Advanced Network Programming V

Juliusz Chroboczek

27 March 2025

1 Proxies

During your networking class, you saw a number of examples of interctonnection of networks at the lower layers:

- at the physical layer, *hubs* interconnect multiple Ethernet cables;
- at the link layer, *switches* forward frames between Ethernet segments;
- at the network layer, *routers* forward packets between network links.

A *proxy* is a process that forwards upper-layer data. A transport layer proxy copies data between transport-layer connections without interpreting the data; examples of layer 4 proxies include SOCKS proxies, the built-in proxying mechanisms of ssh, as well as the *Tor* network.

An application layer proxy is a process that copies data between application-layer sessions. For example, SMTP forwarders ("smarthosts") accept e-mails from a submitter in order to forward them to a remote SMTP server. An HTTP proxy is a process that acts simultaneously as an HTTP server and an HTTP client, and forwards any request that it receives to a destination server (that might itself be a proxy).

There are multiple kinds of HTTP proxies, which we describe in this section.

1.1 Forward proxies

A traditional *forward proxy* is a proxy that is explicitly configured in the HTTP client (the web browser). The client makes its requests to the proxy, which forwards the request to the destination, and then forwards the reply to the client.

One application of traditional forward proxies is to provide web access to clients that re not directly connected to the Internet. In the 1990s, it was usual for client networks to not be connected to the Internet (at layer 3), in order to protect Windows hosts from attacks. Web access was provided by a single host that was both accessible from the local network and the Internet, and could therefore forward client requests to the global Internet. This kind of network setup is fortunately obsolete nowadays[1].

Forward proxies have other uses, such as improving performance by implementing a shared cache (see Section 2 below), filtering client requests by destination, or logging user behaviour (known in some circles as spying).

1. Except at the University of Paris-Cité.

1.1.1 Intercepting proxies

A special case of a forward proxy is an *intercepting proxy*. An intercepting proxy is a proxy that is not explicitly configured in the client; the network infrastructure (the routers) intercept clients' HTTP traffic and redirect it to a proxy, without the client being aware.

Intercepting proxies have a number of serious problems. From a theoretical point of view, they violate the layer structure, as the router (a layer 3 entity) detects HTTP traffic (a leyer 7 property) and redirects the traffic to a proxy (a layer 7 entity). From a practical point of view, neither the client nor the server are aware of the proxy, which makes debugging particularly challenging[2].

1.2 Inverse proxy

Unlike forward proxies, which are close to the client, inverse proxies are close to the server. The server sends its requests to the destination encoded in the DNS, which happens to be a proxy, which forwards the requests to the end host.

Inverse proxies complicate the setup of web servers, but they have a number of useful applications. First of all, they allow setting up multiple server on a single address, which avoids using multiple IP addresses or multiple ports. Second, they make it possible to isolate the end server from the global Internet. Finally, they can implement caching of server responses, which is useful if the server is written in a slow language such as PHP or Python.

2 Caching

As I may have mentioned during the lecture, the speed of light is glacially slow. Since the data being accessed is usually far away from the client, in order to incraease performance, it is useful to keep a temporary copy of the data closer to the destination. The data structure that keeps local copies of remote data is called a *cache* (pronounced [kæ⊠], just like *cash*).

2.1 Digression: disk cache

Modern SSDs have a latency on the order of a few tens of microseconds; traditional spinning disks have latencies on the order of 20 milliseconds (half a rotation). These latencies are orders of magnitude larger than the latencies of main memory. Thus, it is profitable to keep a copy of data read from disk in main memory.

The *disk cache*, or *buffer cache* under Unix, is a data structure that keeps a copy of on-disk data in main memory. When a datum is read from disk by the application, it is read from disk, it is first copied to the disk cache; if the same datum is subsequently read again, it is copied directly from the cache, with no access to the disk. When a datum is written to disk, it is stored in the disk cache, and the `write` system call returns immediately, allowing the application to continue executing; the data will be written back to disk at some later time in a process called *write-back*.

While the disk cache is meant to be invisible to applications, it may cause issues in the case of a a system crash: after a `write` system call, the data temporarily resides in the disk cache; if the system crashes before it is written back to disk, the data might be lost. What is more, data

2. System administrators who set up intercepting proxies will be the first against the wall when the revolution comes.

might be written in an order different from the one mandated by the application, which causes consistency issues. Applications must compensate for this effect with the judicious use of the `fsync`, `fdatasync` and `sync` system calls.

2.2 Digression: CPU cache

The technology used for main memory, *dynamic RAM* (DRAM), has a latency on the order of 100 ns; which is enough time for the CPU to perform on the order of 1000 operations. In order to hide the latency of main memory, CPUs implement caches that keep a local copy of recently accessed data.

On multi-core processors, there is usualy one cache per core. Thus, a datum might be present in main memory, in main mamory and one CPU cache, or in main memory and multiple CPU caches.

2.3 Cache expiration and consistency

Caching is an essential tool for improving performance, but it causes a number of issue that must be solved. In addition to the persistence effects noted above (Section 2.1), there are two main issues: cache expiration and cache consistency.

2.3.1 Cache expiration

The cache is a finite resource, typically much smaller than the original data. Thus, data must be regularly removed from the cache; this is called cache expiration. The simplest cache expiration algorithm is the *least-recently used* (LRU) algorithm, which consists in expiring the data that has been least recently used. A number of better algorithms have been developed over the years, and, in fact, there is a large amount of scientific litterature on the subject.

2.3.2 Cache consistency

In the presence of caches, there are multiple copies of the same data. If various agents (e.g. processes running on different cores) access different copies of the data, they might see different values. The property about the data that a given cache system guarantees is known as the caches *consistency* guarantee, and the programmer must be aware of these properties in order to produce correct programs.

In general, cache consistency is managed explicitly by the application. For example, the C and C++ programmming languages provide a set of atomic operations with various levels of consistency (declared in the header `<stdatomic.h>`).

3 HTTP caches

There are multiple layers of caching in HTTP, which must be taken into account by web applications.

3.1 The application cache

A web application typically maintains local copies of remote data. For example, in the chat example that we used in the lab, there is a local copy of the sequence of chat messages; while not usually formalised as such, these data constitute a cache, and therefore all issues related to caching apply to them.

3.2 The browser cache

Every web browser maintains one or more caches of remote data. The use of the cache avoids loading the same data multiple times; think for example of a logo that is displayed on every page of a website.

3.3 Caches in forward proxies

Many forward proxies implement a cache. Imagine a typical university lab connected to the web through a proxy; the day after a football match, a large fraction of the students are likely to consult the same website; the use of a caching forward proxy avoids downloading the website multiple times.

Caching forward proxies were very popular at the beginning of teh millenium, but they are seldom used nowadays.

3.4 Caches in reverse proxies

At first sight, caching in reverse proxies does not seem to be useful: since the proxy is close to the application, caching does not bring the data significantly closer to the consumer. However, recall that most web applications are written in slow languages, such as PHP or Python, by relative amateurs, while web proxies are written in fast languages by network specialists. Caching data in the proxy allows avoiding hitting the (slow) application on every request, and therefore increases performance and reduces energy consumption[3].

3.5 CDN

A *Content Delivery Network* (CDN) is a distributed set of caches designed to serve a large client population. For example, *Netflix* maintains caches within the data centers of most Inernet service providers, which avoids streaming the same video multiple times across the Atlantic ocean. Similarly, when Apple distribute a system update to a billion iPhone users, they do so through a commercial CDN.

The widespread use of CDNs raises a number of privacy and availability issues. A large CDN, such as *Cloudflare*, serves a large number of nominally independent websites, and is therefore in a privileged situation to correlate the behaviour of users across different websites. What is more, most CDNs are subject to US law; as the American administration grows increasingly unreliable,

3. In light of the energy consumed by servers implemented in Python, the teaching of Python at an undergraduate level should be considered a criminal offense.

there exists a realistic threat that major CDNs will be compelled to block parts of the Internet for a given client population.

4 HTTP cache coherence

HTTP caches do not guarantee any coherence properties: an HTTP request might return data that is *stale* (obsolete data copied from a cache). HTTP does, however, provide tools to the application developer that make it possible to implement a range of consistency properties.

4.1 Validators

A *validator* is a piece of metadata that is associated with a datum and that has the property that the validator changes whenever the datum changes. Validators are used to verify that a datum is *fresh* (not stale).

Originally, HTTP did not include a strong validator, and applications had to use the *Last-Modified* header. This caused two types of issues: last-modified dates have a granularity of one second, which makes it impossible to detect multiple changes that happen within the same second. What is more, the last-modified date is not suitable as a validator: as an HTTP reply can depend on various properties of the request, having the same last-modified date does not imply that the data are identical.

HTTP/1.1 introduced a server-generated validator, the *Entity Tag*, carried by the `ETag` header. How the ETag is generated is up to the server: the client is not supposed to analyse ETags, it just compares them for equality.

There are two kinds of ETags. *Strong* ETags are strong validators: if the ETag of a resource has not changed, then the data is bytewise identical. *Weak* ETags only indicate that the data are equivalent, in some application-specific sense, and are therefore essentially useless in practice.

It is important to note that the ETag only applies to a single resource: if two distinct resources have the same ETag, nothing can be conluded about their values.

4.2 Cache control directives

HTTP provides a number of headers that allow both the client and the server to control the behaviour of caches. HTTP/1.0 used the `Pragma` and `Expires` headers, which are today obsolete: they are ignored by servers and clients if the newer directives are present.

Since HTTP/1.1, caches are controlled with the `Cache-Control` header. The value of this header is a comma-separated list of directives. There are many directives, but only some are useful in practice.

Request directives A client can request end-to-end revalidation by specifying the `no-cache` directive. Unlike what it name implies, this directive does not prevent caching, it merely requests that all caches perform a revalidation with the origin serving, thus yielding a reply that was guaranteed to be fresh at some time between the time the request was sent and the time the reply was received.

Response directives On a response, the directive `no-cache` indicates that the value of the resource may change at any time, and therefore that all caches must revalidate the datum on every request. Just like the corresponding request directive, it does not prevent caching.

The directive `no-store` requests that caches should not store the datum. It was originally intended to prevent the storage of sensitive data, such as passwords or credit card numbers, which is a very naive approach to security. It is not useful in practice.

The directive `max-age=` specifies the maximum time, in seconds, that the dataum can be served to clients without revalidation with the orgin. The value `max-age=0` is essentially equivalent to `no-cache`.

It is usually useful to serve dynamic resources with `no-cache`, and static resource with a suitable `max-age=` directive. In the absence of a `Cache-Control` header, caches retain stale data for a implementation-dependent time, which makes applications more difficult to debug (this is called "heuristic expiration").

4.3 Revalidation and conditional requests

The process of verifying whether a datum held in cache is still fresh is called *revalidation*.

The simplest way to revalidate a datum is to send a `HEAD` request to the server, and compare the validator returned in the reply (either the ETag, or, if not present, the last-modified time). Unfortunately, this approach causes an extra round-trip in the case where the validation fails[4].

HTTP/1.1 introduced a more efficient mechanism, called *conditional requests*. The `If-None-Match` header carries one or more ETags; if the datum held on the server has an ETag equal to at least one of the ETags in the request header, the server replies with a result of 304 ("Not modified"), and an empty body. Otherwise, the server replies with a full reply, as though the header was not present.

The `If-Modified-Since` header is similar, but applies to last-modified times.

4. And the speed of light is glacially... I guess you got it by now.