example: chat server in Erlang (V2, user)

```
user(Sock, Room) ->
  receive
    {line, Data} ->
      gen_tcp:send(Sock, Data),
      user(Sock, Room);
    {tcp, _, Data} ->
      Room ! {line, Data},
      user(Sock, Room);
    {tcp_closed, _} ->
      Room ! {leave, self()};
    {tcp_error, _, _} ->
      Room ! {leave, self()}
  end.
```

- User interacts with remote clients through socket, and with room;
- The only code which deals with the wire protocol;

