Advanced Network Programming II

Juliusz Chroboczek

6 March 2025

In the previous lecture, we recalled the structure of the Internet protocol stack, and how the Network Layer (layer 3) plays the role of a convergence layer: IP-based protocols are supposed to work in an Internet where IP packets are transferred (with non-zero probability) in an unrestricted manner. We saw that this model is obsolete: IP packets are no longer able to transit across the public internet, due to firewalls, NAT devices, and other middleboxes.

We therefore need a new convergence layer protocol, one that is able to cross the Internet with no restrictions. In the first part of this course, we will use HTTP(S) as a convergence layer protocol. The use of HTTP(S) as a convergence layer protocol has a number of unpleasant consequences:

- HTTP(S) is limited to client-server interaction, peer-to-peer communication is not possible, especially when TLS is being used;
- HTTP(S) is not adapted to real-time communication.

In the second part of this course, we will use UDP as a convergence layer protocol. While this solves the limitations above, it causes a fair amount of extra complexity.

1 The HTTP protocol

HTTP was originally developed as a protocol for implementing a distributed hypertext known as the *World Wide Web* (WWW). Hypertext systems existed before the WWW was developed, but they were centralised (they had a central index of link targets), which ensured that all links pointed somewhere. The WWW avoids the need for a centralised index by allowing broken links: the *404 Not Found* result is the main innovation of the WWW.

1.1 HTTP/0.9

The original version of HTTP, implemented in 1989, did not have a version number; it has been retroactively renamed to HTTP/0.9. It is a very simple request-response protocol. The client connects to the server (over TCP), then sends a single line consisting of two lexemes:

- the word `GET` ;
- the path of the requested file.

The server sends the contents of the requested file which is implicitly interpreted as HTML (there are no headers), then signals the end of data by closing the transport layer connection.

1.2 HTTP/1.0

HTTP/0.9 is a very limited protocol. HTTP/1.0, documented in 1996 in RFC 1945 (but deployed earlier) adds a number of important features:

- an explicit version number, which allows evolving the protocol;
- metadata in the form of headers, both on the request and on the response, which most significantly allows negotiating the type of the content (HTTP/1.0 is no longer limited to just HTML);
- the ability to send data from the client to the server (`POST` and `PUT` requests);
- a status line in the server's response, which, among others, makes it possible to indicate errors explicitly.

An HTTP/1.0 request consists of a request line followed by zero, one or more lines of headers. The request is terminated by an empty line.

The request line consists of three lexemes:

- the *method*, which can be, among others, `GET`, `PUT`, `POST` or `PATCH`;
- the URL (which is no longer just a path, it may contain parameters);
- the protocol version `HTTP/1.0`.

The server's answer consists of a status line consisting of three lexemes:

- the server's protocol version, `HTTP/1.0`;
- the result code, for example 200 (OK), 301 (Redirect) or 404 (Not found);
- a human-readable error message.

The status line is followed by an arbitrary number of headers. Among them, the `Content-Type` header is used to indicate to the client how to interpret the data in the body. The `Content-Length` header indicates the length of the body, which avoids the ambiguity (and layering violation) due to signalling the end of the body with a transport-layer connection close.

1.3 HTTP/1.1

HTTP/1.1, standardised in 1997 (RFC 2616), is a much more complex protocol that solves many of the issues with HTTP/1.0. The main improvements of HTTP/1.1 are the ability to reuse a single TCP connection for multiple transactions, support for reliable streaming (the *chunked* encoding), and an extensive set of metadata for controlling caching (`ETag`, `If-Modified-Since`, `Cache-Control`) which we will describe in detail in a leter lecture.

1.4 HTTP/2 and /3

HTTP/2 and /3 are (almost) semantically identical to HTTP/1.1: every HTTP/1.1 request can be mapped to these later versions, and most of the HTTP/2 and /3 transactions can be implemented in HTTP/1.1.

HTTP/2 is a protocol layered above TLS and TCP, just like earlier versions. The main improvement of HTTP/2 is the ability to multiplex multiple simultaneous transactions over a single connection, which in principle avoids the need to perform multiple TCP and TLS handshakes without causing head-of-line blocking; in practice, however, since TCP is not designed for multiplexing, performance is mediocre in many cases. HTTP/2 is widely deployed.

HTTP/3 is layered over UDP, and includes its own cryptographic protocol based on TLS. HTTP/3 avoids the need to perform multiple handshakes (a single handshake is used for both the transport establishment and the cryptographic key exchange), and performs efficient multiplexing. However, its deployment is limited by its complexity. In March 2025, HTTP/3 was supported by all major browsers (95% of the installed base) but only a small fraction of servers outside of Google.

2 Web applications

In recent years, it has become increasingly difficult to deploy native applications. In order to distribute a native application, the developers need to negotiate with a number of "app stores" (Google's, Apple's, Microsoft's) which impose arbitrary but onerous restrictions (for example, Apple's web store forbids applications that improve on Apple's own applications) and take a heavy tax on the authors' income. The situation is no better in the Free world, where distribution is funnelled through a number of package distributors with arbitrary and often inflexible policies.

HTTP was initially designed for the distribution of hypertext documents encoded in HTML. Over the years, the Web platform has been improved with a number of features that make it suitable for implementing applications, which nicely solves the problem of distribution: an application is simply contained in a web page. An application that runs in a web browser is called a *web application*.

2.1 Server-side generation

Originally, web servers served static documents. Since HTTP/1.0, the protocol makes it possible to send arbitrary data from the client to the server, which makes it possible to generate web pages in response to the user's input. A web page that is dynamically generated by the server is called a *server-side generated* page.

Server-side generation makes it possible to implement a number web applications, such as discussion boards and even webmail systems. Such applications tend to have limited interactivity: every user interaction needs to go through the server and wait for a new page to be generated, which involves latencies on the order of hundreds of milliseconds or even seconds.

We manipulated a simple server-side generated application in the first lab.

2.2 Javascript

In 1995, Brendan Eich, at the time a Netscape employee, added a Scheme interpreter to the *Netscape Navigator* web browser, and exported the internal data structures to the scripting language: this is the origin of the modern *DOM*. Since Java was then the fashionable technology, he

was ordered to redesign the programming language with a Java-like syntax (adding curly brackets and stuff). Legend claims he did that in two weeks; the result is called *Javascript*.

Since then, an HTML page can include a Javascript script, which runs in the browser. Initially, Javascript was mainly used for animations and form validation, and did not fundamentally change the nature of web applications.

2.3 AJAX

In March 1999, Microsoft Internet Explorer 5.0 added the JavaScript function `XMLHtmlRequest`, which makes it possible for JavaScript to make an HTTP request to the server. This apparently innocent addition fundamentally changes the nature of the client-server interaction: requests are no longer necessarily made in reaction to user input, but can be made asynchrounously by JavaScript code.

`XMLHtmlRequest` made a new kind of web applications possible, known as *AJAX* applications, and marketed as *Web 2.0*. The server sends a static user interface (UI) template to the client, together with a JavaScript script that makes further requests and populates the UI. As the user interacts with the application, the web page is edited in place by JavaScript code, with little or no full page reloads.

The function `XMLHtmlRequest` is obsolete: it has been replaced with the function `fetch`, which is cleaner, more powerful, and, above all, doesn't suffer from inconsistent capitalisation.

3 Web API

An AJAX application communicates with the server using a more or less well-defined protocol; in the web community, a protocol layered above HTTP is called a *Web API*.

It is usually desirable to have native applications in addition to the web application. In some cases, the web application and the native application use completely different protocols; for example, a typical e-mail application uses an ad-hoc, underspecified, HTTP-based protocol in the web interface, and standard protocols (SMTP and IMAP) in the native clients. In many cases, however, a single HTTP-based protocol (or API) is used by both the web application and the native applications.

Just like native protocols, Web APIs need to be carefully designed in order to balance efficiency, flexibility, and implementability. In the next lecture, we will describe a number of design techniques for Web APIs and explore their tradeoffs.