Advanced Network Programming VI
Asynchronous notifications

Juliusz Chroboczek

3 April 2025

1 Problem statement

HTTP is strictly a request-response protocol: every interaction is initiated by the client, and consists of a single request-response pair. This strict discipline prevents some modes of interaction.

One important limitation of the request-response discipline is that it is impossible for the server to initiate an interaction. As we saw during the previous labs, this makes it difficult for clients to react in real-time to changes in the server's state. Similar situations arise in many applications: consider for example an online chess game (the server needs to notify the client when the opponent plays a move), or an email client (the server needs to notify the client when a new email arrives).

2 Polling

The solution that we employed during the previous labs is called *polling*. The client periodically sends a request to the server in order to inquire about any state changes. While this solves the problem, it results in either slow reaction times (if the polling interval is too large) or high network traffic and associated battery usage (if it is too low). Designing an algorithm to dynamically adapt the polling interval to user and server activity is at best difficult, and often impossible.

3 Asynchronous notifications

The solution to the problems outlined above is to allow the server to initiate a message. A message sent by the server to the client without a prior request is called an *asynchronous notification*. There are many technologies that may be used to implement asynchronous notifications.

3.1 Raw TCP

It would be tempting to abandon HTTP and use a raw TCP connection for notifications. This has a number of problems.

First of all, TCP endpoints are socket addresses (pairs of an IP address and a port number), not URLs; therefore, TCP integrates poorly within the web ecosystem. Second, TCP is a stream protocol, not a message protocol; thus, in order to send messages, some form of framing is necessary.

More seriously, TCP is not available in the browser, and, as we saw during the lecture, there are good reasons why it cannot be.

3.2 Long polling

The data model of HTTP is not quite clear: it is not well defined whether the body of an HTTP reply is a message, or whether it is a stream of bytes. Long polling is a technique that treats a reply's body as a stream in order to provide server notifications to a client.

In long polling, the client sends a request to the server. The server keeps the connection open, and sends notifications to the client whenever necessary. The server may close the connection at any time, in which case the client will make a new request.

Long polling has a number of issues. First of all, not all proxies will flush partial replies in a timely manner; if a proxy decides to delay a reply (e.g. in order to wait for a complete body), then the client will not receive the notification.

More seriously, long polling is unidirectional. This has two consequences. First of all, for any non-trivial application, the client will also need to send requests to the server; if long polling is used, then these request will by necessity be sent over HTTP. Since the requests and replies use a different TCP connection than events, there is no guaranteed ordering between the two, and correlating the two must be done by the application.

More seriously, long polling does not provide any way for the client to provide feedback to the server: the server has no way to ascertain whether the client is still alive, and whether it is reading the data in a timely manner. If the server sends data faster than the network or the client is able to accept it, the data will be stored in TCP's buffers, and increase latency. (This phenomenon is part of the *bufferbloat* problem.)

3.2.1 Server-sent events

Server-sent events (SSE) is a standardised framing format for messages sent over a long-polled connection. SSE is much more convenient than manually coded long polling, since it is natively implemented in modern browsers and there are server-side libraries that implement it in most programming languges. However, since it is just a form o long polling, it has all of the problems described above.

3.3 WebSockets

WebSockets are a bidirectional message protocol layered above TCP (and usually TLS). Unlike raw TCP, a WebSocket connection is between a web client and a URL; unlike HTTP, WebSockets implement bidirectional streams of messages between client and server.

A WebSocket connection is established by sending an HTTP request to the WebSocket endpoint with a header indicating the client's wish to switch to the WebSocket protocol. If the server accepts the connection, it sends an HTTP reply idicating that it is switching protocols, and keeps the connection open, which is then used to exchange messages encoded as TLVs between client and server.

Since WebSocket is layered above TCP, it inherits its properties: it is a reliable and ordered protocol. This implies, in particular, that it suffers from head-of-line blocking (a large message will delay all subsequent messages) and of unbounded delays when a packet is lost in the network.

3.4 SCTP

SCTP is a transport layer protocol that was originally designed for telephony signalling (call setup, routing, and billing in the telephone network). SCTP is a very complete protocol, and has a number of useful features:

- SCTP implements multiple streams of messages (limited to 16 kB each);
- SCTP implements ordering (messages within a stream are ordered), partial ordering (messages in different streams are not ordered with respect to each other), and unordered delivery, at the programmer's choice;
- SCTP implements unreliable delivery, if requested by the programmer.

As it is a transport layer protocol, SCTP does not cross most middleboxes (such as NAT routers), and is therefore useless in the modern Internet. However, SCTP has also been implemented over UDP, and this variant is able to cross most middleboxes.

Modern browsers implement SCTP over UDP as part of the WebRTC protocol suite (as "WebRTC data channels"). WebRTC is a complex technology, and out of scope for this course.

3.5 Proprietary server push

A number of platforms use proprietary techniques to implement notifications. For example, the Android platform supplies a protocol called *Firebase Cloud Messaging* (FCM), which requires applications to use Google's cloud offering (Firebase); Android applications that use FCM enjoy a number of advantages, such as better background scheduling. I have no idea how this kind of bundling can possibly be legal in the European Union.

Apple provides something similar for iOS (*Apple Push Notification Service*), but I am not familiar with the details.

4 Bidirectional protocols

In plain HTTP, there are just two kinds of messages: client requests and server responses. Once we switch to a bidirectional protocol, there are potentially no less than six kinds o messages:

- client requests and server responses;
- server requests and client responses;
- server notifications (that require no client reply) and client notifications (client messages that require no server reply).

Of course, making such a protocol reliable (deadlock-free and unambiguous) is significanly more difficult than when a unidirectional request-response discipline is being obeyes, as in HTTP.

4.1 Write-write deadlock

It is obvious that if the client and server are both waiting for a message, the protocol will deadlock. What is less obvious is that if the client and server both attempt to send a message, the protocol might deadlock if TCP's buffers are full. This second kind of deadlock is difficult to reproduce (it only happens with large messages and small buffers), and therefore does not usually appear in testing.

There are two techniques that can be used to avoid such write-write deadlocks. The older only consists in only ever using non-blocking operations, and performing a read operation whenever a write would block. Thus, older network applications contain the following somewhat puzzling pattern:

```
while(1) {
    rc = write(...);
    if(rc < 0 || errno == EAGAIN) {
        rc = read(...);
        if(rc >= 0)
            handle_message(...);
        }
    } else {
        break;
    }
}
```

The modern technique consists in using multiple threads, and making sure that a single thread only ever performs reads or only ever performs writes. Since sent and received messages are not in general independent, this requires some non-trivial communication between threads.

4.2 Ordering ambiguity

If notifications are allowed in both the server-to-client and client-to-server directions, an interesting ambiguity arises even if the communciation channel preserves ordering. Consider a client that sends a message that changes the server's state, and then receives a notification from the server; was the server notification sent before the server received the client message, and therefore reflects the new server state, or was it sent before, and therefore reflects the old state?

No such ambiguity occurs if the client message is a request: the ordering between the server answer and the server notification indicates the ordering of the notification with respect to the request. Thus, it is necessary to either ensure that timing ambiguities can be resolved (for example by including a sequence number in all packets), or only allow notifications in one direction (typically from server to client).