Contents

Intro

kolonimys

November 2020

1 Introduction

1.1 DNS

DNS is a distributed database system that translates hostnames to IP addresses and IP addresses to hostnames (e.g., it translates hostname miles.somewhere.net to IP address 192.168.244.34). DNS is also the standard Internet mechanism for storing and accessing several other kinds of information about hosts; it provides information about a particular host to the world at large. DNS clients include any program that needs to do any of the following:

- Translate a hostname to an IP address;
- Translate an IP address to a hostname;
- Obtain other published information about a host;

Fundamentally, any program that uses hostnames can be a DNS client. This includes essentially every program that has anything to do with networking, including both client and server programs for Telnet, SMTP, FTP, and almost any other network service. DNS is thus a fundamental networking service, upon which other network services rely.

How does DNS work? Essentially, when a client needs a particular piece of information (e.g., the IP address of host ftp.somewhere.net), it asks its local DNS server for that information. The local DNS server first examines its own cache to see if it already knows the answer to the client's query. If not, the local DNS server asks other DNS servers, in turn, to discover the answer to the client's query. When the local DNS server gets the answer (or decides that it can't for some reason), it caches any information it got[34] and answers the client. For example, to find the IP address for ftp.somewhere.net, the local DNS server first asks one of the public root nameservers which machines are nameservers for the net domain. It then asks one of those net nameservers which machines are nameservers for the somewhere.net domain, and then it asks one of those nameservers for the IP address of ftp.somewhere.net. This asking and answering is all transparent to the client. As far as the client is concerned, it has communicated only with the local server. It doesn't know or

care that the local server may have contacted several other servers in the process of answering the original question.

1.2 Packet Filtering Characteristics of DNS

There are two types of DNS network activities: lookups and zone transfers. Lookups occur when a DNS client (or a DNS server acting on behalf of a client) queries a DNS server for information, e.g., the IP address for a given hostname, the hostname for a given IP address, the name server for a given domain, or the mail exchanger for a given host. Zone transfers occur when a DNS server (the secondary server) requests from another DNS server (the primary server) everything the primary server knows about a given piece of the DNS naming tree (the zone). Zone transfers happen only among servers that are supposed to be providing the same information; a server won't try to do a zone transfer from a random other server under normal circumstances. People occasionally do zone transfers in order to gather information (this is OK when they're calculating what the most popular hostname on the Internet is, but bad when they're trying to find out what hosts to attack at your site).

For performance reasons, DNS lookups are usually executed using UDP. If some of the data is lost in transit by UDP (remember that UDP doesn't guarantee delivery), the lookup will be redone using TCP. There may be other exceptions. Figure 1 shows a DNS name lookup.

A DNS server uses well-known port 53 for all its UDP activities and as its server port for TCP. It uses a random port above 1023 for TCP requests. A DNS client uses a random port above 1023 for both UDP and TCP. You can thus differentiate between the following:

- A client-to-server query - source port is above 1023, destination port is 53.
- A server-to-client response - source port is 53, destination port is above 1023.
- A server-to-server query or response - at least with UDP, where both source and destination port are 53; with TCP, the requesting server will use a port above 1023.

DNS zone transfers are performed using TCP. The connection is initiated from a random port above 1023 on the secondary server (which requests the data) to port 53 on the primary server (which sends the data requested by the secondary). A secondary server must also do a regular DNS query of a primary server to decide when to do a zone transfer.

1.3 Messages

1.3.1 DNS packet structure

All communications inside of the domain protocol are carried in a single format called a message. The top level format of message is divided into 5 sections

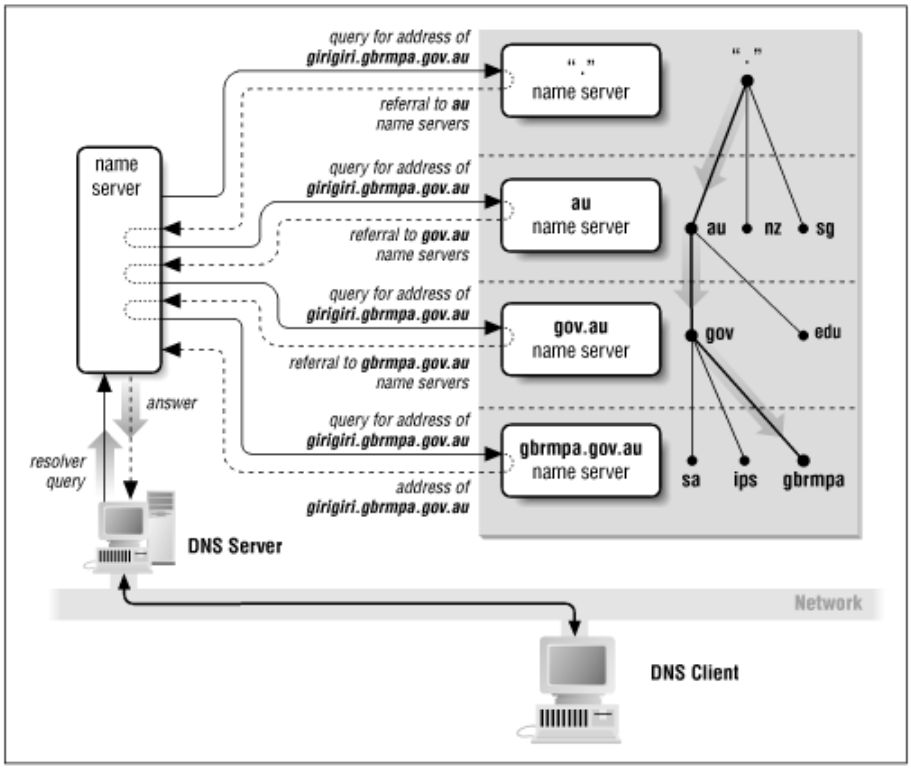


Figure 1: DNS name lookup

(some of which are empty in certain cases). Figure 2

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | RRs answering the question
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

Figure 2: DNS format

Header - The DNS header of the packet, 12 octets in length.
Question section - in this section, the DNS client sends requests to the DNS

server informing about which name it is necessary to resolve (resolve) the DNS record, as well as what type (NS, A, TXT, etc.). When the server responds, it copies this information and gives it back to the client in the same section.
Answer section - the server informs the client about the answer or several responses to the request, in which it reports the above data.
Authoritative section - contains information about which authoritative servers were used to obtain information included in the DNS response section.
Additional record section - additional records that are related to the request, but are not strictly answers to the question.

There may be several entries in the sections, or none at all. Everything is determined by the header.

1.3.2 Header section format

The header contains the following fields (3):

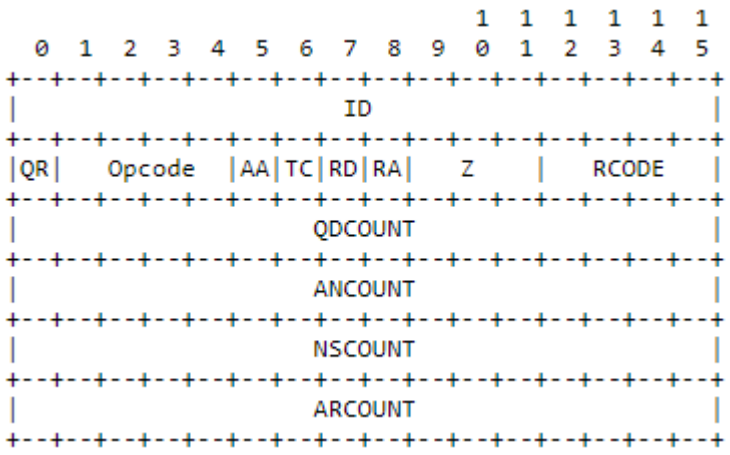


Figure 3: Header section format

where:
ID - A 16 bit identifier assigned by the program that generates any kind of query. This identifier is copied the corresponding reply and can be used by the requester to match up replies to outstanding queries.
QR - A one bit field that specifies whether this message is a query (0), or a response (1).
OPCODE - A 4 bit using this code, the client can specify the type of request, where the usual value:

- 0 - a standard query (QUERY);
- 1 - an inverse query (IQUERY);
- 2 - a server status request (STATUS);

- 3-15 - reserved for future use;

AA — A one bit, this field is meaningful only in DNS responses from the server and indicates whether the response is authoritative or not.
TC — A one bit, this flag is set in the response packet if the server was unable to put all the necessary information into the packet due to existing restrictions.
RD — A one bit, this one-bit flag is set in the request and copied in the response. If it is a flag set in the request, it means that the client is asking the server not to tell it intermediate responses, but to return only the IP address.
RA — A one bit, sent only in responses, and reports that the server supports recursion.
Z — A 3 bit, are reserved and always equal to zero.
RCODE — A 4 bit, this field is used to notify clients whether the request succeeded or failed.

- 0 — No error condition.
- 1 — Format error - The name server was unable to interpret the query.
- 2 — Server failure - The name server was unable to process this query due to a problem with the name server.
- 3 — Name Error - Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.
- 4 — Not Implemented - The name server does not support the requested kind of query.
- 5 — Refused - The name server refuses to perform the specified operation for policy reasons. For example, a name server may not wish to provide the information to the particular requester, or a name server may not wish to perform a particular operation (e.g., zone transfer) for particular data.
- 6-15 — Reserved for future use.

QDCOUNT – A 16 bit, the number of records in the query section. **ANCOUNT** – A 16 bit, the number of entries in the answers section. **NSCOUNT** – A 16 bit, the number of records in the Authority Section. **ARCOUNT** – A 16 bit, the number of records in the Additional Record Section.

1.3.3 Question section format

The question section is used to carry the "question" in most queries, i.e., the parameters that define what is being asked. The section contains QDCOUNT (usually 1) entries, each of the following format (4):
where:
QNAME - Each request and response record begins with a NAME. This is the domain name to which this record is linked or "owned". It is coded as a series of

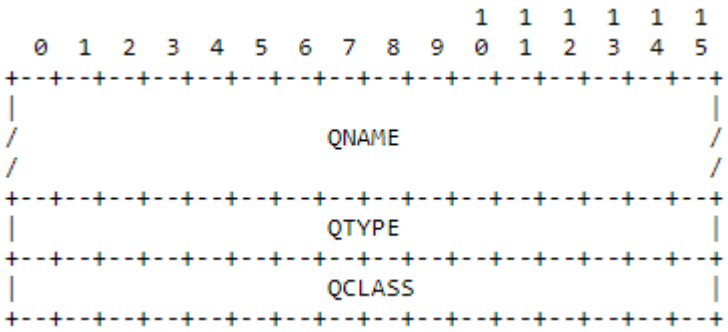


Figure 4: Question section format

tags. At this point, we should dwell in more detail. The original DNS protocol provided two types of labels, which are defined by the first two bits:

- 00 (standard label) means the remaining 6 bits define the label length followed by the given number of octets. Accordingly, the label length cannot be more than 63 bytes (For example, nslookup will display the message "is not a legal name (label too long)" when trying to resolve a host with a long label). The recording ends with the code 0x00.
- 11 (compressed label) - then the next 14 bits define a reference to the starting address of the label series. Experience has shown that it can also contain a compressed label to another address. As a rule, there are no such labels in the request.

Also, the label can contain the value 0x00 (zero length), which means that this is the root domain name (root). The maximum length for NAME is ¡= 255. This is for ease of implementation.
QTYPE - The type of DNS record we are looking for (NS, A, TXT, etc.).
QCLASS - The defining request class (IN for Internet).

1.3.4 Resource record format

The answer, authority, and additional sections all share the same format: a variable number of resource records, where the number of records is specified in the corresponding count field in the header. Each resource record has the following format (5):
where:
NAME - Same format as QNAME in the query section.
TYPE is the type of resource record. Determines the format and purpose of this resource record.
CLASS - resource record class; theoretically, it is believed that DNS can be used not only with TCP / IP, but also with other types of networks, the code

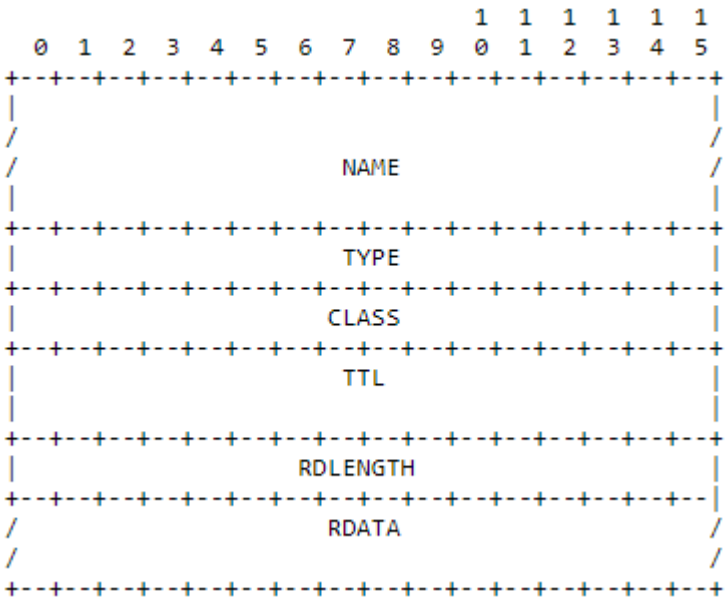


Figure 5: Resource record format

in the class field determines the type of network. Mostly IN for Internet (Code 0x0001)
TTL - (Time To Live) - allowable time for keeping this resource record in the cache of the nonresponsive DNS server.
RDLENGTH - the length of the data field (RDATA).
RDATA - a data field, the format and content of which depends on the type of record.

1.3.5 Practice[1]

Consider a packet from a real request and response. Launch your favorite sniffer and resolve habrahabr.ru.
Request

```
9b ce 01 00 00 01 00 00 00 00 00 00 09 68 61 62 72 61 68 61 62
72 02 72 75 00 00 01 00 01
```

Let's analyze the structure of the dns header.
Transaction ID = 0x9bce
Next are the flags. 01 00 is represented as a binary value 0'0000'0'0'1'0'000'0000 (hereinafter I separate the bits with an apostrophe for better visual representation of the division of the flags)

QR = 0 means this packet is a request;
OPCODE = 0000 - Standard request;
AA = 0 - this field is meaningful only in DNS responses, therefore it is always 0;
TC = 0 - this field is meaningful only in DNS responses, therefore it is always 0;
RD = 1 - Please return only the IP address;
RA = 0 - sent only by the server;
Z = 000 - always zeros, reserved field;
RCODE = 0000 - Everything went without errors
QDCOUNT = 00 01 - 1 record in the request section
ANCOUNT = 00 00 - The request always contains 0, section for responses
NSCOUNT = 00 00 - The request always contains 0, section for responses
ARCOUNT = 00 00 - The request always contains 0, section for responses
Next we have the request and response sections. With one entry.
Our first octet is 0x09, we represent it as a binary value 00'001001. The first two bits are 00, which means this is a normal label. The label is 9 bytes long (b001001). "68 61 62 72 61 68 61 62 72". These are 9 bytes. It says "habrahabr" (hexadecimal). Move on. Octet 0x02. The first two bits are 00, which means again a regular label with a length of 2 bytes. Here they are: "72 75". It is written "ru". Move on. Octet 0x00. Means the end of the host record. We got two words "habrahabr" and "ru". We combine them with a dot, we get "habrahabr.ru", this is the host that we requested.
QTYPE = 0x0001 - Corresponds to type A (request for host address)
QCLASS = 0x0001 - Corresponds to IN class.
Response

```
9b ce 81 80 00 01 00 01 00 00 00 00 09 68 61 62 72 61 68 61 62
72 02 72 75 00 00 01 00 01 09 48 41 42 52 41 48 41 42 52 c0 16
00 01 00 01 00 00 0c 90 00 04 b2 f8 ed 44
```

Let's analyze the structure of the dns header.
Transaction ID = 0x9bce. It must be equal to the ID from the request.
Flags again. 81 80 represent as binary 1'0000'0'0'1'1'000'0000
QR = 1 means this packet is a response;
OPCODE = 0000 - Standard request;
AA = 0 - Server is not authoritative for the domain;
TC = 0 - All information fit into one package;
RD = 1 - Please return only the IP address;
RA = 1 - Server supports recursion;
Z = 000 - always zeros, reserved field;
RCODE = 0000 - Everything went without errors
QDCOUNT = 00 01 - 1 record in the request section
ANCOUNT = 00 01 - Now we have one record in the answer

NSCOUNT = 00 00 - The request always contains 0, section for responses
ARCOUNT = 00 00 - The request always contains 0, section for responses
Next we have the request and response sections. With two entries. One request record, another record with a response. I will not describe the request section, it will always be 1in1 the same as in the request packet. Let's start with the answers section.
The first octet we have is 0x09, the first two bits are 00, which means an ordinary label with a length of 9 bytes. We read 9 bytes, we get "HABRAHABR". Next comes 0XC0 (b11000000). As you can see, the first two bits have a value of 11, which means that we have a compressed link in front of us. We look at the next 8 bits (this is 0x16 (b00010110)) and combine with the current last 6 bits. We get b00000000010110. A reference to the 22nd byte of the DNS packet (02 72 75 00). Starting at 22 octets, we get the labels again. By the same rules. We get it ".ru". We combine everything we have received, it turns out "HABRAHABR.ru" This is the host that will be discussed further.
QTYPE = 0x0001 - Corresponds to type A (request for host address)
QCLASS = 0x0001 - Corresponds to IN class.
TTL = 0x00000c90 - data actuality time 3216 seconds.
RDLENGTH = 0x0004 - Data length 4 octets.
RDATA = "b2 f8 ed 44".

2 HTTP — Hypertext Transfer Protocol

The World Wide Web of interlinked hypertext documents and a browser to browse this web started as an idea set forward in 1989 at CERN. The protocol to be used for data communication was the Hypertext Transfer Protocol, or HTTP.

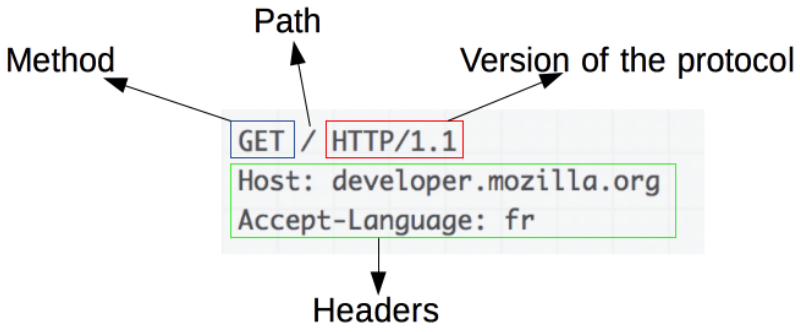
2.1 Request and Response

HTTP uses a simple request and response mechanism. When we enter http://www.mozilla.org/ into Safari, it sends an HTTP request to the server at www.mozilla.org. The server in turn sends back a (single) response which contains the document that was requested.

There's always a single request and a single response. And both requests and responses follow the same format. The first line is the request line (request) or status line (response). This line is followed by headers. The headers end with an empty line. After that, an empty line follows the optional message body.

2.1.1 Request

An example HTTP request:
Response
Requests consists of the following elements:

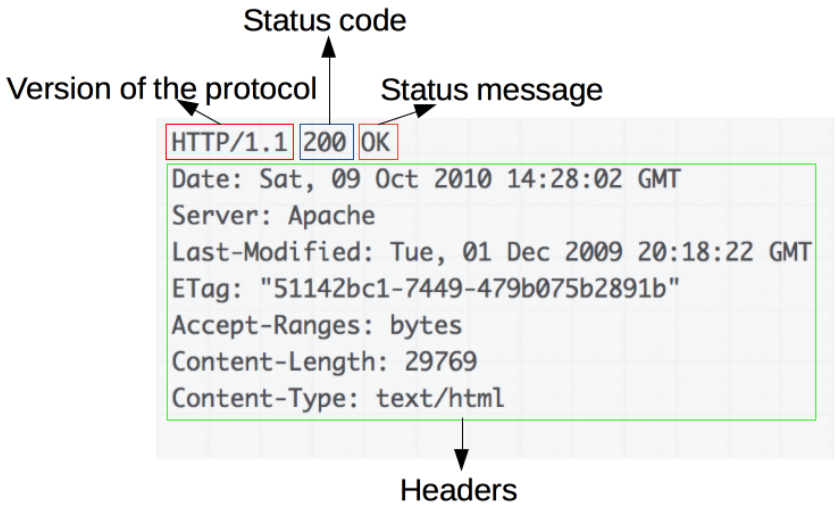


- An HTTP method, usually a verb like GET, POST or a noun like OPTIONS or HEAD that defines the operation the client wants to perform. Typically, a client wants to fetch a resource (using GET) or post the value of an HTML form (using POST), though more operations may be needed in other cases.
- The path of the resource to fetch; the URL of the resource stripped from elements that are obvious from the context, for example without the protocol (http://), the domain (here, developer.mozilla.org), or the TCP port (here, 80).
- The version of the HTTP protocol.
- Optional headers that convey additional information for the servers.
- Or a body, for some methods like POST, similar to those in responses, which contain the resource sent.

2.1.2 Response

An example HTTP response:
Response
Responses consist of the following elements:

- The version of the HTTP protocol they follow.
- A status code, indicating if the request was successful, or not, and why.
- A status message, a non-authoritative short description of the status code.
- HTTP headers, like those for requests.
- Optionally, a body containing the fetched resource.



2.2 HTTP/2

HTTP/2 began as the SPDY protocol, developed primarily at Google with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. This protocol served as a template for HTTP/2 when the Hypertext Transfer Protocol working group httpbis of the IETF (Internet Engineering Task Force) put the standard together, culminating in the publication of HTTP/2 in May 2015. From the beginning, many browsers supported this standardization effort, including Chrome, Opera, Internet Explorer, and Safari. Due in part to this browser support, there has been a significant adoption rate of the protocol since 2015, with especially high rates among new sites.

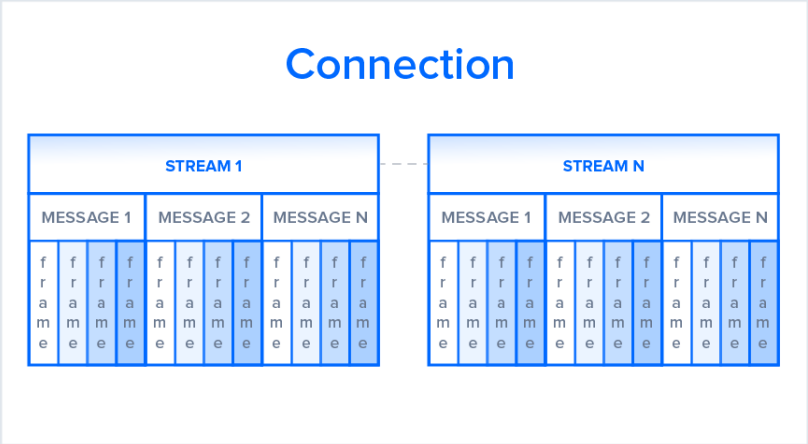
From a technical point of view, one of the most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack. As opposed to HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers. An application level API would still create messages in the conventional HTTP formats, but the underlying layer would then convert these messages into binary. This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

The conversion of messages into binary allows HTTP/2 to try new approaches to data delivery not available in HTTP/1.1, a contrast that is at the root of the practical differences between the two protocols.

2.2.1 Delivery Model

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple *streams* of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called *frames*:



At the most granular level, the communication channel consists of a bunch of binary-encoded frames, each tagged to a particular stream. The identifying tags allow the connection to interleave these frames during transfer and reassemble them at the other end. The interleaved requests and responses can run in parallel without blocking the messages behind them, a process called *multiplexing*. Multiplexing resolves the head-of-line blocking issue in HTTP/1.1 by ensuring that no message has to wait for another to finish. This also means that servers and clients can send concurrent requests and responses, allowing for greater control and more efficient connection management.

Since multiplexing allows the client to construct multiple streams in parallel, these streams only need to make use of a single TCP connection. Having a single persistent connection per origin improves upon HTTP/1.1 by reducing the memory and processing footprint throughout the network. This results in better network and bandwidth utilization and thus decreases the overall operational cost.

A single TCP connection also improves the performance of the HTTPS protocol, since the client and server can reuse the same secured session for multiple requests/responses. In HTTPS, during the TLS or SSL handshake, both parties

agree on the use of a single key throughout the session. If the connection breaks, a new session starts, requiring a newly generated key for further communication. Thus, maintaining a single connection can greatly reduce the resources required for HTTPS performance. Note that, though HTTP/2 specifications do not make it mandatory to use the TLS layer, many major browsers only support HTTP/2 with HTTPS.

Although the multiplexing inherent in the binary framing layer solves certain issues of HTTP/1.1, multiple streams awaiting the same resource can still cause performance issues. The design of HTTP/2 takes this into account, however, by using stream prioritization.

Stream prioritization not only solves the possible issue of requests competing for the same resource, but also allows developers to customize the relative weight of requests to better optimize application performance. In this section, we will break down the process of this prioritization in order to provide better insight into how you can leverage this feature of HTTP/2.

As we know now, the binary framing layer organizes messages into parallel streams of data. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each stream. The higher number indicates higher priority. In addition to this, the client also states each stream's dependency on another stream by specifying the ID of the stream on which it depends. If the parent identifier is omitted, the stream is considered to be dependent on the root stream. This is illustrated in the following figure 6.

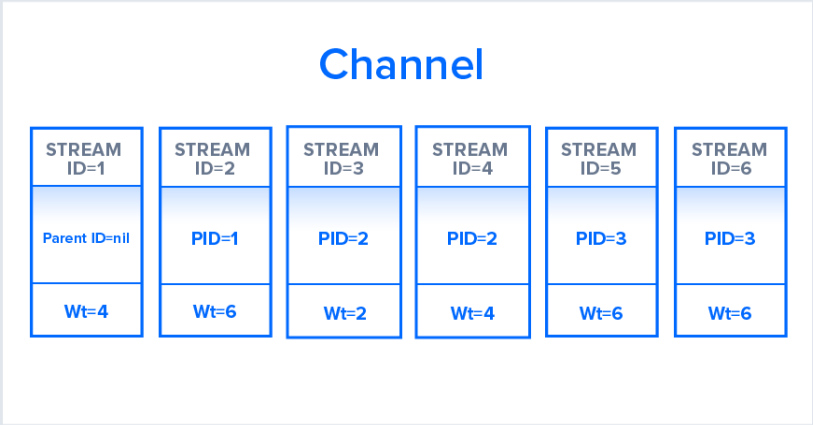


Figure 6: Channel.

In the illustration, the channel contains six streams, each with a unique ID and associated with a specific weight. Stream 1 does not have a parent ID associated with it and is by default associated with the root node. All other streams have some parent ID marked. The resource allocation for each stream

will be based on the weight that they hold and the dependencies they require. Streams 5 and 6 for example, which in the figure have been assigned the same weight and same parent stream, will have the same prioritization for resource allocation.

The server uses this information to create a dependency tree, which allows the server to determine the order in which the requests will retrieve their data. Based on the streams in the preceding figure, the dependency tree will be as follows 7.

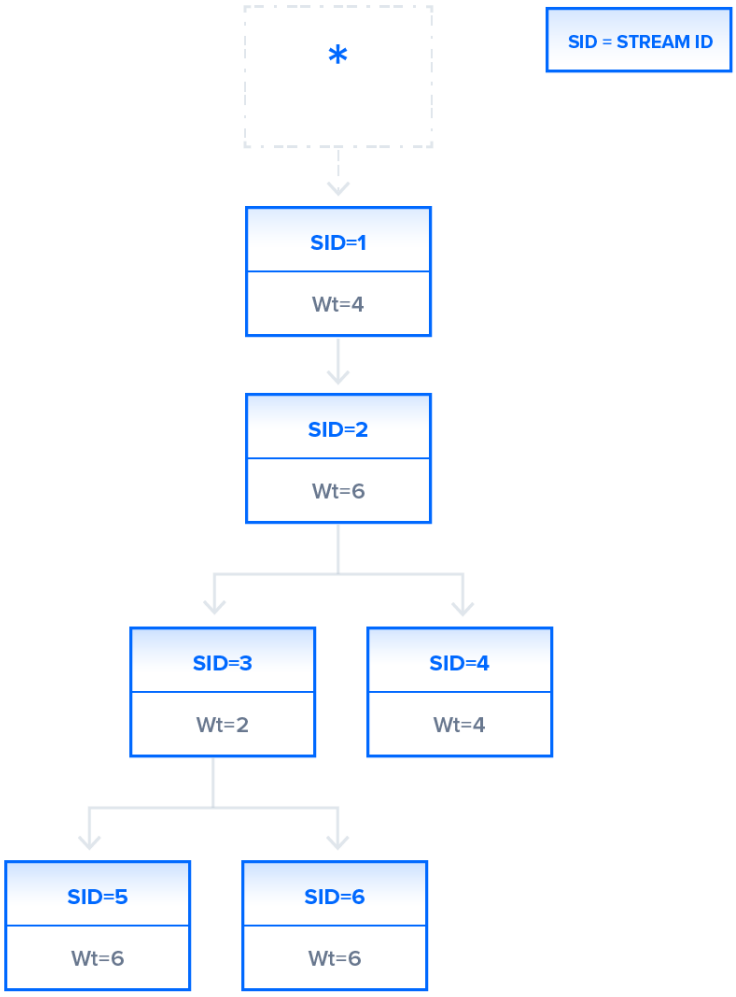


Figure 7: Dependency tree.

In this dependency tree, stream 1 is dependent on the root stream and there is no other stream derived from the root, so all the available resources will allocate to stream 1 ahead of the other streams. Since the tree indicates that stream 2 depends on the completion of stream 1, stream 2 will not proceed until the stream 1 task is completed. Now, let us look at streams 3 and 4. Both these streams depend on stream 2. As in the case of stream 1, stream 2 will get all the available resources ahead of streams 3 and 4. After stream 2 completes its task, streams 3 and 4 will get the resources; these are split in the ratio of 2:4 as indicated by their weights, resulting in a higher chunk of the resources for stream 4. Finally, when stream 3 finishes, streams 5 and 6 will get the available resources in equal parts. This can happen before stream 4 has finished its task, even though stream 4 receives a higher chunk of resources; streams at a lower level are allowed to start as soon as the dependent streams on an upper level have finished.

2.2.2 Buffer Overflow

HTTP/2 multiplexes streams of data within a single TCP connection. As a result, receive windows on the level of the TCP connection are not sufficient to regulate the delivery of individual streams. HTTP/2 solves this problem by allowing the client and server to implement their own flow controls, rather than relying on the transport layer. The application layer communicates the available buffer space, allowing the client and server to set the receive window on the level of the multiplexed streams. This fine-scale flow control can be modified or maintained after the initial connection via a **WINDOW_UPDATE** frame.

Since this method controls data flow on the level of the application layer, the flow control mechanism does not have to wait for a signal to reach its ultimate destination before adjusting the receive window. Intermediary nodes can use the flow control settings information to determine their own resource allocations and modify accordingly. In this way, each intermediary server can implement its own custom resource strategy, allowing for greater connection efficiency.

This flexibility in flow control can be advantageous when creating appropriate resource strategies. For example, the client may fetch the first scan of an image, display it to the user, and allow the user to preview it while fetching more critical resources. Once the client fetches these critical resources, the browser will resume the retrieval of the remaining part of the image. Deferring the implementation of flow control to the client and server can thus improve the perceived performance of web applications.

In terms of flow control and the stream prioritization mentioned in an earlier section, HTTP/2 provides a more detailed level of control that opens up the possibility of greater optimization. The next section will explain another method unique to the protocol that can enhance a connection in a similar way: predicting resource requests with *server push*.

2.2.3 Predicting Resource Requests

Since HTTP/2 enables multiple concurrent responses to a client's initial **GET** request, a server can send a resource to a client along with the requested HTML page, providing the resource before the client asks for it. This process is called *server push*. In this way, an HTTP/2 connection can accomplish the same goal of resource inlining while maintaining the separation between the pushed resource and the document. This means that the client can decide to cache or decline the pushed resource separate from the main HTML document, fixing the major drawback of resource inlining.

In HTTP/2, this process begins when the server sends a **PUSH_PROMISE** frame to inform the client that it is going to push a resource. This frame includes only the header of the message, and allows the client to know ahead of time which resource the server will push. If it already has the resource cached, the client can decline the push by sending a **RST_STREAM** frame in response. The **PUSH_PROMISE** frame also saves the client from sending a duplicate request to the server, since it knows which resources the server is going to push.

It is important to note here that the emphasis of server push is client control. If a client needed to adjust the priority of server push, or even disable it, it could at any time send a **SETTINGS** frame to modify this HTTP/2 feature.

Although this feature has a lot of potential, server push is not always the answer to optimizing your web application. For example, some web browsers cannot always cancel pushed requests, even if the client already has the resource cached. If the client mistakenly allows the server to send a duplicate resource, the server push can use up the connection unnecessarily. In the end, server push should be used at the discretion of the developer.

2.2.4 Compression

One of the themes that has come up again and again in HTTP/2 is its ability to use the binary framing layer to exhibit greater control over finer detail. The same is true when it comes to header compression. HTTP/2 can split headers from their data, resulting in a header frame and a data frame. The HTTP/2-specific compression program HPACK[7] can then compress this header frame. This algorithm can encode the header metadata using Huffman coding, thereby greatly decreasing its size. Additionally, HPACK can keep track of previously conveyed metadata fields and further compress them according to a dynamically altered index shared between the client and the server.

2.3 HTTP/3

There are several differences between HTTP/2 and HTTP/3, but the main one is that HTTP/3 runs over a transport layer network protocol called QUIC that uses UDP as the transport layer protocol instead of TCP, resulting on performance and security improvements.
The table below shows some of the differences between HTTP/2 and HTTP/3:

Feature	HTTP/2	HTTP/3
Header compression algorithm	HPACK	QPACK
Handshake protocol	TCP + TLS	iQUIC
Handshake negotiation	At the certificate stage via ALPN=Application-Layer Protocol Negotiation (ALPN) protocol	After certificate negotiation via "Alt-Svc:" HTTP response header
HTTP scheme	HTTP (not well adopted) / HTTPS	HTTPS
Prioritization	Yes	No, although HTTP/3 streams can have a "PRIORITY" frame to implement it

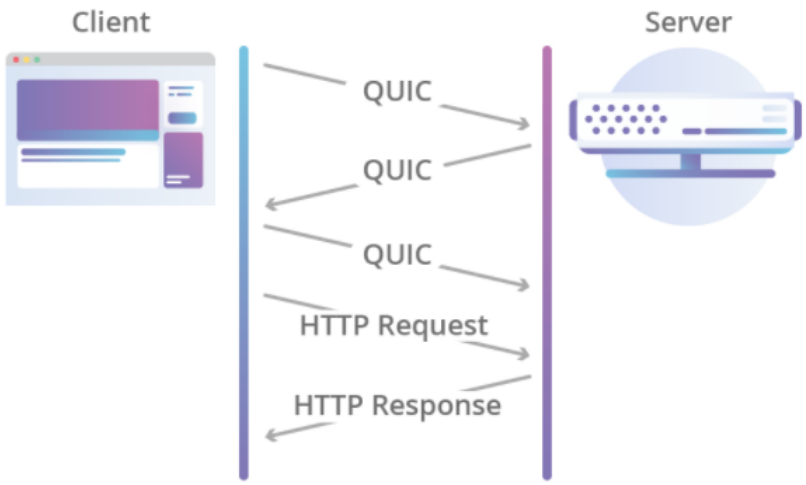


Figure 8: HTTP Request over QUIC.

2.3.1 HTTPS:// URLs

HTTP/3 will be performed using *HTTPS://* URLs. The world is full of these URLs and it has been deemed impractical and downright unreasonable to introduce another URL scheme for the new protocol. Much like HTTP/2 did not

need a new scheme, neither will HTTP/3.

The added complexity in the HTTP/3 situation is however that where HTTP/2 was a completely new way of transporting HTTP over the wire, it was still based on TLS and TCP like HTTP/1 was. The fact that HTTP/3 is done over QUIC changes things in a few important aspects.

Legacy, clear-text, *HTTP://* URLs will be left as-is and as we proceed further into a future with more secure transfers they will probably become less and less frequently used. Requests to such URLs will simply not be upgraded to use HTTP/3. In reality they rarely upgrade to HTTP/2 either, but for other reasons.

2.3.2 Bootstrap with Alt-svc

The alternate service (Alt-svc:) header and its corresponding **ALT-SVC** HTTP/2 frame are not specifically created for QUIC or HTTP/3. They are part of an already designed and created mechanism for a server to tell a client: *"look, I run the same service on THIS HOST using THIS PROTOCOL on THIS PORT*".

A client that receives such an Alt-svc response is then advised to, if it supports and wants to, connect to that given other host in parallel in the background - using the specified protocol - and if it is successful switch its operations over to that instead of the initial connection.

If the initial connection uses HTTP/2 or even HTTP/1, the server can respond and tell the client that it can connect back and try HTTP/3. It could be to the same host or to another one that knows how to serve that origin. The information given in such an Alt-svc response has an expiry timer, making clients able to direct subsequent connections and requests directly to the alternative host using the suggested alternative protocol, for a certain period of time.

An HTTP server includes an Alt-Svc: header in its response:

```
Alt-Svc: h3=":50781"
```

This indicates that HTTP/3 is available on UDP port 50781 at the same host name that was used to get this response.

A client can then attempt to setup a QUIC connection to that destination and if successful, continue communicating with the origin like that instead of the initial HTTP version.

2.3.3 QUIC streams and HTTP/3

HTTP/3 is made for QUIC so it takes full advantage of QUIC's streams, where HTTP/2 had to design its entire stream and multiplexing concept of its own on top of TCP. HTTP requests done over HTTP/3 use a specific set of streams.

HTTP/3 frames - HTTP/3 means setting up QUIC streams and sending over a set of frames to the other end. There's but a small fixed number of known frames in HTTP/3. The most important ones are probably:

- HEADERS, that sends compressed HTTP headers
- DATA, sends binary data contents
- GOAWAY, please shutdown this connection

HTTP Request - The client sends its HTTP request on a client-initiated bidirectional QUIC stream. A request consists of a single HEADERS frame and might optionally be followed by one or two other frames: a series of DATA frames and possibly a final HEADERS frame for trailers. After sending a request, a client closes the stream for sending.

HTTP Response - The server sends back its HTTP response on the bidirectional stream. A HEADERS frame, a series of DATA frames and possibly a trailing HEADERS frame.

QPACK headers -The HEADERS frames contain HTTP headers compressed using the QPACK algorithm. QPACK is similar in style to the HTTP/2 compression called HPACK, but modified to work with streams delivered out of order. QPACK itself uses two additional unidirectional QUIC streams between the two end-points. They are used to carry dynamic table information in either direction.

2.3.4 Prioritization

Priorisation between streams has been removed from the main HTTP/3 specification to be worked on separately. This was due learnings from the HTTP/2 prioritisation model and it's implementation in the real world.

A simpler prioritisation model than HTTP/2 has been proposed using HTTP header fields with a limited number of priority settings. This is a key change from the Dependency and Weight flags in the HTTP/2 Header frames and allows better understanding at the application layer.

Reprioritisation, and whether to support this, is still being discussed. HTTP/2 had Prioritisation frames to handle this but the true independent streams in QUIC and HTTP/3 makes this more complicated so the benefits versus the complexity are still being debated.

2.3.5 Server push

HTTP/3 server push is similar to what is described in HTTP/2, but uses different mechanisms.A server push is effectively the response to a request that the client never sent! Server pushes are only allowed to happen if the client side has agreed to them. In HTTP/3 the client even sets a limit for how many pushes it accepts by informing the server what the max push stream ID is. Going over that limit will cause a connection error. If the server deems it likely that the client wants an extra resource that it hasn't asked for but ought to have anyway, it can send a **PUSH_PROMISE** frame (over the request stream) showing what the request looks like that the push is a response to, and then send that actual response over a new stream. Even when pushes have been said

to be acceptable by the client before-hand, each individual pushed stream can still be canceled at any time if the client deems that suitable. It then sends a **CANCEL_PUSH** frame to the server.

3 Transport Layer Security

Transport Layer Security, or TLS, is a cryptographic protocol that protects data exchanged over a computer network. TLS has become famous as the S in *HTTPS*. More specifically, TLS is used to protect web user data from network attacks.

TLS was designed as a more secure alternative to its predecessor *Secure Sockets Layer* (SSL). Over the years, security researchers have discovered heaps of vulnerabilities affecting SSL, which motivated IETF to design TLS in an effort to mitigate them.

TLS versions and their respective RFC documents can be found in the list below:

- TLS 1.0 was published as RFC 2246[2] in 1996
- TLS 1.1 was published as RFC 4346[3] in 2006
- TLS 1.2 was published as RFC 5246[4] in 2008
- TLS 1.3 was published as proposed standard in RFC 8446[8] in 2018.

3.1 TLS 1.3 security

A core tenet of TLS 1.3 is simplicity. In the new version, all key exchange algorithms, except the *Diffie-Hellman* (DH) key exchange, were removed. TLS 1.3 has also defined a set of tried and tested DH parameters, eliminating the need to negotiate parameters with the server.

What's more, TLS 1.3 no longer supports unnecessary or vulnerable ciphers, such as CBC-mode and the RC4 cipher. These ciphers are known to be susceptible to attacks, but were still supported in most TLS implementations for legacy compatibility. Fortunately, the recent rush of downgrade attacks affecting early TLS versions motivated IETF to entirely remove such ciphers from TLS 1.3.

In addition, TLS 1.3 requires servers to cryptographically sign the entire handshake, including the cipher negotiation, which prevents attackers from modifying any handshake parameters. This means that TLS 1.3 is architecturally impervious to the downgrade attacks that affected earlier TLS versions.

Finally, the signatures themselves were also improved by implementing a new standard, called RSA-PSS[5]. RSA-PSS signatures are immune to cryptographic attacks affecting the signature schemes employed in earlier TLS versions.

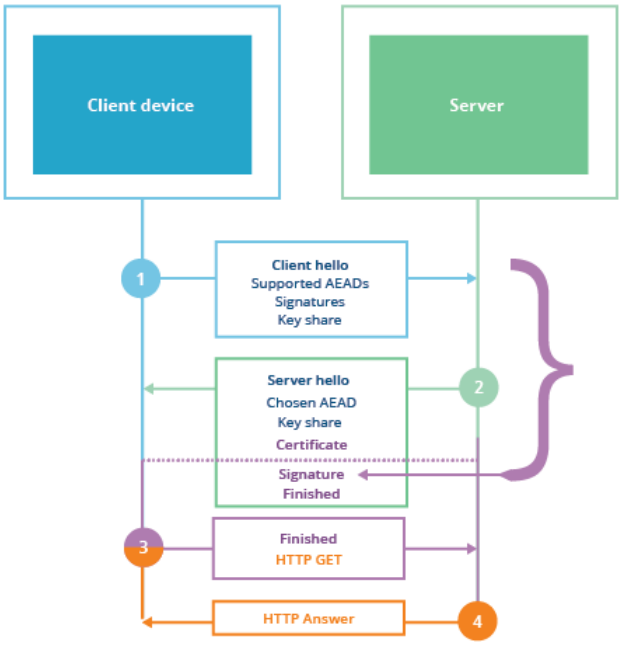


Figure 9: TLS 1.3

3.2 TLS 1.3 performance

Besides improved security, the reduced set of parameters and the simplified handshake in TLS 1.3 also contributes to improving overall performance. Since there is only one key exchange algorithm (with baked-in parameters) and just a handful of supported ciphers, the absolute bandwidth required to set up a TLS 1.3 channel is considerably less than earlier versions.

Furthermore, TLS 1.3 now supports a new handshake protocol, called *1-RTT mode*. In 1-RTT, the client can send DH key shares in the first handshake message, because it can be fairly certain of the parameters the server will use. In the rare case that the server does not support them, it can produce an error so that the client will send a different configuration.

Instead of negotiating the parameters first and then exchanging keys or key shares, TLS 1.3 allows a client to set up a TLS channel with only one round-trip transaction (instead of two, as it was previously done). This can have a great cumulative effect in the processing, power, and network resources that are required for a client to communicate with a server over TLS 1.3.

Performance optimizations don't stop here though, with another TLS 1.3 feature, called the *0-RTT Resumption mode*. When a browser visits a server for the first time and successfully completes a TLS handshake, both the client and the server can store a pre-shared encryption key locally.

If the browser visits the server again, it can use this resumption key to send

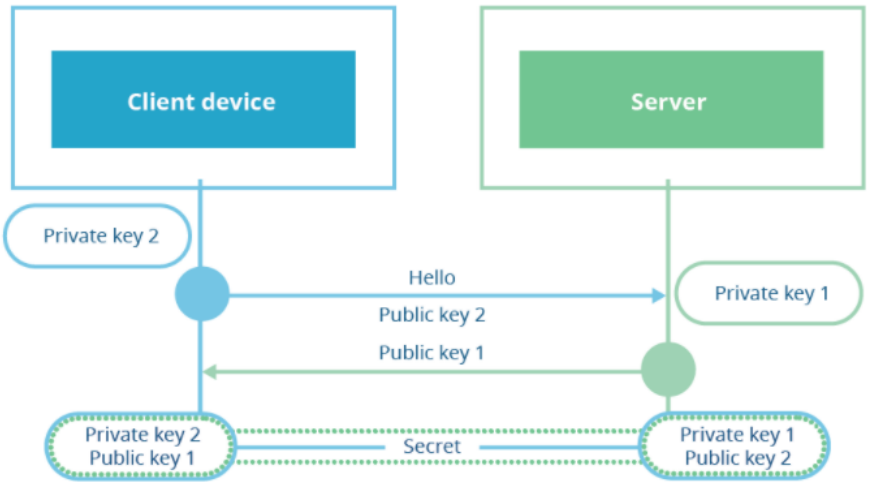


Figure 10: DH 1.3 handshake.

encrypted application data in its first message to the server. This has the same latency as unencrypted HTTP, because initial handshakes are not required.

It should be noted that there have been some security concerns[6] about 0-RTT mode in the past.

3.3 TLS 1.3 privacy

Learning from past mistakes, TLS 1.3 now offers an extension[9] that encrypts SNI information. When used correctly, this extension prevents attackers from leaking the remote server's domain name, so they have no method of tracking HTTPS user history. This feature provides greater privacy to Internet users than previous versions of TLS.

4 DNS-over-HTTPS

The DNS-over-HTTPS protocol is a recent invention. It was created a few years back and was proposed as an internet standard 2018 October. The protocol itself works by changing how DNS works. Until now, DNS queries were made in plaintext, from an app to a DNS server, using the DNS settings of the local operating system received from its network provider – usually an internet service provider (ISP).

DoH changes this paradigm. DoH encrypts DNS queries, which are disguised as regular HTTPS traffic – hence the DNS-over-HTTPS name. These DoH queries are sent to special DoH-capable DNS servers (called DoH resolvers), which resolve the DNS query inside a DoH request, and reply to the user, also in an encrypted manner.

4.1 The HTTP Exchange

4.1.1 The HTTP Request

A DoH client encodes a single DNS query into an HTTP request using either the HTTP GET or POST method and the other requirements of this section. The DoH server defines the URI used by the request through the use of a URI Template.

The URI Template is processed without any variables when the HTTP method is POST. When the HTTP method is GET the single variable "dns" is defined as the content of the DNS request, encoded with base64url.

DoH servers MUST implement both the POST and GET methods. When using the POST method the DNS query is included as the message body of the HTTP request and the Content-Type request header field indicates the media type of the message. POST-ed requests are generally smaller than their GET equivalents. Using the GET method is friendlier to many HTTP cache implementations.

These examples use a DoH service with a URI Template of "https://dnsserver.example.net/dns-query?dns" to resolve IN A records. The requests are represented as application/dns-message typed bodies. The first example request uses GET to request www.example.com:

```
:method = GET
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query?dns=AAABAAABAAAAAAAAA3d3dwdleGFtcGxlA2NvbQAAAQAB
accept = application/dns-message
```

Figure 11: DoH GET example.

The same DNS query for www.example.com, using the POST method would be:

```
:method = POST
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query
accept = application/dns-message
content-type = application/dns-message
content-length = 33

<33 bytes represented by the following hex encoding>
00 00 01 00 00 01 00 00  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 01 00
01
```

Figure 12: DoH POST example.

In this example, the 33 bytes are the DNS message in DNS wire format starting with the DNS header. The DNS query, expressed in DNS wire format, is 94 bytes represented by the following:

```
00 00 01 00 00 01 00 00  00 00 00 00 01 61 3e 36
32 63 68 61 72 61 63 74  65 72 6c 61 62 65 6c 2d
6d 61 6b 65 73 2d 62 61  73 65 36 34 75 72 6c 2d
64 69 73 74 69 6e 63 74  2d 66 72 6f 6d 2d 73 74
61 6e 64 61 72 64 2d 62  61 73 65 36 34 07 65 78
61 6d 70 6c 65 03 63 6f  6d 00 00 01 00 01

:method = GET
:scheme = https
:authority = dnsserver.example.net
:path = /dns-query? (no space or CR)
        dns=AAABAAABAAAAAAAAAWE-NjJjaGFyYWN0ZXJsYWJl (no space or CR)
        bC1tYWtlcy1iYXNlNjR1cmwtZGlzdGluY3QtZnJvbS1z (no space or CR)
        dGFuZGFyZC1iYXNlNjQHZXhhbXBsZQNjb20AAAEAAQ
accept = application/dns-message
```

4.1.2 The HTTP Response

The only response type is "application/dns-message", but it is possible that other response formats will be defined in the future. Different response media types will provide more or less information from a DNS response. For example, one response type might include information from the DNS header bytes while another might omit it. The amount and type of information that a media type gives is solely up to the format, and not defined in this protocol. Each DNS request-response pair is mapped to one HTTP exchange. The responses may be processed and transported in any order using HTTP's multi-streaming functionality

DNS response codes indicate either success or failure for the DNS query. A successful HTTP response with a 2xx status code is used for any valid DNS response, regardless of the DNS response code. For example, a successful 2xx HTTP status code is used even with a DNS message whose DNS response code indicates failure, such as **SERVFAIL** or **NXDOMAIN**. HTTP responses with non-successful HTTP status codes do not contain replies to the original DNS question in the HTTP request. DoH clients need to use the same semantic processing of non-successful HTTP status codes as other HTTP clients. This might mean that the DoH client retries the query with the same DoH server, such as if there are authorization failures. It could also mean that the DoH client retries with a different DoH server, such as for unsupported media types, or where the server cannot generate a representation suitable for the client and so on.

This is an example response for a query for the IN AAAA records for "www.example.com" with recursion turned on. The response bears one answer record with an address of 2001:db8:abcd:12:1:2:3:4 and a TTL of 3709 seconds.

```
:status = 200
content-type = application/dns-message
content-length = 61
cache-control = max-age=3709

<61 bytes represented by the following hex encoding>
00 00 81 80 00 01 00 01  00 00 00 00 03 77 77 77
07 65 78 61 6d 70 6c 65  03 63 6f 6d 00 00 1c 00
01 c0 0c 00 1c 00 01 00  00 0e 7d 00 10 20 01 0d
b8 ab cd 00 12 00 01 00  02 00 03 00 04
```

4.2 HTTP/2

HTTP/2 is the minimum **RECOMMENDED** version of HTTP for use with DoH. The messages in classic UDP-based DNS are inherently unordered and have low overhead. A competitive HTTP transport needs to support reordering, parallelism, priority, and header compression to achieve similar performance. Those features were introduced to HTTP in HTTP/2. Earlier versions of HTTP are capable of conveying the semantic requirements of DoH but may result in very poor performance.

References

[1] Alexander @admsasha. Dns packet structure, 2019.

[2] T. Dierks and C. Allen. Rfc 2246, 1999.

[3] T. Dierks and E. Rescorla. Rfc 4346, 2006.

[4] T. Dierks and E. Rescorla. Rfc 5246, 2008.

[5] et al. Moriarty. Rsa cryptography specifications version 2.2, 2016.

[6] Nick Naziridis. Network attacks and security issues, 2020.

[7] Peon and Ruellan. Hpack - header compression for http/2, 2015.

[8] E. Rescorla. Rfc 8446, 2018.

[9] et al. Rescorla. Encrypted server name indication for tls 1.3, 2019.