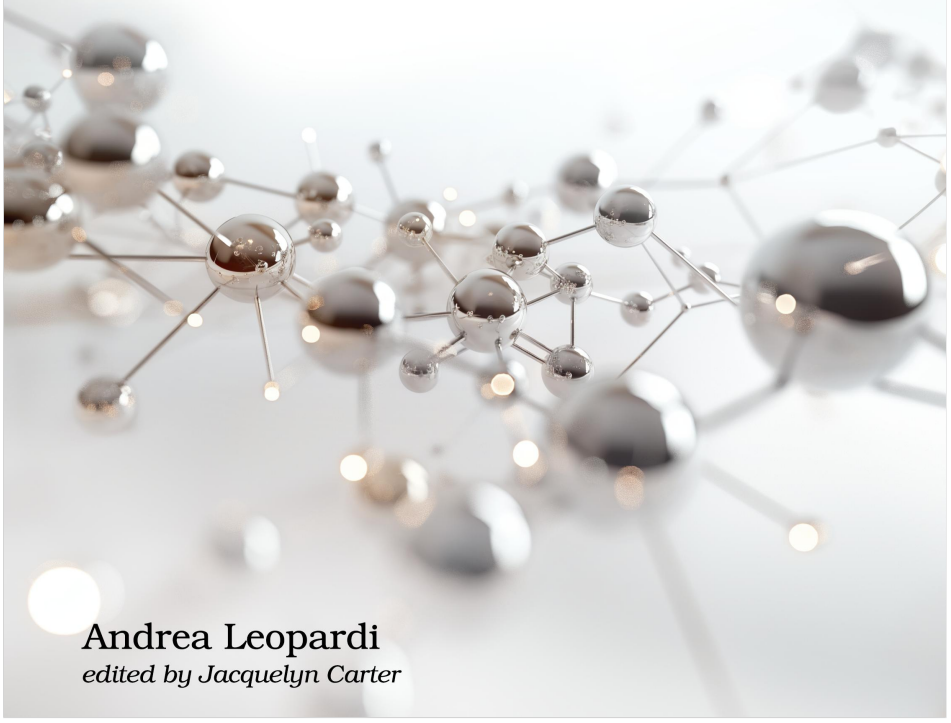
The Pragmatic Programmers

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.

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 sorts of specifications when working with existing networking code. If you're curious, there are a few "official" protocol specifications that you can check out to get an idea of what a real-world, detailed protocol spec looks like:

- The Redis Serialization Protocol (RESP) specification[1]
- The spec for the HPACK header-serialization protocol[2] used in HTTP/2
- The header format of TCP itself (section 3 of the original spec[3] from 1981)

In this section, we'll come up with our simple chat protocol and talk a bit more about protocols in general. One of the most important distinction we can make between protocols is the one 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 paying attention to 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. The reason is that many possible values of that byte, 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] or the underlying data serialization format used by TCP itself (which we talk about in Appendix 3, TCP Protocol Details, on page ?).

Textual protocols are protocols that are 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 such a protocol is JSON.[7] JSON is a data serialization format that comes from the syntax of JavaScript objects. Here's an example of a JSON object:

```
{
  "type": "user_message",
```

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

```
  "contents": "Hello! My name is \"Bernaco\""
}
```

You can read this example because JSON objects are readable by humans. They also have elements, such as brackets ({}) and quotes ("), that allow machines to correctly parse the data that they encode. In this example, we can see how this requirement sometimes leads to having to *escape* characters in order for the data to remain possible to parse (\"). 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 we can give to this sort of question is: "it depends." One of the main factors to take into consideration is 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 quite common in real-world protocols, such as HTTP/2 or DNS. In general, serialization and deserialization programs are also faster and more memory efficient when dealing with binary data, since it's easier to make information such as the size of data part of the protocol itself. The downside of binary protocols is that they're mostly unintelligible for 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 be looking at a bunch of gibberish-looking bytes.

Textual protocols, in 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 throughout their careers. This readability comes at the expense of speed and space efficiency. For example, in JSON you have to escape some characters (adding to the size of the data) and have to parse whole objects in order to know where that object ends.

Endianness in Binary Protocols

Endianness[a] is the order in which bytes are interpreted. For example, take the binary `<<0, 1>>`. When interpreted from right to left (least significant byte on the right), these bytes encode the number `256` in base ten. This is called "big endianness". However, if interpreted from left to right, these bytes represent the number `1` ("little endianness").

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

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

Now, let's figure out which type of protocol is the right choice for our chat. A possible choice would be JSON itself. It would mean we don't need to come up with the serialization layer itself, but only with the semantics of the protocol (such as which kinds of messages to support). However, by using a binary protocol you'll have a chance to learn more about how networks often operate and are 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 from the requirements of the protocol. 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 seems to suggest that we need to attach a type to each message. We know we're not going to have too many different kinds of messages, so we'll use a single byte to encode the message type. A byte can encode 255 different values, so we'll have plenty to use.

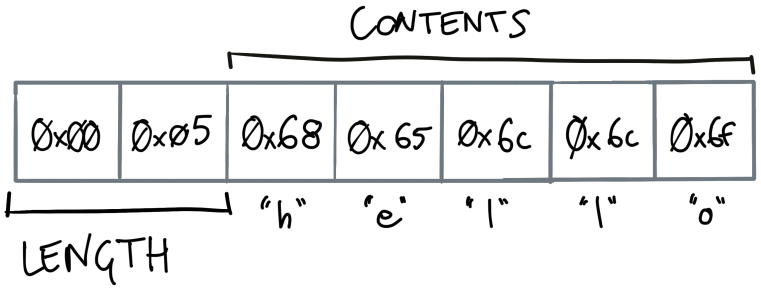
Representing Bytes

Let's talk about how to generally represent bytes, since that's an important thing to do when discussing binary protocols. Some possibilities are representing 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 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 the three digits (which would go up to 999).

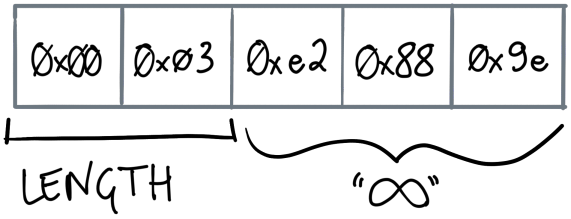
It seems that the most common representation of single bytes is the *hex notation*, denoted as two digits in hexadecimal base. That works out great, because two hexadecimal digits can represent the same values as eight bits (a byte). If you want to make sure of that, you can verify that 16^2 (sixteen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

Then, we'll 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 two bytes to encode the length in 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 *contents* of the string itself, otherwise they'd just be a bunch of bytes without meaning. We'll go

with UTF-8,[8] the most common standard for encoding and one that Elixir supports well. To get an idea of how we'd encode a string, the string "hello" would be encoded as in the following image.



This encoding works well for strings with Unicode codepoints that span more than one byte as well. For example, the character ∞ takes a whopping *three 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, you would do byte_size(<<"∞"/utf8>>).

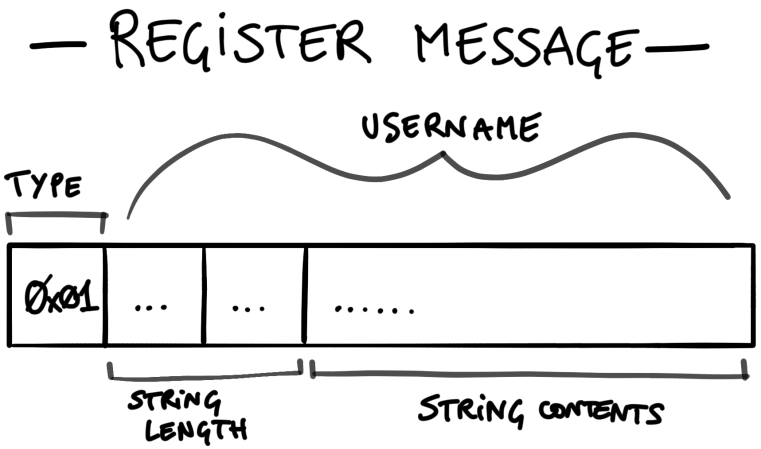
Now, say you have a binary and want to get its byte-by-byte hex representation. In Elixir, you can use the base: :hex option when inspecting the binary. For example, inspect("∞", base: :hex) will return the string "<<0xE2, 0x88, 0x9E>>". This is only available for Elixir. In

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

Getting the Byte Size and Hex Representation of a Binary

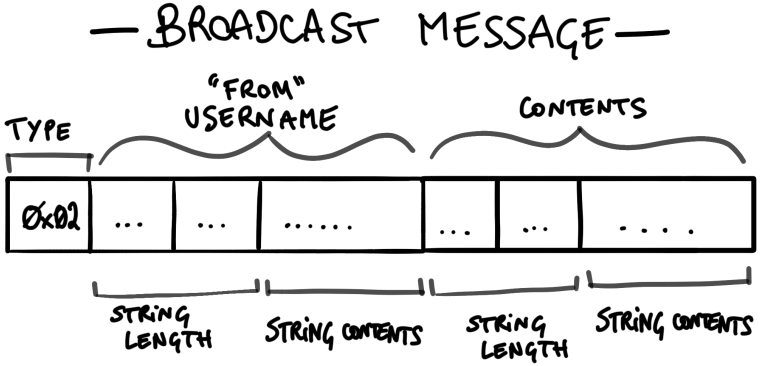
Erlang, you'll have to write some custom code to do that. You can find ideas on how to do that on the Internet.[9]

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 out with *register messages*. These are unidirectional messages that clients *must* send as the first message after establishing a connection. When the server receives this type of message, it must identify that connection through the specified username. Usernames must be unique across connected clients. The type byte for this message has value `0x01`. Following that, we have a single string containing the username. Here's a visual representation.

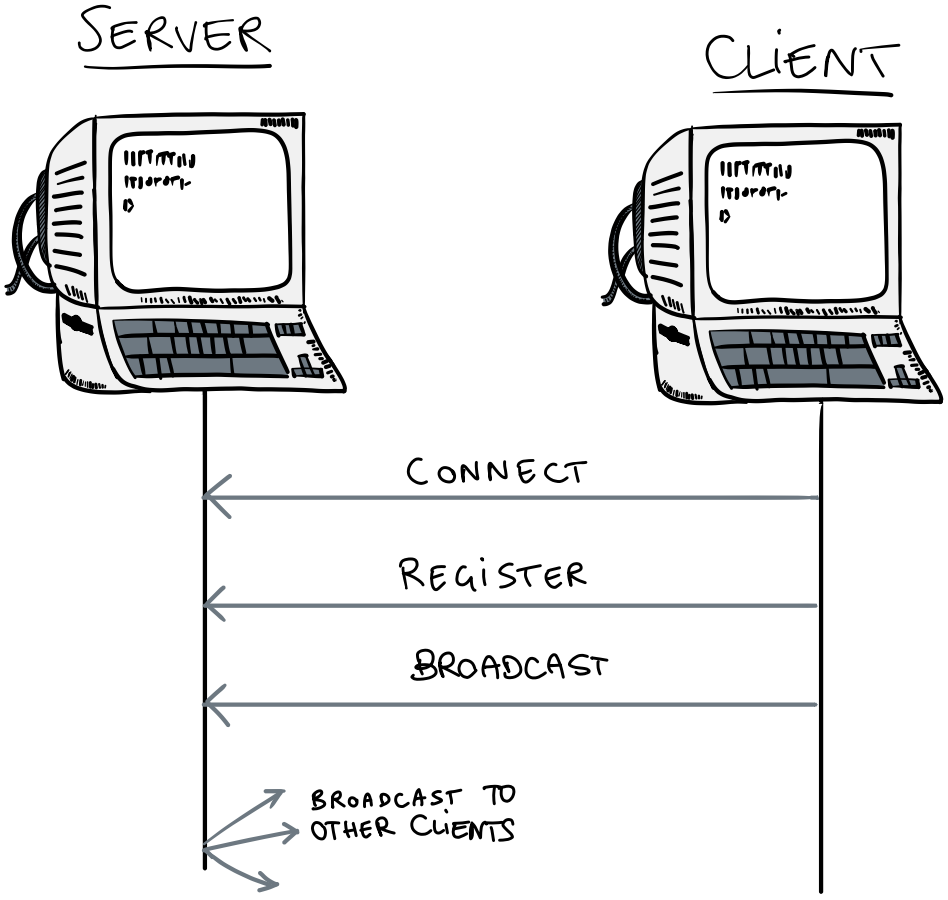


Next up we have *broadcast messages*. These are bidirectional messages. Their type byte has value `0x02`, and it's followed by a string representing the "from" username of the message, and then one more string with the contents of the message itself. When a client sends a broadcast message to the server, `from_username` must be an empty string. When the server sends a broadcast message to clients, `from_username` will be the username of the sender of the broadcast message, or an empty string if the broadcast message comes from the server itself. Keeping the `from_username` field empty sometimes lets us reuse this message type and make it bidirectional, as opposed to having to create two separate types. The visual representation of the message looks like this:

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



We got our messages down. The following image 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. Then, we came up with a binary protocol for our chat server. The binary protocol we got have here is simple, but that's a good thing! It means it will be easy to reason about and easy to implement. Let's move one to writing code for our protocol.