Advanced Network Programming III

Juliusz Chroboczek

14 March 2025

When desigining a protocol based on HTTP (a "web API"), the use of HTTP imposes a number of constraints:

- the protocol is client-server;
- the protocol obeys a strict request-response discipline;
- requests are not necessarily ordered, they may arrive at the server in an order different from the one in which they were sent (for example because they were sent on different connections in HTTP/1.1, or because of magic in HTTP/2 and /3);
- messages (requests and responses) are encoded as streams of bytes together with a set of headers.

In this lecture, we discuss usual ways of encoding data structures within streams of bytes, and usual ways of structuring a protocol based on HTTP.

1 Data encoding

HTTP bodies are streams of bytes (or octets). The stream of bytes is pervasive in networking for historical reasons (one of the main applications of the early Internet was remote access, and therefore protocols simulated a serial line), but it is almost always the wrong abstraction: a byte is way too little data to be useful, and therefore some form of encoding is necessary.

1.1 Natural encodings

In many cases, there is a natural encoding imposed by previously defined standards. For example, plain text is almost always encoded in UTF-8, and images are encoded using a number of standard encodings, such as PNG or JFIF (JPEG).

Since these encodings are expressed in terms of sequences of bytes, they are readily carried by an HTTP request or response. Note however that even in the presence of a standard encoding, some details might need to be specified by the protocol: for example, protocols that carry plain text must define whether lines are terminated with a plain newline "`\n`" or by a carriage return-line feed pair "`\r\n`".

1.2 Ad hoc encodings

In simple cases, it is expedient to define an encoding specifically for the protocol at hand. For example, in the second lab I needed to send a sequence of message identifiers, and I chose to send them as text, one identifier per line. Had I needed to send a sequence of pairs, I could have simply sent two items per line, separated by a space or a comma (as *comma separated values*, CSV).

More generally, it is easy to write a parser for an arbitrary textual format as long as the format can be defined by an LL(1) grammar.

As far as binary formats are concerned, there are many possible techniques. The one that I tend to prefer is the *type-length-value* triple, or TLV, which we will use during the project.

1.3 XML

For more complex cases, a number of generic formats have been designed. In the 1970s, S-expressions were dominant (they are still used in some old protocols). In the 1980s, SGML was designed, a framework for enriched text formats, was designed, and used for the first versions of HTML. SGML begat XML.

XML originated as an enriched text format; it is therefore overly complex and unwieldy for encoding data structures. For example, there is no standard way of encoding a list of integers (an integer array) in XML; various protocols might use

```
<list><int value="1"/><int value="2"/><int value="3"/></list>
```

or

```
<list><int>1</int><int>2</int><int>3</int></list>
```

or

```
<list>1 2 3</list>
```

or

```
<list value="1 2 3"/>
```

or even

```
<list value1="1" value2="2" value3="3"/>
```

The early 2000s were a period of collective insanity where everything was being coded in XML. Since then, a new generation of protocol designers has matured, which understands that XML's complexity is not only unnecessary, but even harmful, and XML has mostly been replaced by JSON in modern usage.

1.4 JSON

JSON is a subset of the syntax of Javascript. It is a very simple format: JavaScript values are numbers, strings, dictionaries (keyed by strings) and arrays. Its syntax is very strictly defined (for example, a comma is not allowed before a closing bracket, and comments are not allowed), which makes it simple to implement and generally interoperable.

Following JavaScript, JSON is an untyped format; for example, it is possible to encode heterogeneous arrays such as

```
[1, "two", {"three": 3}]
```

This causes some minor issues when consuming JSON data in a strictly-typed language.

JSON does not disinguish between integers and floating point numbers: JSON numbers are (finite) IEEE floating-point numbers. They can accurately represent integers of a size up to 53 bits, beyond that, overflow leads to loss of precision.

Data structures that are not directly represntable in JSON. Since JSON strings can only contain Unicode text, they are unable to represent arbitrary binary data; the usual solution is encode binary data in base 64 before storing them in a string. Similarly, JSON cannot directly encode 64-bit integers, which need to be encoded as strings, either in Base 64 or in hexadecimal. There are at least two conventions for representing dates: as JavaScript dates (integers representing a number of milliseconds since the Unix epoch), or as strings in ISO 8601 format.

1.5 Binary formats

Textual formats are easy to manipulate and make software easier to debug; however, they are less compact and costlier to parse than well-designed binary formats. Thus, in high-performance applications, it may be preferable to use a binary format.

In most cases, the TLV technique, already mentioned above, is suitable. There are of course many generic binary formats; my current favourite is CBOR, which is well-designed, compact, and can be easily repurposed to encode JSON. Alternatives include *MessagePack* and BSON.

A different approach is taken in Google's *Protocol Buffers* (Protobuf). A data structure is defined in a machine-readable format (similar to a superset of C), and the Protobuf compiler generates a serialiser and a deserialiser using a serialisation format that is automatically generated and is optimised for the specific data structure. While Protobuf yields extremely compact encodings, it generates a different encoding for every data structure; therefore, maintaining compatibility between different versions of a protocol is at best problematic.

2 Structure of web protocols

Historically, web application developers used ad hoc protocols, designed with no concern for the coherence and ease of evolution of the protocol. The protocol changed in an incompatible manner with every version of the application, which was worked around by updating the client-side (JavaScript) code in sync with the server side. Obviously, this made it difficult or impossible to develop native applications using the same protocol.

Over the years, a number of "best practices" for structuring web protocols have evolved. While we will see fairly pure examples of these structures during the labs, in practice web protocols tend to evolve organically over months or years, and therefore consist of a mixture of various structures.

2.1 XML-RPC, SOAP

A function (procedure) call can be interpreted as an exchange of two messages: a message containing the call's parameters followed with a message containing the function's result. If the messages are exchanged between different network nodes, such an exchange is called a *Remote Procedure Call* (RPC). Notable examples of protocols based on the RPC paradigm include the *Network File System* (NFS), and Microsoft's OLE (which is local only, but still follows the RPC paradigm).

In practice, RPC does not work very well. A program that makes local function calls can assume that there are no communication outages and that latencies are reasonably low; these assumptions are broken when the calls are remote, and RPC-based protocols tend therefore to be brittle and perform slowly.

In the late 1990s, RPC was implemented using XML, a protocol framework called *XML-RPC*. Since XML-RPC didn't work very well, it was extended into an inscrutably complex protocol framework called SOAP. SOAP applications manipulated complex structured messages, and used a single HTTP endpoint for all communication.

There is a rumour that there are three people in the world who understand SOAP. Nobody is quite sure who they are.

2.2 REST

The REST framework was developed in reaction to the perceived complexity of SOAP-like protocols. In REST, instead of requests that perform an application-specific action (for example, print a document), there are requests that manipulate the state of a remote data structure (for example, create a print job). The main properties of REST protocols are:

- every remote resource is represented by a different HTTP endpoint;
- there is no session state (per-client state) on the server;
- compound data structures are avoided, they are represented by multiple related resources;
- operations on resources are represented by standard HTTP methods (`GET`, `PUT`, etc.);
- relations between resources are represented as hyperlinks.

The HTTP methods are used as follows:

- `GET` retrieves the current state of a remote resource;
- `PUT` creates a remote resource or changes its state;
- `DELETE` removes a remote resource;
- `POST` is used in two manners:
    - it creates a member of a collection, at a URL chosen by the server; and
    - it modifies a resource in a manner that will be processed by the server.

In ideal conditions, REST requests are cachable, and the `ETag` mechanism (`If-Match`, `If-None-Match`, etc.) is used for cache validation. This will be studied in detail in a future lecture.

While REST is generally a good framework for desigining web protocols, it tends to cause many HTTP requests, which is wasteful of network resources[1]. In practice, compromises need to be made, and protocols do not strictly obey the REST principles.

In addition, some distributed algorithms are not easily expressed as operations on remote resources; for example, it would be difficult to express distributed Bellman-Ford as a REST protocol.

1. A friend tells me that his Android phone consumes 4.5 MB of data per hour when idle. It is not clear whether this is due to naive usage of REST, or to the verbosity of XML. Probably both.