PubNub

Overview of Realtime Streaming Protocols

Table of Contents

Introduction 3

Communication Paradigms 5

Request-Response Protocols 8

HTTP 1.1 9

Persistent Connections 10

Pipelining 11

Long Polling 12

HTTP Streaming 13

WebSocket 14

Piggybacking on HTTP 1.1 16

BOSH – or XMPP over HTTP 16

HTTP for constrained environments 17

CoAP 17

HTTP/2 20

Publish/Subscribe Protocols 21

MQTT 22

XMPP PubSub 23

WebSub 24

Conclusions 24

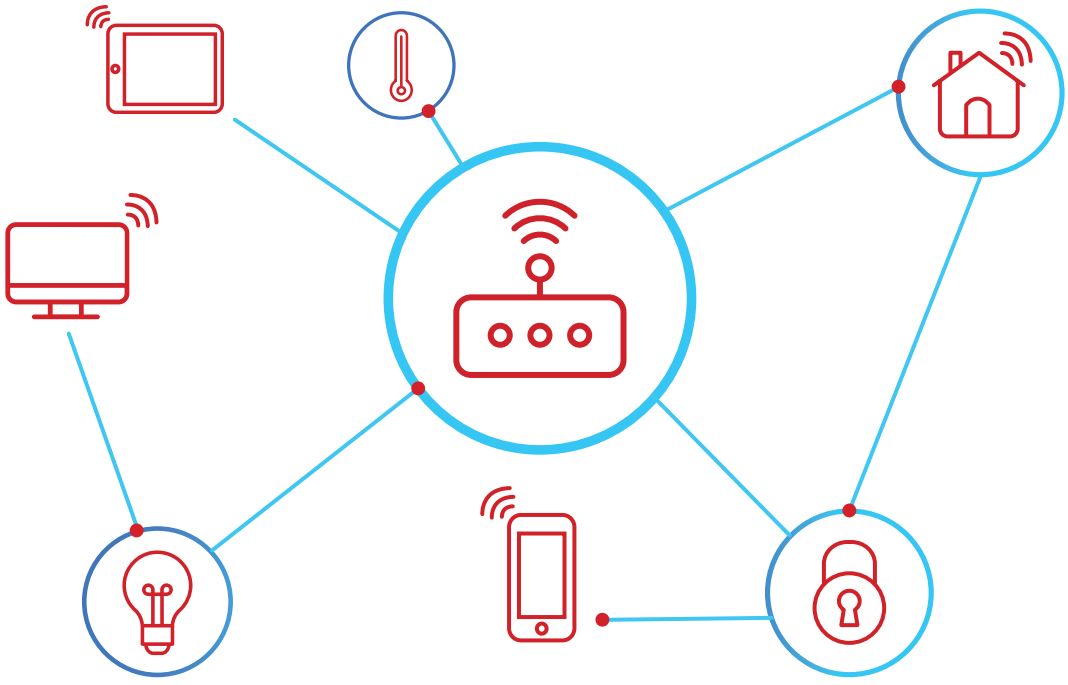
Overview of Realtime Streaming Protocols

It is no exaggeration to say that the World Wide Web has now become the universal computing platform, used for every form of data-driven interaction between humans, humans and machines and between machines. These diverse interactions are based on a few key communication paradigms and protocols, honed and adapted over the past forty years to their current state as the common substrate which powers the entire Web.

This White Paper provides an overview of these paradigms and key protocols. It collects many topics we have touched on in various PubNub blogs into a comprehensive story.

This overview describes the key communications protocols developed for the Web, by open standards setting bodies like the IETF and W3C, and which are now built into operating systems or available as royalty-free, open source implementations.

To keep this White Paper to a reasonable length, we will concentrate on a large subset of these protocols, leaving out those used for voice and video communications and media delivery as these require more elaboration on topics that would divert from our main thesis. We will concentrate on those that are typically browser-based (for human-to-human and human-to-machine interactions) or data transfer for machine-to-machine (M2M) communications, such as required for the burgeoning area of Internet of Things (IoT).



Communication Paradigms

All interactions require communication between entities, and the two key ways to do so broadly mimic the way humans gather and consume information.

If we know who we wish to talk to, we open a communication channel to them – using carrier pigeons, letters, phone calls, emails, instant messages, and so forth as each improvement of technology allows. We ask questions and we get answers, or we share information both ways once a steady communication channel is open. This is more formally known as the request-response paradigm.

When we don't know where the information we seek is, or we chose not to bother with the task of "hunting and gathering" information, we leave this task to others. Newspapers, magazines, bulletin boards, news portals – again, the same pattern, but adapted to newer technologies allow us to get information at the rate at which it becomes available or changes, without the effort of actively soliciting or searching for it. This is the publish-subscribe paradigm.

Understanding these two ways of information gathering/sharing is key to following the different protocols in the rest of this White Paper. We'll describe these from now on using communication networks (wired or wireless) between the communicating entities, called client/server or publisher/subscriber as appropriate.

REQUEST-RESPONSE PARADIGM

As the name suggests, one party (formally, the client) sends a query (request) to another (the server) which responds with an answer (response). The two communicating entities can keep up such request-response exchanges between themselves until either party drops out.

This simple pattern requires a lot of heavy lifting by infrastructure, to avoid burdening the client and server with delivery issues. By now, the

underlying communication network on the Web is almost always TCP/IP, the data communication protocols tuned over the last fifty years by the IETF to provide an error-free, bi-directional communication path. A request-response protocol operating on top of a TCP/IP connection can be guaranteed that requests or response will be delivered in order, at most once and without corruption. If the data needs to be protected against eavesdropping or alteration while in transit, the TLS protocol, operating on top of TCP, provides the necessary protection. In fact, the protocol stack in **Figure 1(a)** is by now the de facto communication suite for almost all Web interactions.

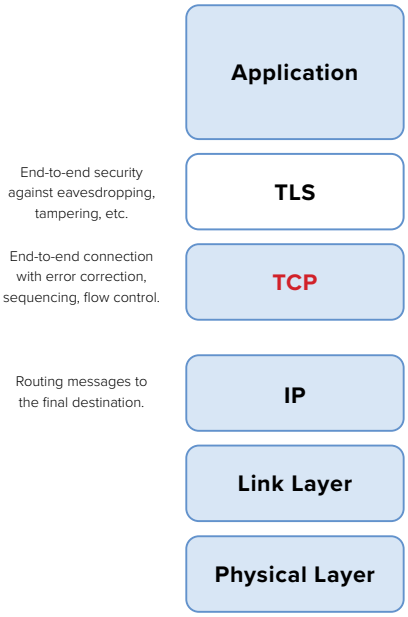


Figure 1(a)

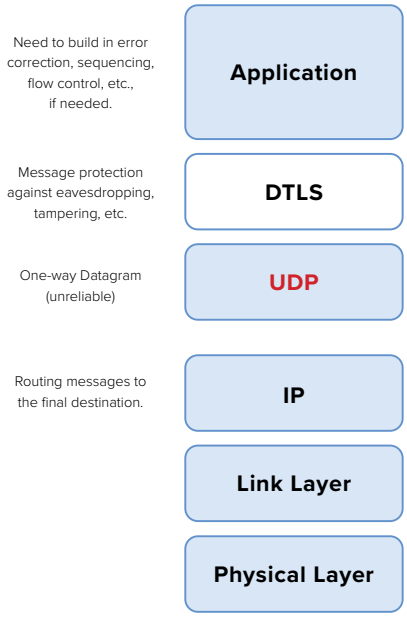


Figure 1(b)

There's another protocol, UDP, operating at the same level as TCP – see **Figure 1(b)** – which provides a light-weight, one-way data transfer mechanism. If this is used, and it is for certain types of applications, an upper-layer protocol has to take care of most of the features that TCP/IP provides. The request-response protocol has to ensure that each request is numbered, so that a response can be correlated with the corresponding request, duplicated requests and/or responses are discarded, lost requests are resent, and so on. Thus, this type of data transfer is best suited for those cases where the application is resilient to loss and duplication – a simple query, say, where the request can be

repeated multiple times without causing any harm. (Such queries which can be resent repeatedly without harmful consequences are more formally called idempotent.)

PUBLISH-SUBSCRIBE

The request/response interaction paradigm represents a tightly coupled interaction between known entities on pre-defined types of information. In contrast to this, the Publish-Subscribe pattern is a loosely coupled interaction style where consumers of information (the subscriber) are decoupled from the sources of information (the publisher). Subscribers express interest in information on particular subjects, which publishers provide. Information (more formally called events) can be variously described – one common method is via topics (e.g., stock quotes), while more sophisticated forms include specific content (e.g., price of a particular stock) or even expressions (e.g., when the price of the stock drops below or rises above a certain threshold).

To operate at scale, such interactions are almost always mediated by a third party, variously called a message bus or event broker. Publishers make available the sorts of information (events) they are willing to publish to the mediator, which advertises it to subscribers, who subscribe to those events they are interested in and are notified of its availability. The available event can be pushed to subscribers who've expressed an interest in it, or they can periodically pull the events from the mediator by polling it.

This decoupling between the publishers and subscribers occur in several dimensions, which gives this interaction pattern its usefulness and scalability. These are:

Time: A subscriber or publisher need not be available all the time, and more importantly not active at the same time. This is particularly useful in situations with limited or poor connectivity. The mediator retains the published events and ensures that it is available to subscribers at a time and manner of their choosing.

Space: Publishers and subscribers do not know each other's location or identities. Except for the nature of the information that is shared, they are completely unaware of each other.

Manner: In contrast to the request/response interaction, where the client and server have to be active and engaged in the interaction at the same time (i.e., the interaction is synchronous), the publish/subscribe interaction is asynchronous. Publishers provide information when available, while subscribers pick up the information when they wish or are able (in other words, asynchronously). Neither party is held up waiting for the other to act, which, as we shall see, can be an issue in some request-response interactions.

Request-Response Protocols

This section will mainly be about HTTP, which has been the workhorse of the World Wide Web since its creation in 1991. From that time on, this protocol has been adapted (contorted, some critics might carp) to meet every conceivable variant of the basic request-response paradigm. It is now used as the basis of ordinary web interactions, of course, but also instant messaging, bulk data transfer, multimedia streaming, etc.

One reason for this need to reuse HTTP for new applications – even if these are not strictly request/response – is because HTTP messages almost always freely move through NATs and firewalls, most of which are configured by default to leave open TCP ports through which HTTP, and its secure variant, HTTPS – HTTP over TLS , communicate. By piggy-backing a new application protocol on top of HTTP, clients behind NATs can reach servers outside and vice versa.

In what follows, we'll describe the ways in which HTTP has been adapted to meet new challenges, as new applications reuse its basic interaction pattern to achieve their objectives.

HTTP 1.1

Figure 2(a) shows a typical HTTP request-response interaction pattern. As it was initially designed for simple web interaction, where web "pages" are delivered (via a simple read request – GET) or online form data submitted (via a write request – POST) or metadata retrieved (a HEAD), a series of multiple request-response pairs was sufficient to achieve this type of application's objectives.

Each request-response was bracketed by the setup and teardown of a TCP/IP connection (a three-message sequence that is not shown in the figure), a step whose inefficiency did not become apparent until HTTP and web surfing became the most prominent form on internet traffic.

Many of the techniques discussed in what follows are ways to get around the inefficiencies that HTTP's use of the simple request-response pattern imposes.

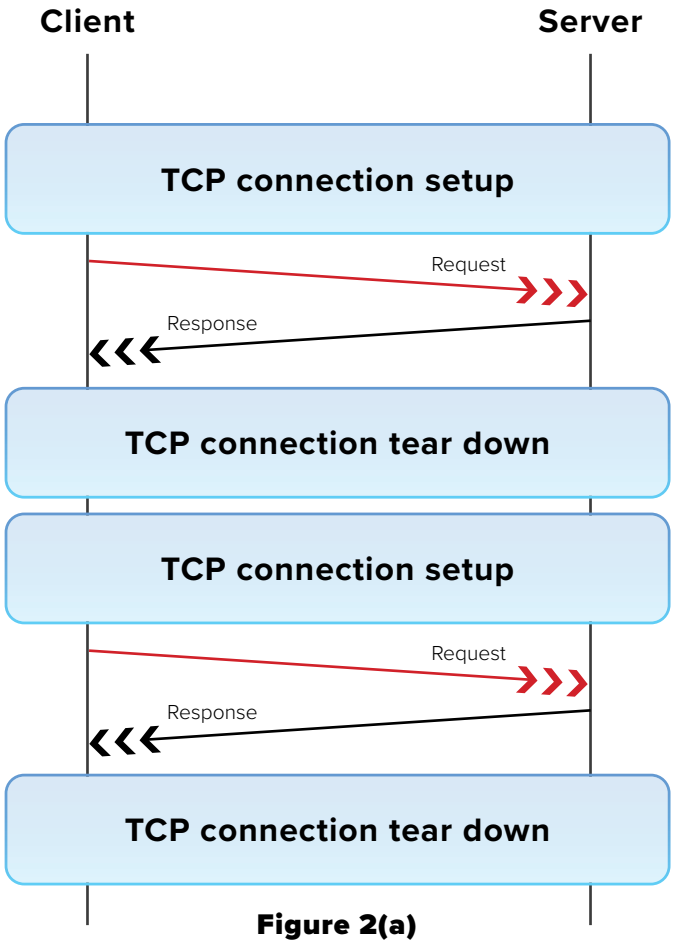


Figure 2(a)

PERSISTENT CONNECTIONS

The simplest change to improve HTTP efficiency is to ensure that the TCP connection is not dropped after each HTTP request-response.

HTTP 1.1 has this as its default behavior, with a change explicitly indicated by a Connection: close header. To allow for the same behavior in HTTP 1.0, a new header, appropriately called Connection: Keep-Alive, was introduced which signals that the TCP/IP connection should not be dropped. (See Figure 2(b)).

Of course, servers and intermediaries must respect this setting if the desired effect is to be achieved. In fact, increasingly, most do.

Another technique is to set up multiple TCP/IP connections to the server. Most browsers allow up to 6. However, such attempts at "parallelism" also suffer from the TCP connection setup overhead as well as managing multiple connections.

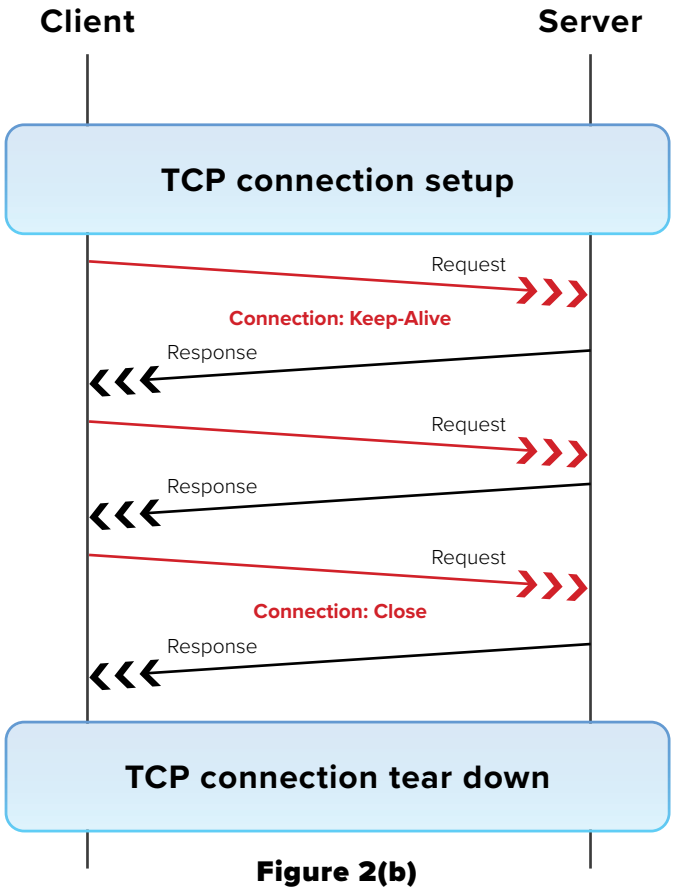


Figure 2(b)

PIPELINING

Instead of waiting for a response before making the next request, the client can send multiple requests one after another – as shown in **Figure 3** – over the same, kept-alive TCP/IP connection, a technique called pipelining.

Unfortunately, the HTTP protocol requires that the server return responses in the order received; so, a long running server operation (for request 2 in the figure) can block a much more quickly completed one (for response 3).

This head of line blocking, as it's called, reduces performance and makes web page loading slow down.

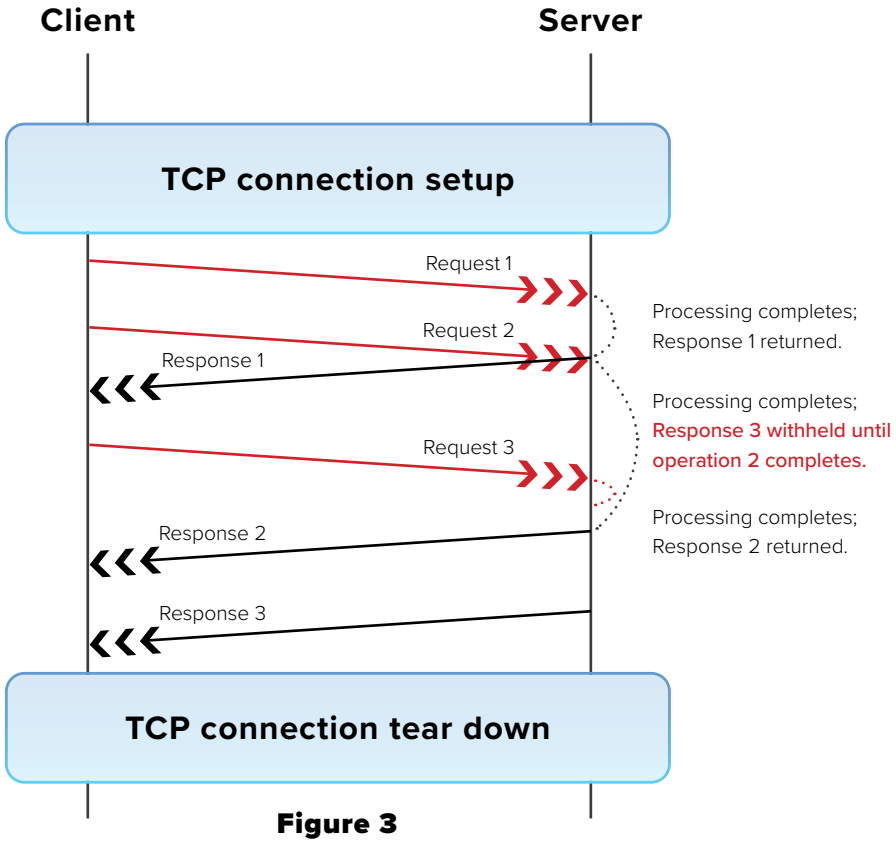


Figure 3

LONG POLLING

The simple request-response interaction is fine so long as it is the client that requests information from the server. The server cannot initiate any communication with the client except in response to a request. This means that server operations that create data at intervals can only be retrieved if the client asks for it.

The client can, of course, poll the server at regular intervals to retrieve new data since its previous request. If no data is available at that time, an empty response is returned. But this is particularly inefficient when the server data updates are irregular, as most of the responses will be empty. Small footprint devices or those communicating over bandwidth limited networks, such as many IoT devices talking to a central controller, are most affected by this.

Long Polling is a clever workaround of the above shortcoming of frequent polling with empty responses. As shown in **Figure 4**, instead of returning an empty response when no data is available at the server, the latter simply holds back – hence the word "long" in the name – the poll response until there is data to send back.

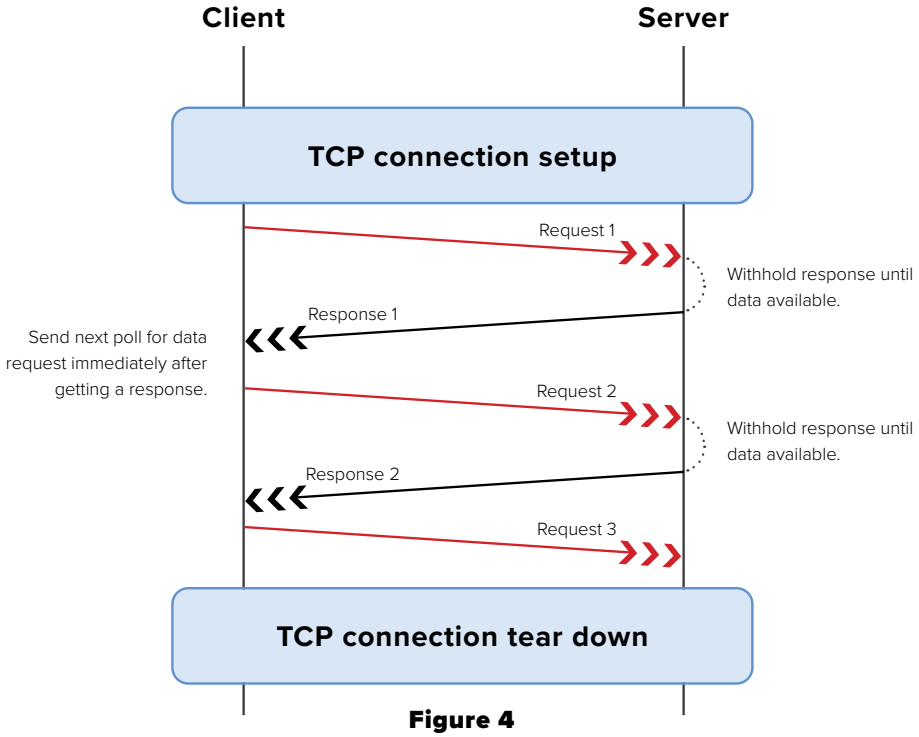


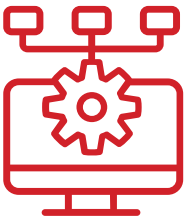
Figure 4

There is a configurable timeout to prevent the client from being held up by a "hanging GET" indefinitely. But if the timer value is chosen carefully, a response is usually returned before the timeout.

And once the response is returned, the client immediately sends out another poll (request) – and the cycle continues.

Long polling is clearly superior to periodic polling in those situations when data is available at irregular intervals.

HTTP STREAMING



The word "streaming" in this context (defined and used long before OTT video became synonymous with this term) refers to a server keeping the HTTP connection open indefinitely, even after it has pushed the first piece of data in a response message. In effect, the response body in HTTP streaming might be considered as open ended, allowing, in effect, multiple responses to be sent for a single request.

This is signaled to the client by the inclusion of a HTTP header Transfer-encoding: chunked in the response. The response isn't considered complete until the server sends an End of File (EOF) or either side explicitly closes the connection. Each data item in a "chunk" of data in a response is identifiable by a length (in hex) and a CR, a size 0 indicating the end of that chunked response.

Again, as before, intermediaries in the message path may not always respect the signals for HTTP streaming and retain a chunked response until the entire response is received.

SERVER-SENT EVENTS

Server-sent Events (SSE), defined by the W3C as a part of the HTML5 suite, is both a browser API and an on-the-wire data format for the types of notifications that can be sent by a server. It is a replacement for long polling for browser-based web applications, and provides a way for the server to push data to a client as data becomes available. Unfortunately, the data can only be text-based; therefore, any binary data needs to

be base 64 encoded, but which can be compressed – using gzip, say. Even with this limitation, it's a simple way to send notifications from the server to the client asynchronously for applications like status updates.

The mechanism is quite simple, and makes use of both Persistent Connections and HTTP Streaming described earlier. It is triggered by the invocation of the SSE API in the client application code. Over a long-lived (i.e., Keep-Alive connection as described earlier) HTTP connection, a client identifies, in a request to the notification data source on the server, an interface where it can receive notifications as an event stream – the data format referred to earlier, identified by a HTTP header Accept: text/event-stream.) The client also sets up a listener, which lets it know that data has been received on this interface, or an error.

Notification data, as soon as generated by the server, is sent to this interface as an event stream in chunks (indicated by the previously mentioned HTTP header Transfer-encoding: chunked) in the response message body. As these events are received at the interface, the client application code is notified by the event listener of new data, which can then be retrieved.

WEBSOCKET

We've chosen to include WebSocket as a subsection of HTTP enhancements because while it can be used independent of HTTP, it almost always never is.

In its HTTP manifestation, WebSocket is an API (defined by the W3C) through which a browser can create a bi-directional data channel with a server over which arbitrary application data can be exchanged. Hiding behind this simple API is an on-the-wire handshake protocol (defined by the IETF) to set up this bi-directional channel and a data framing format.

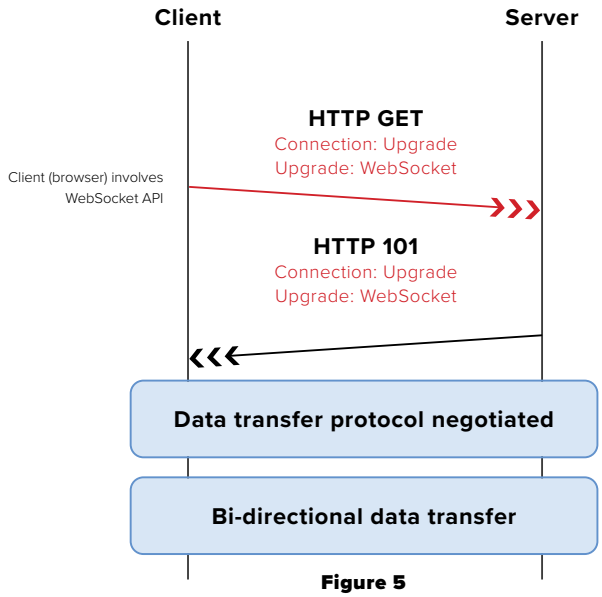


Figure 5

Simplifying a lot of details, **Figure 5** shows how the data channel is set up by the client using a HTTP Request with two new headers – Connection: Upgrade and Upgrade: websocket.

The server accepts the set-up request with a HTTP Response code 101 Switching Protocols, and reflecting back the two headers.

Other headers for authentication and the protocol to be used over this data channel may also be exchanged during this setup phase.

After a successful setup, arbitrary application data can flow from either side over this channel.

WebSocket is the latest in a series of HTTP 1.1 improvements to break the strict request-response pattern and allow both client-to-server as well as server-to-client interactions over a single TCP connection. Using HTTP to set up the channel retains all the goodness of easy NAT and intermediary traversal that HTTP support has now baked into our networking infrastructure. It also removes the limitation in Server-Sent Events of sending text-only data, and all the latency and processing inefficiencies of Long Polling and HTTP Streaming.

However, using the raw byte stream that WebSocket sets up requires that the application using this channel now do all the hard lifting of correlating requests with responses, caching, etc.

PIGGYBACKING ON HTTP 1.1

The ubiquitous use of HTTP for almost all types of interactions has created a networking infrastructure of intermediaries (caches, proxies, NATs, firewalls, etc.) that are adapted to the protocol. Perhaps most importantly, ports 80 (and 443) are now almost always left open for HTTP (respectively HTTPS) access to the public internet from a private network. This has led to all sorts of applications which take advantage of piggybacking their protocol on HTTP. What follows is one example of using HTTP to tunnel through a firewall.

BOSH – OR XMPP OVER HTTP

First, a few words about XMPP – Extensible Messaging and Presence Protocol. It's an open standard created by the IETF, and its client-server model allows a wide range of applications that take advantage of the protocol to send one-way or bidirectional messages between entities. Applications of XMPP include instant messaging, user presence status updates, group chat, data syndication and many more.

We won't describe the protocol here but only illustrate the creative way to use XMPP services when direct access to an XMPP server is blocked by a firewall. This open solution created and maintained by XMPP.org – called Bidirectional Streams over Synchronous HTTP, or BOSH – uses HTTP as an underlying substrate to get past the firewall.

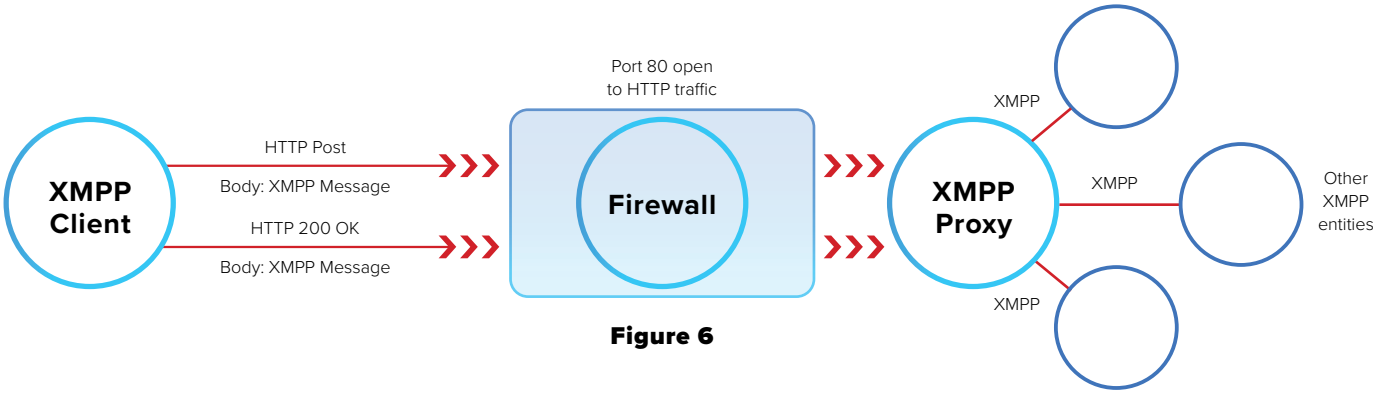


Figure 6

As Figure 6 shows, a XMPP client behind a firewall talks to a XMPP proxy (formally the Connection Manager) on the internet, which acts as a proxy for the client. The XMPP message is sent in the body of a HTTP Request message, which is extracted and forwarded by the Connection Manager to the appropriate XMPP server. And likewise, for the XMPP server response, which the Connection Manager returns in the body of the HTTP Response.

This interaction pattern can be (and indeed is) used by other application layer protocols to get past firewalls.

HTTP FOR CONSTRAINED ENVIRONMENTS

There are environments, typically in the burgeoning area of machine-to-machine (M2M) communications and the Internet of Things (IoT), where devices are constrained in memory and processing power while communicating over low bandwidth, lossy networks. For example, sensors and actuators in hard-to-reach locations or too numerous to individually target, as is often the case in various automation situations.

Querying such a device (now in a server role) for status or to update the firmware using HTTP may not be efficient or even feasible in such constrained situations. However, the request-response pattern and HTTP's familiar semantics (GET to read and POST to write, among other methods) can easily be retained, while improving performance in other dimensions. One such open standard is CoAP – the Constrained Application Protocol.

COAP

The Constrained Application Protocol (CoAP), standardized by the IETF, provides a light-weight request/response interaction pattern which mimics HTTP semantics using a binary protocol for reduced message size and UDP datagrams for message transport. It was designed for machine-to-machine communications in environments with lossy networks and low-footprint end points.

As we noted in an earlier section, the use of UDP means that an upper-layer protocol is needed to provide state management, message sequencing, delivery reliability and other features (e.g., flow control), when needed by the M2M/IoT application.

CoAP provides all this through two logical layers, as shown in **Figure 7**. On the wire, both layers are contained within the binary header.

The request/response layer uses method codes (numbers rather than text) to identify the type of action requested – GET, POST, PUT, DELETE, which closely resemble HTTP methods, although with some nuances/differences which we'll elide over for now for sake of length – and response codes which also resemble and are a subset of HTTP response codes (e.g., 2.00 and 4.04 for the familiar HTTP 200 OK and 404 Not Found) . These methods and responses carry their own ID, Token, for correlating requests and responses, several of which may be sent in either direction and outstanding at any time.

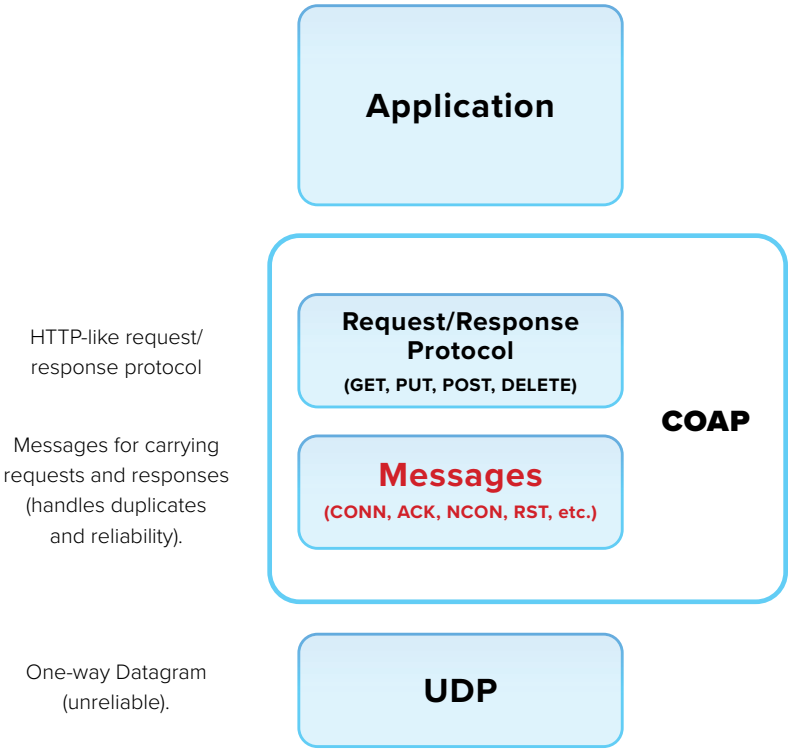


Figure 7

The underlying message handling layer carries a request or a response code in one of four types of messages – CON, for a message whose receipt must be confirmed with an ACK, a NCON which need not be confirmed and a RST message (for reset) to indicate receipt of a message that cannot be processed.

It's easiest to show how CoAP's two layers operate in conjunction through some examples. In **Figure 8(a)**, a request to GET a sensor's reading is sent in a CON message and the response piggy-backed in the ACK. **Figure 8(b)** shows a case where the reading is not immediately available, but promptly ACKnowledged to prevent needless resubmissions. When the reading is available, the server sends it in a new CON message in the opposite direction. Note how the separation of the request/response token ID from the message ID ensures that requests and responses can be correlated despite being sent over different underlying messages. If anything, this example shows how HTTP-like semantics can be achieved and data asynchronously pushed over a one-way datagram network without polling (long or otherwise) or any of the other complexities identified in earlier sections.

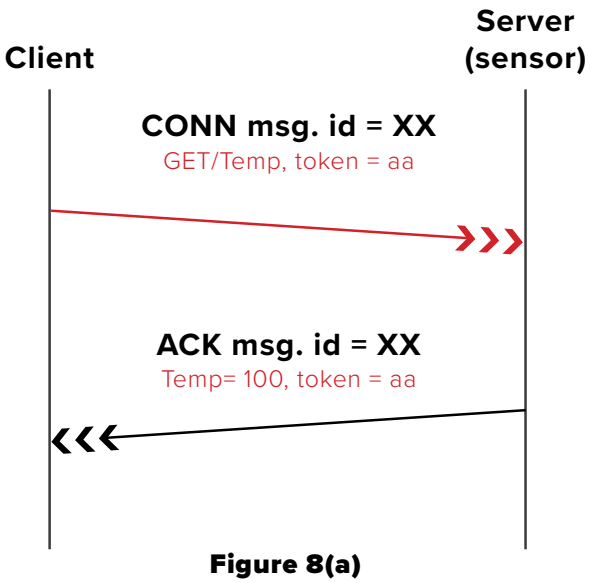


Figure 8(a)

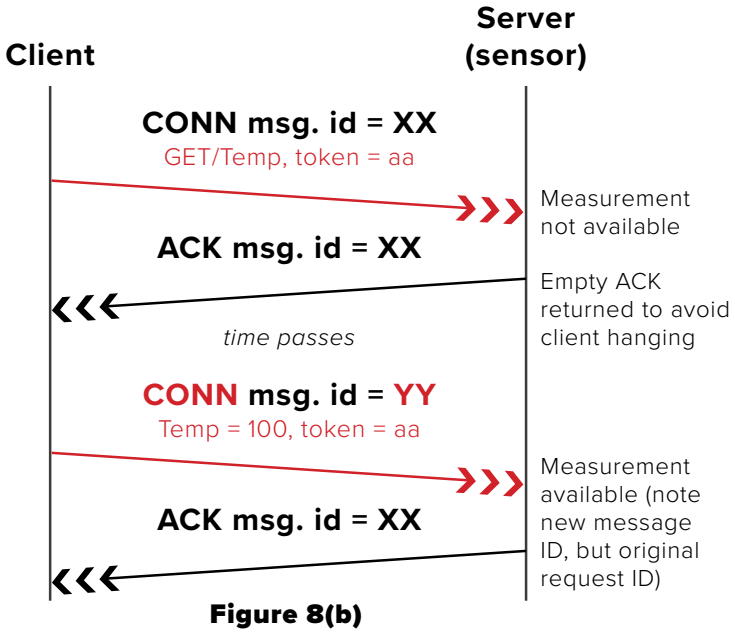


Figure 8(b)

CoAP contains many more options, and the reader should consult the suite of drafts and specifications accompanying the base protocol to understand its versatility.

HTTP/2

The previous sections outlined various solutions that application designers have created over the past years to adapt the HTTP request/response pattern to allow more flexible interactions, culminating in WebSocket's bi-directional communication channel. Instead of bespoke solutions for use in different situations, each bringing new performance or implementation issues, the IETF decided in 2012 to standardize a major overhaul of HTTP using as a basis a protocol, SPDY, introduced by Google and implemented in its Chrome browser.

With the completion of this new version of HTTP in 2014, called HTTP/2, the IETF dealt in one swoop with all the issues that led to the various ad hoc solutions created for HTTP 1.1 described earlier. The result is a completely new on-the-wire protocol but with the same application semantics (the various HTTP methods and URI schemes) as before so that applications remain unchanged. It would take too long to describe the various features of HTTP/2, which we'll cover in future articles on the PubNub blog, but we contrast here the key ways in which it addresses the shortcomings of HTTP 1.1.

HTTP/2 is a binary wire protocol; thus, it's backwards-incompatible on-the-wire with text-based HTTP 1.1. This allows for reduction of the message size, an important consideration in many environments, but also requires that both clients and servers upgrade to the new protocol to benefit from its features.

HTTP/2 requests and responses are multiplexed on logically independent connections called streams, so that a blocked or stalled request does not block the queue for those behind it. Streams can be prioritized and flow controlled so that the critical streams can be acted on first.

HTTP 1.1's header bloat, where header data is repeated for requests to the same resource and often take up more data than the response body, is mitigated by only resending changed or new headers.

And, a server can preemptively push data if it anticipates, based on a current request, that the client is likely to request it sooner or later.

There's much more to it, of course, but it would take a separate article to describe it all. HTTP/2 penetration, as measured by Web Technology Surveys, stands at about 30% at the time of this writing and is offered by almost all major web sites and browsers.

There's also work in IETF on a mapping of HTTP/2 onto a secure, UDP transport named QUIC, the result called HTTP/3, but describing it here would take us further afield.

Publish/Subscribe Protocols

Before providing an overview of publish/subscribe protocols, let's recap our earlier discussion of this interaction pattern. In contrast to the request/response pattern, where pre-defined types of information are consumed from a known source, the publish/subscribe pattern allows for a fully decoupled mode of communication between unknown entities, which may or may not be available at all times. Thus, the need for an interaction pattern that is based on events (information of interest in a given domain) produced by publishers, collected by an intermediary, and disseminated to subscribers that have expressed interest in those events.

Depending on the requirements, event delivery can be made reliable (e.g., using TCP instead of UDP), secure (using TLS) and pushed one-to-many (e.g., using IPv6 multicast). However, independent of these underlying techniques, the basic mechanism remains unaltered.

This pattern is particularly suited for cases where small-sized event data (e.g., a temperature reading) needs to be distributed in large volumes (e.g., from thousands of sensors) over networks with low bandwidth and dubious connectivity. This is the environment in which many IoT devices operate and M2M communication occur. That's why IoT/M2M protocols

have gravitated towards publish/subscribe protocols, some examples of which we outline in what follows.

MQTT

Message Queueing Telemetry Transport – or just MQTT, the current preference being to not spell out the acronym – offers a lightweight messaging mechanism to convey event data from constrained devices across unreliable, bandwidth constrained and high latency networks to remote subscribers. It is being standardized as an open, royalty free protocol by OASIS.

MQTT messages are in a binary format and typically transported over a TCP connection (although an UDP variant has also be standardized) to ensure delivery guarantee. All interactions are mediated by a MQTT broker, to which both Publishers and Subscribers connect using a CONNECT/CONNACK pair.

Please see **Figure 9** for typical message exchanges for a stock price quote publication service.

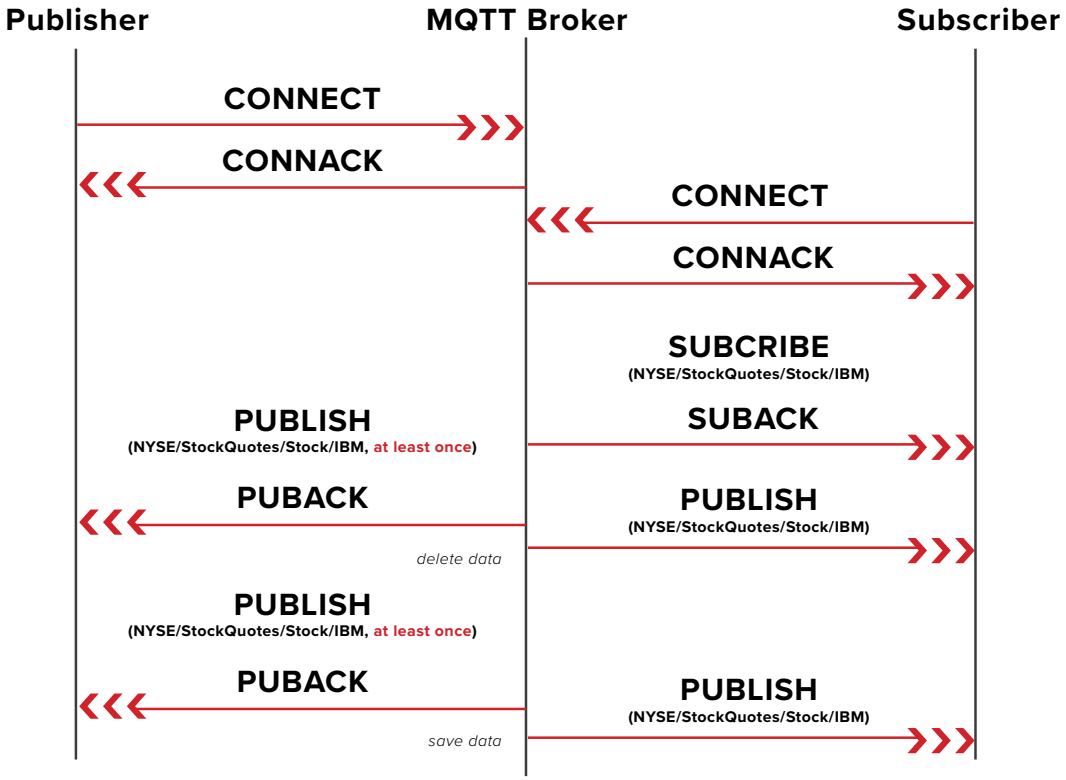


Figure 9

Publishers identify the subjects (called topics) about which they will provide data to the MQTT broker in a PUBLISH message Topics have a very simple grammar, such as NYSE/StockQuotes/Stock/IBM for the stock price, the specific value carried as payload of the message. Payloads can be in any format. Publishers can ask for levels of delivery guarantees when they publish to a broker – at most once (which means best effort), at least once or exactly once – depending on the application's requirements. A Publisher can also ask that the data be retained by the broker. When so requested, a MQTT broker only retains the last value received – that's one reason for deprecating the misleading term "Queuing" in the expansion of the acronym.

Subscribers connect to the broker and listen for publication of data on topics they are interested in using SUBSCRIBE messages, which are acknowledged by the broker using SUBACK messages. Each SUBSCRIBE message lists all the topics of interest to that particular subscriber. Conversely, there is an UNSUBSCRIBE message to drop out of those topics the subscriber is no longer interested in. When data on a topic has been PUBLISHed to the MQTT broker, it, in turn, PUBLISHes it to all entities that have subscribed to that topic. WebSocket is often used the data transfer mechanism from the broker to subscribers, especially as subscribers typically view updates via their browsers.

MQTT is by now firmly established as a major M2M communication mechanism, and its uptake can only grow given the number of IoT devices expected to be deployed in the coming years.

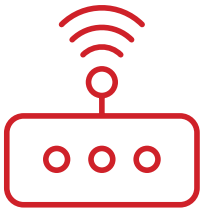
XMPP PUBSUB

The XMPP protocol has an extension XEP-0060, maintained by XMPP.org, which implements the publish/subscribe interaction pattern. It defines certain XMPP nodes (much like the MQTT brokers described earlier) to which XMPP publishers publish data, which the node distributes to interested XMPP subscribers. The language for publishing and subscribing is verbose, being based on XML. All the usual interesting use cases are possible, and the PubSub mechanism is included in all major XMPP implementations.

WEBSUB

WebSub (formerly known as PubSubHubbub) is a W3C Recommendation, and used for pushing web content like blog updates (from WordPress.com, for example, among others) to feed readers (e.g., Google Reader, Bloglines, etc.). It's another in a line of openly standardized web-scale publish-subscribe mechanisms, but one based entirely on HTTP.

Subscribers search for sources of events (topics) on interest via a HTTP GET, and subscribe (using a HTTP POST) to the "hub" from which such events are published. The subscriber includes a callback URL in the subscription request, to which events are posted when available. Publishers notify their hubs when content has been updated using any mechanism. The hub, in turn, POSTs the changes to the callback URL on the subscriber. (See here for an illustrative figure of the interactions.)



Conclusions

As the reader who has come this far will no doubt realize, there has been an enormous amount of effort over the past decades to create a variety of open, royalty-free standards for real time data transfer between communicating entities. All are based on one of two major communication paradigms, each best suited for a certain set of requirements and use cases. The request-response interaction pattern, and its implementation in the HTTP protocol with variants created for specific situations such as bidirectional data transfer and M2M communication, is perhaps the most widely-deployed open protocol. New versions such as HTTP/2 and HTTP/3, that mitigate many of the shortcomings of the earlier versions, have the potential to replace many today's bespoke solutions with a truly universal communication protocol.

PubNub

www.pubnub.com

(415) 223-7552

@pubnub

Contact Us