

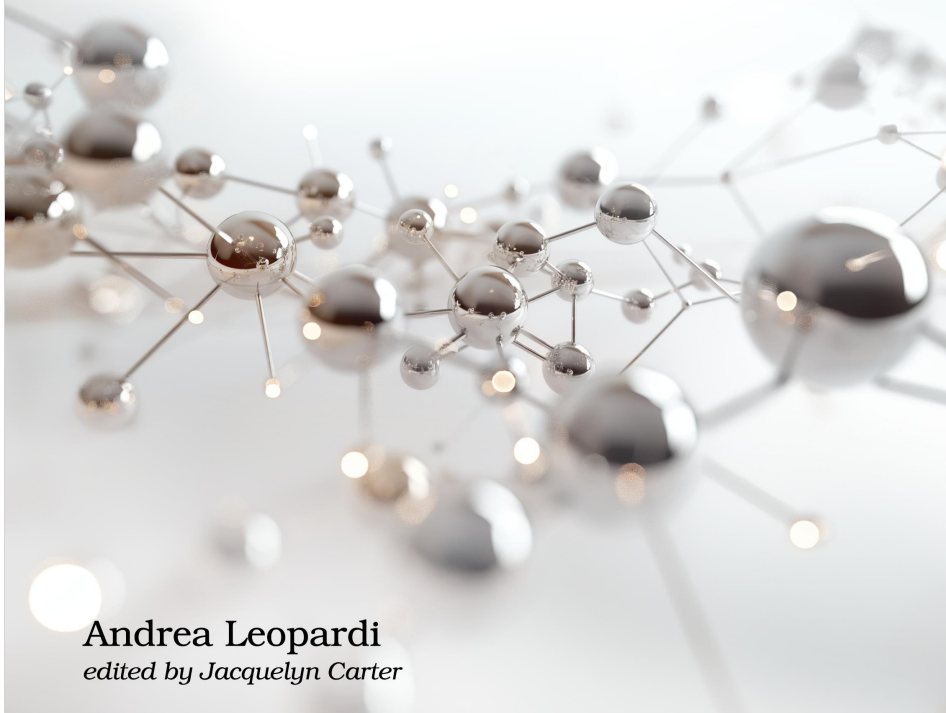
The
Pragmatic
Programmers



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.

Designing a Simple Chat Protocol

Having a well-defined protocol is a must in network systems, and you'll stumble upon a lot of these specifications when working with existing networking code. If you're curious, there are a few official protocol specifications you can check out to get an idea of what a real-world, detailed protocol spec looks like:

- The Redis Serialization Protocol (RESP) specification¹
- The spec for the HPACK header serialization protocol used in HTTP/2²
- The header format of TCP itself (section 3 of the original spec from 1981)³

In this section, we'll define our simple chat protocol and talk a bit more about protocols in general. One of the most important protocol distinctions we can make is between binary and textual protocols, so let's kick off with that.

Binary Protocols and Textual Protocols

A *binary protocol* is generally characterized by encoding information in bytes or bits, without worrying about whether messages in the protocol can be read as text by humans. For example, a protocol that uses the first byte of a message to signal the length of the rest of the message is a binary protocol. Many of that byte's possible values, such as 0, don't represent valid characters in most encodings. Real-world examples of binary protocols are HTTP/2, the protocol used by the Cassandra database,⁴ the Protobuf serialization format,⁵ and the underlying data serialization format used by TCP itself (which we talk about in [Appendix 2, TCP Protocol Details, on page ?](#)).

Textual protocols are protocols meant to be readable by machines and humans alike. Bytes of information are interpreted through a specified encoding, most commonly ASCII.⁶ One of the most famous examples of this kind of protocol is JSON.⁷ JSON is a data serialization format derived from the syntax of JavaScript objects. Here's an example of a JSON object:

```
{
  "type": "user_message",
  "contents": "Hello! My name is \"Bernaco\""
}
```

1. <https://redis.io/docs/reference/protocol-spec/>
2. <https://http2.github.io/compression-spec/compression-spec.html>
3. <https://www.ietf.org/rfc/rfc793.txt>
4. https://cassandra.apache.org/_/index.html
5. <https://protobuf.dev>
6. <https://en.wikipedia.org/wiki/ASCII>
7. <https://www.json.org/json-en.html>

You can read this example because JSON objects are readable by humans. They also have elements such as braces ({}) and quotes (") that allow machines to correctly parse the data they encode. In this example, we can see how this requirement sometimes means having to escape characters (\") so that the data can be parsed. Another well-known example of a textual protocol is HTTP/1.1.

Which kind of protocol should you choose—textual or binary? As always, the best answer to this sort of question is, “It depends.” Two of the main factors to consider are performance and size.

Well-designed binary protocols are inevitably more space-efficient when serializing data, since they can use a granularity of a single bit to encode information. This is common in real-world protocols such as HTTP/2 and DNS. In general, serialization and deserialization programs are also faster and more memory-efficient when dealing with binary data, since it’s easier to include information such as the size of data in the protocol. The downside of binary protocols is that they’re mostly unintelligible to humans. For example, if you happen to intercept an HTTP/2 request and you don’t have software to help you make sense of it, chances are you’ll just see a bunch of bytes that look like gibberish.

Textual protocols, by contrast, are easy for humans to read and easy to write by hand. We’re willing to bet that many readers have written at least one JSON object by hand in their careers. But this readability comes at the expense of speed and space. For example, in JSON you have to escape some characters (adding to the size of the data) and parse whole objects to know where they end.

Endianness in Binary Protocols

Endianness^a is the order in which bytes are interpreted. For example, when the binary <<0, 1>> is interpreted from right to left (with the least significant byte on the right), these bytes encode the number 256 in base ten. This is called *big endianness*. If interpreted from left to right, however, these bytes represent the number 1 (*little endianness*).

When working with binary protocols, the specification should clearly define the endianness used in the protocol. Textual protocols generally don’t have to clarify this.

a. <https://en.wikipedia.org/wiki/Endianness>

Now, let’s figure out which type of protocol is right for our chat. A possible choice is JSON itself. In that case, we wouldn’t need to come up with the serialization layer itself but only the semantics of the protocol (such as which kinds of messages to support). But by using a binary protocol, you’ll have a

chance to learn more about how real networks are often designed. So, let's try to come up with a binary protocol that makes sense.

Specifying Our Chat Binary Protocol

When designing a protocol, start with its requirements. Messages in our binary protocol need to encode different information based on their *type*. For example, broadcasted messages have to carry information about the contents of the message, while handshake messages only have to carry the username. This already suggests that we need to attach a type to each message. We know we won't have too many kinds of messages, so we'll use a single byte to encode the message type. A byte can encode 255 different values, which will be plenty.

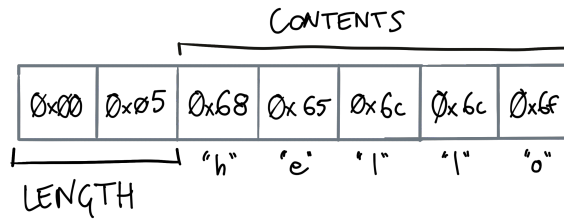
Representing Bytes

Let's talk about how to generally represent bytes. We could represent a byte with the value of its bits, in decimal base, or in hexadecimal base. The binary base tends to be hard to read for humans due to the long sequences of digits. The decimal base works, but it leads to some nasty cases: for example, the biggest single byte has value 255, which means that we cannot use all three digits up to 999.

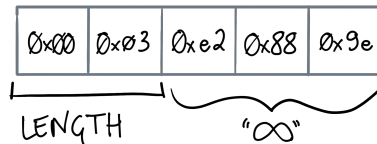
The most common representation of single bytes seems to be in *hex notation*, denoted as two digits in hexadecimal base. This works great because two hexadecimal digits can represent 8 bits (a byte). If you want to see for yourself, you can verify that 16^2 (sixteen possible values with two digits) is the same as 2^8 (two possible values with eight digits). For example, you can represent the byte 10110111 (in base two) with the hexadecimal value b7. It's common to prefix the hexadecimal value with 0x, which we'll do throughout the book. So, we would write out the byte in the previous example as 0xb7.

We'll then need to encode strings of text. A common way to do that in binary protocols is to use a few bytes to encode the length of the string, followed by the contents of the string itself. Let's go with 2 bytes to encode the length for our case. It's more than enough, since it lets us encode strings of up to 65536 bytes. We also need to specify the encoding of the string's contents—otherwise, the bytes won't mean anything. We'll go with UTF-8,⁸ the most common encoding standard and one that Elixir supports well. To get an idea of how we'd encode a string, "hello" would be encoded as in the following image:

8. <https://en.wikipedia.org/wiki/UTF-8>



This encoding also works well for strings with Unicode code points that span more than one byte. For example, the character ∞ takes a whopping 3 bytes to be represented (as 0xe2 0x88 0x9e), as you can see in the following image:



In binary protocols, some messages are unidirectional (client-to-server or server-to-client), while others are bidirectional, meaning that they can be sent and interpreted by both clients and servers. We'll have one of each.

Getting the Byte Size and Hex Representation of a Binary

You can use the `Kernel.byte_size/1` function in Elixir or `erlang:byte_size/1` function in Erlang to get the number of bytes inside a binary, regardless of how the binary is supposed to be interpreted. This is useful when working with Unicode strings, which is how Elixir represents strings by default. For example, in Elixir, `byte_size("∞")` returns 3. In Erlang, it would be `byte_size("<<"∞"/utf8>>)`.

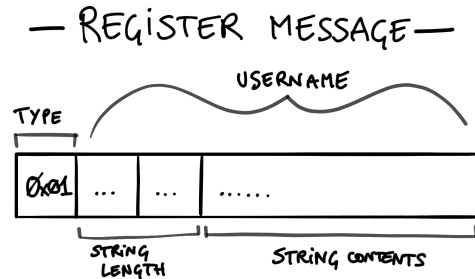


Now, say you have a binary and want to get its byte-by-byte hex representation. In Elixir, you can use the `base: :hex` option when inspecting the binary. For example, `inspect("∞", base: :hex)` will return the string `"<<0xE2, 0x88, 0x9E>>"`. This is only available in Elixir. In Erlang, you'll have to write some custom code. You can find ideas on how to do that on the Internet.⁹

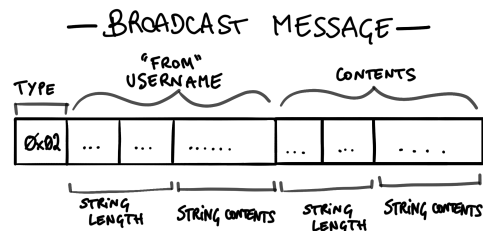
Now that we have the building blocks for our binary protocols, let's go ahead and define each type of message that clients can send to servers. We'll start with the *register message*. This is a unidirectional message that clients *must* send as the first message after establishing a connection. When the server receives this type of message, it must identify the connection through the specified username. Usernames must be unique across connected clients.

9. <https://stackoverflow.com/questions/3768197/erlang-ioformatting-a-binary-to-hex>

The type byte for this message has value 0x01. Following that, we have a single string containing the username. Here's a visual representation.



Next up are *broadcast messages*. These are bidirectional messages. Their type byte has value 0x02, followed by a string representing the *from* username of the message, and then one more string with the contents of the message itself. When a client sends a broadcast message to the server, *from_username* must be an empty string—the connection already stores the client's username, so including it would be redundant. When the server sends a broadcast message to clients, *from_username* will be the username of the sender of the broadcast message or, if the broadcast message comes from the server itself, an empty string. Keeping the *from_username* field empty sometimes lets us reuse it as a bidirectional message type instead of creating two separate types. The visual representation of the message looks like this:



We got our messages down. The image [shown on page 8](#) shows you a visual representation of an example session, in which a client connects, registers, and then broadcasts a message.

Okay, we have designed our first protocol. We started by looking at the difference between textual and binary protocols. We then came up with a binary protocol for our chat server. Our binary protocol is simple, but that's a good thing! It means it will be easy to reason about and implement. Let's move on to writing code for our protocol.

