

# Going low level with TCP sockets and :gen\_tcp

**Orestis Markou**

@orestis

orestis.gr

# About

Orestis Markou

@orestis

orestis.gr

# Why?

External hardware control

- ^ PJLink protocol for projectors

- ^ Obscure IP-powerbars

- ^ Terse official docs

- ^ Foundation of the Internet

- ^ Will make you wonder how everything even *works*

- ^ Will make you appreciate the engineers that designed it

# Internet Protocol (IP in TCP/IP)

- Like passing notes in school
- Put data in a packet, pass it on
- Hope for the best!

Packet arrived?

^ Data too big?

^ Ordering?



# Transport Control Protocol (TCP in TCP/IP)



Photo by Annie Spratt on Unsplash



# TCP/IP

- Point-to-point,
- stream
- two-way



make connection/dial phone

^ just talk / listen (write data/read data)

# BSD Sockets API

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `gethostname()`
- ...

1983

^ network as file

^ POSIX standard

^ Most common TCP/IP API

^ Implemented by your OS

^ Slightly breaks the illusion

# :gen\_tcp

- `accept/1,2`
- `close/1`
- `connect/3,4`
- `listen/2`
- `recv/2,3`
- `send/2`
- `shutdown/2`
- `controlling_process/2`

## BSD Sockets in the BEAM

^ Arcane & intricate

^ reflects the intricacies of BSD & BEAM together

^ works hand in hand with inet



# :inet

- `:inet.gethostbyname/1,2`
- `:inet.setopts/2`
- ...

setopts is where a lot of documentation lives  
^ see the various types

# Hello world, server

- Accept connections on port 4001
- Send the current datetime
- Close the connection

# Demo

example1.exs



# Server Code

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary,
                                                reuseaddr: true])

  server_handler(listen_socket)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  d = DateTime.utc_now() |> DateTime.to_string()
  :ok = :gen_tcp.send(socket, d <> "\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

# Server Code

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary,
                                                reuseaddr: true])

  server_handler(listen_socket)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  d = DateTime.utc_now() |> DateTime.to_string()
  :ok = :gen_tcp.send(socket, d <> "\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

:binary vs charlist for data

^ reuseaddr useful for demo purposes

# Server Code

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary,
                                                reuseaddr: true])

  server_handler(listen_socket)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  d = DateTime.utc_now() |> DateTime.to_string()
  :ok = :gen_tcp.send(socket, d <> "\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

Listen socket vs actual socket

^ accept is blocking

^ send does not mean send – just that the data got accepted by the OS

^ shutdown is more gentle – wait for data to get sent before closing

^ recurse for the next connection



# Client code

```
def client do
  {:ok, socket} = :gen_tcp.connect('localhost', 4001,
                                   [:binary, active: true])
  client_handler(socket)
end

def client_handler(socket) do
  receive do
    {:tcp, ^socket, data} ->
      IO.write data
      client_handler(socket)
    {:tcp_closed, ^socket} -> IO.puts "== CLOSED =="
  end
end
```

# Client code

```
def client do
  {:ok, socket} = :gen_tcp.connect('localhost', 4001,
                                   [:binary, active: true])
  client_handler(socket)
end

def client_handler(socket) do
  receive do
    {:tcp, ^socket, data} ->
      IO.write data
      client_handler(socket)
    {:tcp_closed, ^socket} -> IO.puts "== CLOSED =="
  end
end
```

Host is always a charlist (see docs for other types)

^ active mode turns incoming data into erlang messages (default)

^ active mode only relevant for receiving

# Client code

```
def client do
  {:ok, socket} = :gen_tcp.connect('localhost', 4001,
                                   [:binary, active: true])
  client_handler(socket)
end

def client_handler(socket) do
  receive do
    {:tcp, ^socket, data} ->
      IO.write data
      client_handler(socket)
    {:tcp_closed, ^socket} -> IO.puts "== CLOSED =="
  end
end
```



**One server, many clients?**

**One server, many clients?**

# Server Code

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary,
                                                reuseaddr: true])

  for _ <- 0..10, do: spawn(fn -> server_handler(listen_socket) end)
  Process.sleep(:infinity)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  :ok = :gen_tcp.send(socket, "Hello!\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

One process listens, multiple processes accept  
^ Use Ranch for this: <https://github.com/ninenines/ranch>

# Two-way stream - server

- Accept connections
- Send "HELLO?"
- Wait for name
- Send "Hello, <name>!"
- Close the connection



# Two-way stream - client

- Connect
- Wait for "HELLO?"
- Send name
- Read all data
- Wait until connection is closed

# Two way - demo

example2b.exs

# Two-way - server

```
def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  :ok = :gen_tcp.send(socket, "HELLO?")
  receive do
    {:tcp, ^socket, data} ->
      :ok = :gen_tcp.send(socket, "Hello, #{data}!\r\n")
  end
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

two way communications

# Two-way - client

```
def client_handler(socket) do
  receive do
    {:tcp, ^socket, "HELLO?"} ->
      d = IO.gets("Enter your name: ") |> String.trim()
      :ok = :gen_tcp.send(socket, d)
      client_handler(socket)
    {:tcp, ^socket, data} ->
      IO.write data
      client_handler(socket)
    {:tcp_closed, ^socket} -> IO.puts "== CLOSED =="
  end
end
```

# Two-way - client

```
def client_handler(socket) do
  receive do
    {:tcp, ^socket, "HELLO?"} ->
      d = IO.gets("Enter your name: ") |> String.trim()
      :ok = :gen_tcp.send(socket, d)
      client_handler(socket)
    {:tcp, ^socket, data} ->
      IO.write data
      client_handler(socket)
    {:tcp_closed, ^socket} -> IO.puts "== CLOSED =="
  end
end
```



# Passive mode

- Closer to the original BSD API
- Read/write to a "file"
- Blocking API with timeouts
- Provides back-pressure

send is the same

^ recv is changing

# Passive mode server

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary, reuseaddr: true,
                                                active: false])

  server_handler(listen_socket)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  :ok = :gen_tcp.send(socket, "HELLO?")
  {:ok, data} = :gen_tcp.recv(socket, 0)
  :ok = :gen_tcp.send(socket, "Hello, #{data}!\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

# Passive mode server

```
def server do
  {:ok, listen_socket} = :gen_tcp.listen(4001, [:binary, reuseaddr: true,
                                                active: false])

  server_handler(listen_socket)
end

def server_handler(listen_socket) do
  {:ok, socket} = :gen_tcp.accept(listen_socket)
  :ok = :gen_tcp.send(socket, "HELLO?")
  {:ok, data} = :gen_tcp.recv(socket, 0, 5000)
  :ok = :gen_tcp.send(socket, "Hello, #{data}!\r\n")
  :ok = :gen_tcp.shutdown(socket, :read_write)
  server_handler(listen_socket)
end
```

active: false (default is true even for servers)  
^ read zero bytes, blocking call, timeout  
5000 millis

# Passive mode client

```
def client do
  {:ok, socket} = :gen_tcp.connect('localhost', 4001,
    [:binary, active: false])
  client_handler(socket)
end

def client_handler(socket) do
  case :gen_tcp.recv(socket, 0, 5000) do
    {:ok, "HELLO?"} ->
      d = IO.gets("Enter your name: ") |> String.trim()
      :ok = :gen_tcp.send(socket, d)
      client_handler(socket)
    {:ok, data} ->
      IO.write data
      client_handler(socket)
    {:error, :closed} -> IO.puts "== CLOSED =="
  end
end
```

# Passive mode client

```
def client do
  {:ok, socket} = :gen_tcp.connect('localhost', 4001,
    [:binary,
     active: false])
  client_handler(socket)
end
```



# Passive mode client

```
def client_handler(socket) do
  case :gen_tcp.recv(socket, 0, 5000) do
    {:ok, "HELLO?"} ->
      d = IO.gets("Enter your name: ") |> String.trim()
      :ok = :gen_tcp.send(socket, d)
      client_handler(socket)
    {:ok, data} ->
      IO.write data
      client_handler(socket)
    {:error, :closed} -> IO.puts "== CLOSED =="
  end
end
```

# Passive mode

- `recv(socket, length)`
- `recv(socket, length, timeout)`
- `timeout` defaults to `:infinity`
- When `length > 0`, "read exactly `length` bytes"
- When `length == 0`, "read all available"

# Major gotchas

- *Passive:* How many bytes to read?
- *Active:* Will "HELLO?" arrive in a single message?
- **This is by design!**

Something like race condition!

^ Depends on various arcane parameters (OS/BEAM)

^ Works most of the time with tiny payloads like this

^ Will break on real-world usage

^ Another level of abstraction is needed

# Protocols

- Give shape to the data packets
- Common or niche or your own!

Some are even provided by : `gen_tcp`

# Protocol specifications

- What comes next?
- What form does it come in?
- Who is responsible for the next transmission?
- (Distributed state machine)

# Protocol specifications

e.g. Daytime protocol (RFC 867)

## **TCP Based Daytime Service**

One daytime service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 13. Once a connection is established the current date and time is sent out the connection as a ascii character string (and any data received is thrown away). The service closes the connection after sending the quote.



# Protocol specifications

e.g. HTTP/1.1 protocol (RFC 2616)

<176 pages>

e.g. Memcached protocol

<1200 lines>

# Built-in protocols

- Provided by :gen\_tcp
- Limited in scope, non-extensible
- Might be useful

give shape to the packets

^ get returned by recv

^ get sent in active mode

# Prefix header length

[packet: 2]

- Transparently add/strip header
- 1, 2 or 4 byte header length
- Support up to 2GB messages
- Very useful when you control both ends
- Use 0 for "raw" mode (default)

# Line-based messages

```
[packet: :line,  
line_delimiter: ?\n,  
packet_size: 255]
```

- Split incoming messages by newline
- Outgoing messages are your responsibility
- A few gotchas, must evaluate

unfortunately can't set CRLF as delimiter  
^ might not be as bullet proof as needed

# Mutable sockets

- Can change mode on-the-fly (binary, active)
- Active mode can be one shot or N-shot or permanent
- Can change protocols on the fly
- Read a line, extract content length, read raw bytes

# Demo

protocols.exs

**/usr/local/opt/memcached/bin/memcached**



```
def memcached_client_get do
  {:ok, socket} = :gen_tcp.connect('localhost', 11211,
    [:binary, active: false,
     packet: :line])
  :gen_tcp.send(socket, "get elixirconf\r\n")
  {:ok, response} = :gen_tcp.recv(socket, 0, 5000)
  IO.puts "Raw response:"
  IO.inspect response
  <<"VALUE elixirconf ", resp::binary>> = response
  [_, length] = resp |> String.trim() |> String.split()
    |> Enum.map(&String.to_integer/1)
  :inet.setopts(socket, [packet: 0])
  {:ok, data} = :gen_tcp.recv(socket, length, 5000)
  IO.puts "Actual data:"
  IO.inspect data
end
```

```
def memcached_client_get do
  {:ok, socket} = :gen_tcp.connect('localhost', 11211,
    [:binary, active: false,
      packet: :line])
  :gen_tcp.send(socket, "get elixirconf\r\n")
  {:ok, response} = :gen_tcp.recv(socket, 0, 5000)
  IO.puts "Raw response:"
  IO.inspect response
  <<"VALUE elixirconf ", resp::binary>> = response
  [_, length] = resp |> String.trim() |> String.split()
    |> Enum.map(&String.to_integer/1)
  :inet.setopts(socket, [packet: 0])
  {:ok, data} = :gen_tcp.recv(socket, length, 5000)
  IO.puts "Actual data:"
  IO.inspect data
end
```

# **Pain point: Untangle the protocol logic from the socket logic**

- Abstract the "transport" out
- Provide a dummy transport for testing
- Transparently adapt to TLS/SSL, tunnels etc.
- Timeout handling

# Resources

- TCP/IP Illustrated, Volume 1 [Fall & Stevens]
- [http://erlang.org/doc/man/gen\\_tcp.html](http://erlang.org/doc/man/gen_tcp.html)
- <http://erlang.org/doc/man/inet.html>
- <https://ninenines.eu/docs/en/ranch/1.4/guide/>
- [https://github.com/orestis/elixir\\_tcp](https://github.com/orestis/elixir_tcp)

# Thank you!

## Questions?

[\*\*https://github.com/orestis/elixir\\_tcp\*\*](https://github.com/orestis/elixir_tcp)

**Orestis Markou**

@orestis

orestis.gr