TCP Sockets in Erlang

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Module gen_tcp

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);

Some Options

- 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

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

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

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([start_server/1]).

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

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)

```erlang
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;

Flow control

- The default socket option `{active, true}`:
  - allows elegant handling of multiple sources through `receive`;
  - but no flow control: incoming data can overflow a slow receiver;
- An alternative `{active, false}`:
  - allows flow control by an explicit blocking `gen_tcp:recv`;
  - but blocks the process in waiting for a single source;
- Using `{active, once}` achieves the best of both:
  - allows elegant handling of multiple sources using `receive`;
  - delivers only one data message to mailbox;
  - allows flow control by the need to rearm `{active, once}`;
- `{active, N}` generalizes `{active, once}`;

Example: chat server with flow control (V3)

```
-module(chatv3).
-export([start_server/1]).

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

- Using option `{active, once}` on the listening socket, connected sockets will deliver only one data message to mailbox;
- One more data message is allowed from the socket by doing a `inet:setopts(Sock, [{active, once}])`
- Acceptor and Room unchanged from V2;

Example: chat server with flow control (V3, user)

```
user(Sock, Room) ->
  Self = self(),
  receive
    {line, {Self, Data}} ->
      inet:setopts(Sock, [{active, once}]),
      gen_tcp:send(Sock, Data),
      user(Sock, Room);
    {line, {_, Data}} ->
      gen_tcp:send(Sock, Data),
      user(Sock, Room);
    {tcp, _, Data} ->
      Room ! {line, {Self, Data}},
      user(Sock, Room);
    {tcp_closed, _} ->
      Room ! {leave, self()};
    {tcp_error, _, _} ->
      Room ! {leave, self()}
  end.
```

- User sends it own Pid in message to room;
- Rearms active once only when getting back own message;
- If server cannot keep up, it will pause reading from socket;