ERLANG programming language

Search

news articles events downloads community links documentation about

ARTICLES

home » articles » 15

Socket IO

Written by Raimo, 29 Apr 2010

Previous: File IO

The original article and examples was written by Claes Wikstrom in 1998. In april 2008, We removed an example that no longer works and revised the examples to eliminate warnings and use the modern type tests (`when is_list(List)` instead of `when list(List)`). In april 2010 we removed the GS example since that GUI library should be deprecated.

In this section we shall give a number of examples of tcp/ip socket IO. Erlang is well suited for the implementation of a number of the classical internet tcp/ip based protocols. The API to the TCP/UDP/IP etc can be found it the `gen_tcp`, `gen_udp` and the `inet` modules. All socket code written in erlang is the same regardless of the underlying operating system

We start with a simple time server, it receives requests on TCP port 2345 and replies with a string indicating the current time.

A time server

The complete code is here and it is extremely simple. It just opens a socket a replies with the current data and time to anyone that connects to the server.

We have:

```
start(Pno) ->
    spawn(?MODULE, loop0, [Pno]).

loop0(Port) ->
    case gen_tcp:listen(Port, [binary, {packet, 0}, {active, false}]) of
    {ok, LSock} ->
        loop(LSock);
    _ ->
        stop
    end.

loop(Listen) ->
    case gen_tcp:accept(Listen) of
    {ok, S} ->
        gen_tcp:send(S, io_lib:format("~p~n", [{date(), time()}])),
        gen_tcp:close(S),
        loop(Listen);
    _ ->
        loop(Listen)
    end.
```

This is is a so called iterative server. It finishes each request until it accepts a new request. Another type of TCP/IP server creates a new process for each request. This is known as a concurrent or a threaded server. As we can expect erlang is well suited for the implementation of threaded servers.

A threaded chargen server

One of the most strange servers on many UNIX machines is the `chargen` server. It typically resides on port 19 and replies with an endless stream of printable ASCII characters when connected to.

It can thus act as an indefinite source of network input, typically used to test various applications.

Nevertheless, this type of server need to be threaded. Thus we provide an example here with a threaded chargen server.

The best way to write a threaded TCP server in Erlang is to first create the listen socket, then create a new worker process which does the call to `accept`. Once the accept is complete the worker process notifies the servere about this and the server can create yet another worker process. First we need then initial startup code:

```
-define(PORTNO, 2019).

start_link() ->
    start_link(?PORTNO).
start_link(P) ->
    spawn_link(?MODULE, loop0, [P]).

loop0(Port) ->
    case gen_tcp:listen(Port, [binary, {reuseaddr, true},
                               {packet, 0}, {active, false}]) of
    {ok, LSock} ->
        spawn(?MODULE, worker, [self(), LSock]),
        loop(LSock);
    Other ->
        io:format("Can't listen to socket ~p~n", [Other])
    end.


This code spawns off the initial worker process and goes into a loop. The
loop is simple:

```

How do you want to browse the articles?

- by year
- by author
- by tag

```

loop(S) ->
    receive
    next_worker ->
        spawn_link(?MODULE, worker, [self(), S])
    end,
    loop(S).
```

This code spawns off the initial worker process and goes into a loop. The loop is simple:

```
loop(S) ->
    receive
    next_worker ->
        spawn_link(?MODULE, worker, [self(), S])
    end,
    loop(S).
```

It listens for `next_worker` messages and spawns off a new worker for each such message. This way we are guaranteed to always have exactly one worker waiting to accept on the socket. The worker process accepts the socket and start to generate characters. We have:

```
worker(Server, LS) ->
    case gen_tcp:accept(LS) of
    {ok, Socket} ->
        Server ! next_worker,
        gen_chars(Socket, 32);
    {error, Reason} ->
        Server ! next_worker,
        io:format("Can't accept socket ~p~n", [Reason])
    end.


gen_chars(Socket, Char) ->
    Line = make_line(Char, 0),
    case gen_tcp:send(Socket, Line) of
    {error, Reason} -> exit(normal);
    ok -> gen_chars(Socket, upchar(Char))
    end.


make_line(Char, 70) ->
    [10];
make_line(127, Num) ->
    make_line(32, Num);
make_line(Char, Num) ->
    [Char | make_line(Char+1, Num+1)].

upchar(Char) ->
    if
    Char + 70 > 127 ->
        32 + (127 - Char);
    true ->
        Char + 70
    end.
```

Socket server framework

The `chargen` server from the previous section exhibits a behaviour which is truly typical for many internet servers. Many many tcp/ip servers have the typical sequence of listen followed by a loop of accept calls. We can parametrize this behaviour in a manner simmilar to the `with_file/2` function from the first section. We want to have a function called `with_socket/2` which takes a Portnumber and a `Fun` as parameters and the runs a standard tcp/ip server on the port number and applies `Fun` in a separate process to all requests. This is indeed possible and the code for the `with_socket/2` function (ofcourse) resides in klib.erl. This code has some more luggage than the previous `chargen` server, for example we have client functions to query the server about it's status as so on. The code explained:

```
with_socket(Port, Fun) ->
    spawn(?MODULE, with_socket0, [Port, Fun]).

with_socket0(Port, Fun) ->
    process_flag(trap_exit, true),
    case gen_tcp:listen(Port, [binary, {packet, 0},
                    {reuseaddr, true},
                    {active, false}]) of
    {ok, LSock} ->
        P = spawn_link(?MODULE, sock_handler, [self(), Fun, LSock]),
        sock_loop(P, Fun, 1, nil, LSock);
    Other ->
        Other
    end.
```

An ftp daemon

In this section we introduce a complete implementation of an ftp daemon according to RFC 765. We don't comment on the code merely provide it. The server is almost 1000 lines of Erlang code. The code is at ftpd.erl

Previous: File IO

Tags: *[klacke_examples]*