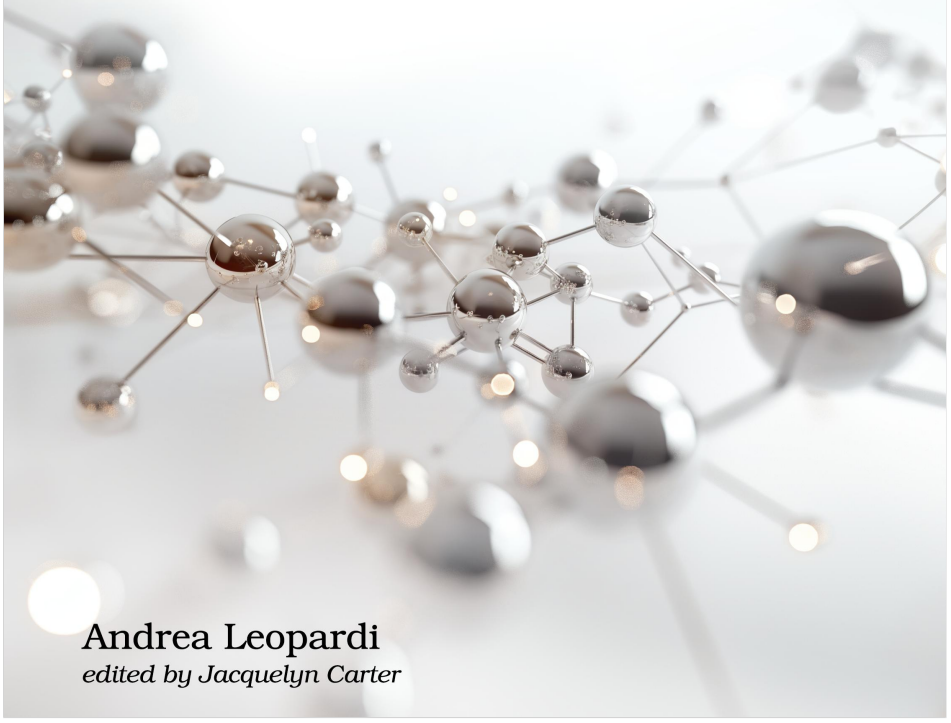
The Pragmatic Programmers

Your Elixir Source

# Network Programming in Elixir and Erlang

## Write High-Performance, Scalable, and Reliable Apps with TCP and UDP



Andrea Leopardi
*edited by Jacquelyn Carter*

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

*I'd love to make a joke about UDP, but I'm afraid you wouldn't get it.*

➤ *Someone on the Internet*

CHAPTER 8

# Same Layer, Different Protocol: Introducing UDP

TCP is the right choice for most applications. After all, the backbone of everything we do on the Internet is TCP: the web, instant messaging, streaming, downloads, and whatnot. But there are times when we need more control. Maybe you just need to get as "close to the network" as you can. Or maybe you don't need persistent connections, or network packets to be always delivered in order (or at all). That's when we reach for UDP.

*UDP* (*User Datagram Protocol*) is a simple and efficient network protocol that is often used as the foundation of more complex applications. TCP is like a cake mix: you throw in a few wet ingredients like eggs and milk, and you've got yourself a nice cake. It's hard to mess up the process, as the cake mix gives you nice "guarantees". UDP is like baking completely on your own. You have more control over the quality of each ingredient and their proportions, but it requires you to do a lot more.

Reach for UDP only when you'll absolutely need to squeeze everything out of the performance of the protocol, and when you can get away with fewer guarantees. For example, video-conferencing protocols usually rely on UDP. This makes sense for that use case since missing or out of order frames can be dealt with—choppier video streams or artifacts, but you still get a usable experience for the user.

We'll start our UDP chapter with a quick review of the protocol. Then, you'll start working on a simple but nifty little application that relies on UDP. We'll build a rudimentary metric-collection system, and then we'll iterate on it, gradually adding features as we go. Let's get started.

• Click HERE to purchase this book now. discuss

## The Basics of the Protocol

In this section, you'll learn about the basics of UDP. We'll use Erlang's standard library to have two UDP peers exchange some packets, just to dip our feet in the ocean. Rather than learning about UDP from scratch, let's explore this protocol by comparing it to TCP and looking at the differences.

*UDP* stands for *User Datagram Protocol*. As a protocol, it sits at the same OSI layer as TCP, that is, the *transport layer* (layer 4). For a quick guide to the OSI model, see Appendix 1, The OSI Model, on page ?. Just like TCP, UDP is also responsible for packaging, routing, and carrying bytes across the network. It routes those packets to the right peers.

The biggest difference between UDP and TCP is that UDP is *stateless*, while TCP is *stateful*. TCP connections are persistent and represent a stateful relationship between the two peers. UDP, on the other hand, doesn't even really deal with the concept of a "connection". UDP relies on sockets and their addresses to send and receive data. When you *open* a UDP socket, all you're doing is getting a handle on the UDP address and port combination. To send data to another UDP socket, you just have to specify its address and port combination. This all sounds a bit abstract, but we'll get our hands dirty soon. First, let's look at quick recap of the differences between the two protocols, in the next table.

| TCP | UDP |
|---|---|
| Stateful connection | No connection |
| Packets cannot get lost | No delivery guarantees |
| Packets are always delivered in order | No order guarantees |

One key characteristic of UDP, which is a product of its statelessness, is that there are just about *no guarantees*. At all. You can't know whether your socket is connected or not. You can't know if the data you send reaches its destination. If the data you send makes it to its destination, you can't assume that it reached it *in the order it was sent*. But there's a reason for all this: *efficiency*. Thanks to the lack of any handshake to initiate connections, and thanks to not having to keep track of sent and received packets—for ordering purposes—UDP stays fast and lightweight. Like most things in the software world, you give something up (guarantees) to get something back (efficiency)—it's all about compromises and use cases. We'll explore how to deal with this lack of guarantees throughout this entire chapter. For now, let's dive in and look at some UDP in action.

• Click HERE to purchase this book now. discuss

## Sending Some Data via UDP

Start off by hopping into a terminal and starting a simple UDP server through netcat,[1] a networking utility that ships with most Unix-based operating systems. The command shown next starts a UDP *echo server*, that is, a server which receives packets and prints them back on the terminal.

```
> nc -u -l 9001
```

The `-u` flag tells netcat to use UDP instead of TCP (TCP is the default). `-l` tells it to *listen* for packets (you can also use netcat as a client). Lastly, `9001` specifies the port we want to listen on. You won't see any output when you run this command, as this server hasn't received any data yet. Let's fix that by sending it some data from the BEAM. Fire up an IEx session and send the bytes `"hello"` to the netcat server using the commands shown next.

```
iex> {:ok, socket} = :gen_udp.open(9002, mode: :binary)
iex> :gen_udp.send(socket, ~c"localhost", 9001, "hello")
:ok
```

If you kept an eye on the running `nc` command, you should have seen the string `hello` printed on the terminal. So, something must be working!


Let's unpack what we just saw. It's our first encounter with the `gen_udp` module.[2] As you might assume, it's the UDP counterpart to the `gen_tcp` module that we've used extensively in the previous chapters. It also ships with the Erlang standard library, and exposes an API that closely resembles the `gen_tcp` one.

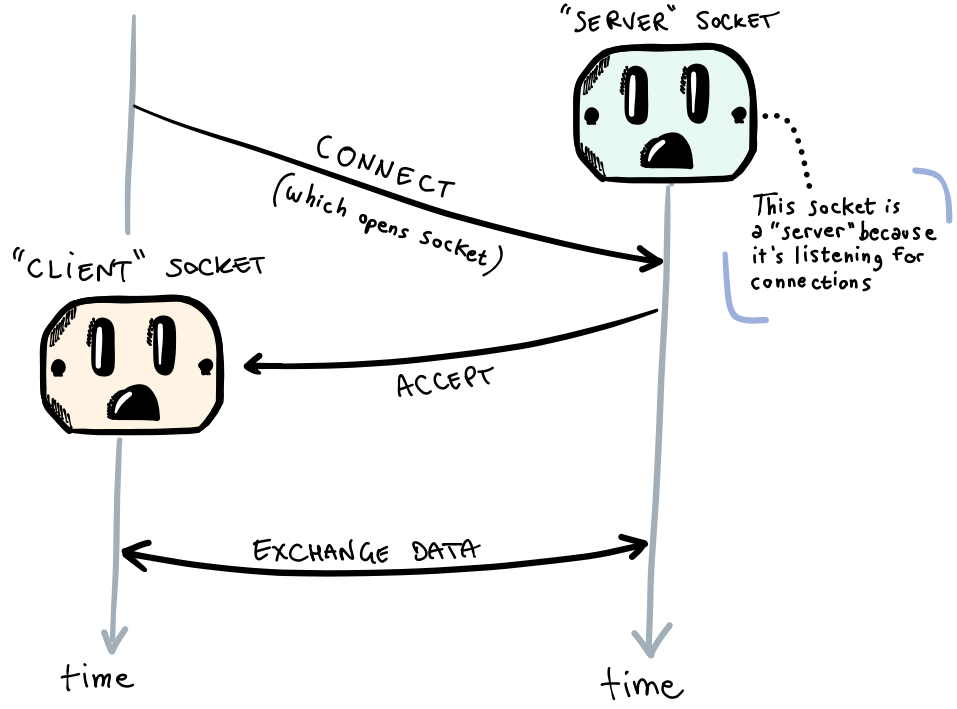
The first function we used is `:gen_udp.open/1`. This function returns a *UDP socket*. There's an important difference between UDP sockets and TPC sockets.

1. https://en.wikipedia.org/wiki/Netcat
2. https://www.erlang.org/doc/man/gen_udp.html

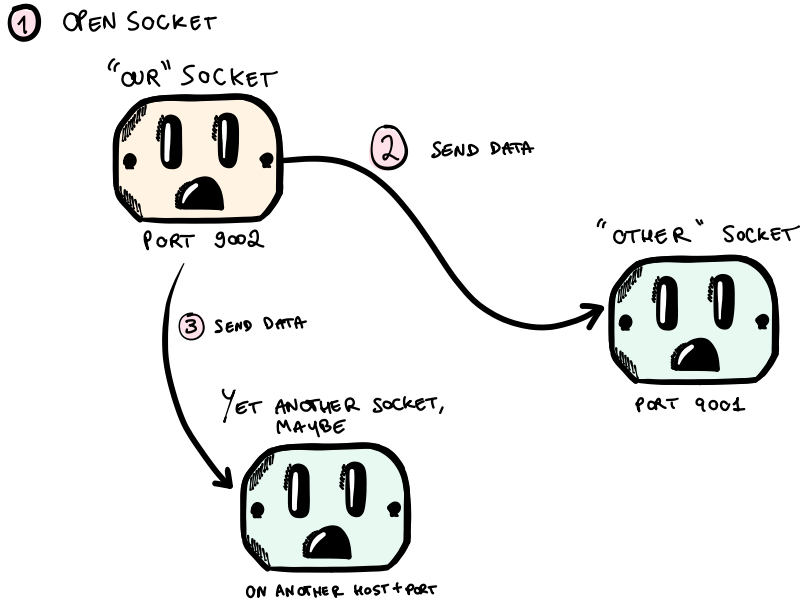
In TCP, a socket needs a peer to connect to. In UDP, however, a socket lives a life of its own, as there is no concept of "established connection". When you open a UDP socket, what you're really doing is *hooking* into the OS UDP socket listening on the given port. Here, we chose port `9002`. `:gen_udp.open/2` is the equivalent of `:gen_tcp.connect/4`, but the name difference is a clear indication of what we just talked about.

We can send UDP data to another UDP socket *through the socket* we opened. That's what we do with the call to `:gen_udp.send/4`. As you can see in the IEx session, we specify a host-and-port combination right in the `send/4` call, without the need for connecting first. If we wanted to send data to *another UDP peer*, we could do it through the *same* `socket`. All we'd need to do is use a different host-and-port combination. This is not possible in TCP, where a specific socket is permanently connected to a specific peer.

The next two figures show the difference between TCP and UDP when it comes to the flow of opening a socket and sending data. First, in the next figure, you can see the TCP flow that you know by now: the client initiates the connection, the server accepts the connection, and the new socket is opened and can be used to exchange data.



Compare that with the next figure, which shows what you can do with UDP sockets. The first step is to open the UDP socket on port 9002. That's also the *only* step when it comes to opening the socket! Once that socket is open, we can use it to *send data* to any UDP socket on any other peer. The numbers in the next figure show you the order of operations, but the only one that really matters is that opening the socket comes first.




## Receiving Data

There's even more to our humble UDP socket. It doesn't just send data, it also *receives data*. To see that in action, let's switch the role of netcat, and use it to send some data to our open Erlang socket. Leave the previous IEx session running, and exit out of the `nc` command that you used before—a `Ctrl-c` will do. Then, run the following command to open a netcat session where you'll be able to send data to our UDP socket, and type the words `hello` and `world`, both followed by `Return`.

```
> nc -u 127.0.0.1 9002
hello
world
```

We just sent the strings `hello\n` and `world\n` to our Erlang UDP socket. In the IEx session, you can use the `flush()` helper[3] to display the messages received

3. https://hexdocs.pm/iex/IEx.Helpers.html#flush/0

by the shell process. We opened the UDP socket with :gen_udp.open/1, which defaults to a socket in *active mode* (see Active and Passive Modes for Sockets, on page ?): that's why the shell process gets the data delivered as Erlang messages.

```
iex> flush()
{:udp, #Port<0.3>, {127, 0, 0, 1}, 63827, "hello\n"}
{:udp, #Port<0.3>, {127, 0, 0, 1}, 63827, "world\n"}
```

The #Port<...> number might be different for you, as might the fourth element of the tuple, which is the *source port*. The messages you see here look somewhat like the {:tcp, socket, data} messages we got used to in previous chapters. However, there's more information here. The structure of these new messages is:

```
{:udp, udp_socket, source_address, source_port, data}
```

When comparing this to the {:tcp, socket, data} messages that the :gen_tcp module uses, the only two additional elements are source_address and source_port. Why include those in the message? That's because the UDP socket is *stateless* and not connected to any particular address and port. This means that it can receive data from *any other UDP socket*. You can verify that by exiting out of netcat (again, just Ctrl-c), starting it again, and sending more data.

```
> nc -u 127.0.0.1 9002
more data
```

If you check the received messages in IEx now, you'll see a new :udp message but *with a different source port*—64927 here instead of the old 63827.

```
iex> flush()
{:udp, #Port<0.3>, {127, 0, 0, 1}, 64927, "more data\n"}
```

Once again, the source_port will likely look different for you. That's just the port that the OS allocates to the running netcat command. The second element of the tuple, though, will look exactly the same as the previous messages, showing you that the UDP socket that received the data is the same.

In this section, you got a general idea of how UDP works, especially when compared against TCP. UDP is stateless and not connection oriented. Sending and receiving data works similar to how it works for TCP, but it accounts for the socket being stateless: you have to specify where to send data every time, and incoming data is "tagged" with its source peer. Now that we've got the basics of UDP down, we can move on to building something that *leverages* the protocol.