# 

[Overview](#_dj3wn3qumkn8)

[Client/server protocol](#_30j0zll)

[HTTP messages](#_1fob9te)

[Request Initial Line](#_3znysh7)

[Response Initial Line](#_tyjcwt)

[TritonHTTP Response codes](#_3dy6vkm)

[TritonHTTP header key-value pairs](#_1t3h5sf)

[Virtual Hosting](#_gg2iykz1oxfu)

[Tips for getting started](#_dagmfvqsx9l1)

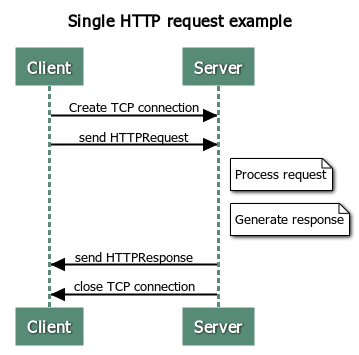
# Overview

This document describes a minimal subset of the HTTP/1.1 protocol specification. For this class, please consider this spec as the definitive guide for implementing HTTP–as such, we are going to call it *TritonHTTP*. Portions of this specification [are courtesy of James Marshall](https://www.jmarshall.com/easy/http/), used with permission from the author.

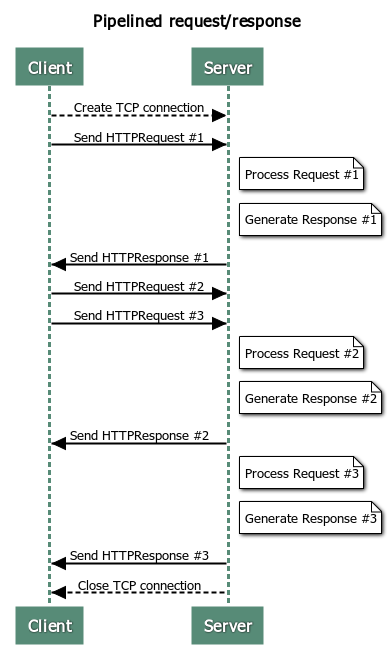
We will also be using Julia Evans’ “HTTP Zine” as well, which will be available via a link in Canvas. We are not posting the zine on a public website since it is not free--there is a cost to get a copy of the guide and our class is covering that cost for you. So please do not redistribute and don’t make it available to others. If you’d like to give the zine to other people who aren’t in the class, you can purchase individual copies by visiting<https://wizardzines.com/zines/http/>

# Client/server protocol

TritonHTTP is a client-server protocol that is layered on top of the reliable stream-oriented transport protocol TCP. Clients issue *request* messages to the server, and servers reply with *response* messages. In its most basic form, a single TritonHTTP-level request-reply exchange happens over a single, dedicated TCP connection. The client first connects to the server, sends the TritonHTTP request message, the server replies with an TritonHTTP response, and then the server closes the connection:



Repeatedly setting up and tearing down TCP connections reduces overall network throughput and efficiency, and so TritonHTTP has a mechanism whereby a client can reuse a TCP connection to a given server. The idea is that the client opens a TCP connection to the server, issues an TritonHTTP request, gets a TritonHTTP reply, and then issues another TritonHTTP request on the already open outbound part of the connection. The server replies with the response, and this can continue through multiple request-reply interactions. The client signals the last request by setting a “Connection: close” header, described below. The server indicates that it will not handle additional requests by setting the “Connection: close” header in the response. Note that the client can issue more than one TritonHTTP request without necessarily waiting for full HTTP replies to be returned.

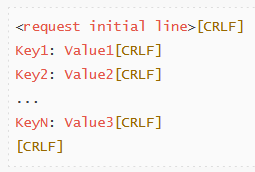


To support clients that do not properly set the “Connection: close” header, the server must implement a timeout mechanism to know when it should close the connection (otherwise it might just wait forever). For this project, you should set a server timeout of 5 seconds. If this timeout occurs and the client has sent part of a request, but not a full request, then the server should reply back with a 400 client error (described below). If this timeout occurs and the client has not started sending any part of a new request, the server should simply close the connection.

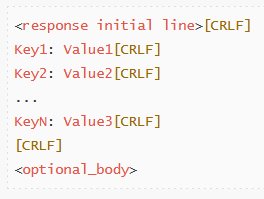
# HTTP messages

TritonHTTP request and response messages are plain-text ASCII (the body of a response can also contain binary data). Both requests and responses start with a header section. Responses optionally contain a body section which is separated from the header section by a blank line. The header consists of an initial line (which is different between requests and responses), followed by zero or more key-value pairs. Every line is terminated by a CRLF (carriage-return followed by a line feed).

A request message has this form:



A response message has this form:



Note that the optional body section is not terminated by a CRLF delimiter. Instead, the end of that body will be indicated via the Content-Length header, described below. There is no specific limit to the size (in bytes) of a request or response message, and no specific limit to the number of key-value pair headers each could contain.

## Request Initial Line

Line of a TritonHTTP request header has three components:

GET <URL> HTTP/1.1

The first keyword (GET) indicates that the client wants to download the content located at the provided URL. Real web servers support other methods such as PUT and POST, which are used to upload data to websites. We will only implement the GET method.

The URL specifies the location of the resource the client is interested in. Examples include /images/myimg.jpg and /classes/fall/cs101/index.html. A well-formed URL always starts with a / character. If the slash is missing send back a 400 error. Note that if only the / is provided, then you should interpret that as if the client requested the URL /index.html.

The version field takes the form HTTP/x.y, where x.y is the highest version that the client supports. For this course we’ll always use 1.1, so this value should be HTTP/1.1.

The fully formed initial request line would thus look something like:



## Response Initial Line

The initial line of an TritonHTTP response also has three components, which are slightly different than those in the request line:

HTTP/1.1 <Code> <Description>

The first term is the highest HTTP version that the *server* supports, in our case HTTP/1.1. The next term is a three-digit numeric code indicating the status of the request (e.g., whether it succeeded or failed, including more fine-grained information about how to interpret this response). The third term is a human-friendly text description of the return code, which can contain spaces.

### TritonHTTP Response codes

In this project we will support three response codes (two more will be shown later)

* 200 OK: The request was successful
* 400 Bad Request: The client sent a malformed or invalid request that the server doesn’t understand
* 404 Not Found: The requested content wasn’t there

## TritonHTTP header key-value pairs

After the initial request line, the TritonHTTP message can optionally contain zero or more key-value pairs that add additional information about the request or response (called “HTTP Headers”). Some of the keys are specific to the request message, some are specific to response messages, and some can be used with both requests and responses. The exact format of a key-value header is:

Key<colon>(<space>\*)<value><CRLF>

The key starts the line, followed by a colon and zero or more spaces, and then the value (each key-value pair is terminated by a CRLF delimiter). If there are spaces present simply ignore them. A few examples:

* Content-length:<space>324
  + Key: Content-length, value: 324
* Content-length:324
  + Key: Content-length, value: 324
* Content-length:<space><space>324
  + Key: Content-length, value: 324

For this assignment, you must implement or support the following HTTP headers:

* Request headers:  
  + Host (required, 400 client error if not present)
  + Connection (optional, if set to “close” then server should close connection with the client after sending response for this request)
  + You should gracefully handle any other valid request headers that the client sends. Any request headers not in the proper form (e.g., missing a colon), should signal a 400 error.
* Response headers:  
  + Date
  + Last-Modified (required only if return type is 200)
  + Content-Type (required only if return type is 200)
  + Content-Length (required only if return type is 200)
  + Connection: close (returned in response to a client Connection: close header.)

You do not need to support duplicate keys in a single request. So, for example, requests will never have two Host headers.

The format for the last-modified header is Last-Modified: <day-name>, <day> <month> <year> <hour>:<minute>:<second> <time-offset>. <time-offset> is in hours. This is the same format as the Date field. Last-Modified refers to the time the file being accessed was last modified, whereas the Date header returns the current time/date on the server. For example, we could have:

Last-Modified: Tue, 19 Oct 2021 18:12:55 -0700

The value of the Content-Type header should be determined by consulting the mime.types files included in the starter code. This file contains a list of file extensions, and the associated Content-Type value. You will need to read in this file at start time, and then consult its values to determine the appropriate type. If a file is requested whose extension is not in this file, you should use the MIME type application/octet-stream.

# Virtual Hosting

Starting with HTTP 1.1, one server at one IP address can be multi-homed, i.e. the home of several web domains. For example, "www.host1.com" and "www.host2.com" can live on the same server (IP address). This is made possible with the “Host” header field. Thus, every HTTP request must specify which host name (and possibly port) the request is intended for.

Suppose you have an AWS instance running. It will be assigned a public IPv4 address.

We can register a domain name (say, domainname1.nameserver.com) and assign it to that IPv4 address. Similarly, we can then register a second domain name (say, domainname2.nameserver.com) and assign it to the same IPv4 address.

Now, if we run a web server on our AWS instance, we can access it, either via the first hostname (domainname1.nameserver.com) or the second hostname (domainname2.nameserver.com). Both should work and both will show the user different content.

At a large organization like UC San Diego, it is common for different departmental websites to all be hosted from the same web server running on a machine with a single IP address. For example, <https://www.cs.ucsd.edu> and <https://jacobsschool.ucsd.edu/> might both share an IP address, and it is through the Host field that the web server knows whether to show the user files for the CS department or the JSOE School of Engineering.

# Tips for getting started

You are free to implement project 2 any way you want, as long as it meets the specification. However, I wanted to pass along one potential strategy that you may find helpful.

One approach you can take to the project is to develop it in pieces, so that you can debug and test each piece as you go. That way, you don’t have to try to debug a huge system, which can be very difficult!

The goal here is to make a web server that works “end to end” very quickly, even though its features are very very limited. Then you add one feature at a time, so you can test as you go.

As an example, consider the following set of milestones for your server:

1. Your server accepts incoming connections and just prints out to the screen whatever the client sends (we went over code that does this in class). This way, you should be able to see the HTTP request and headers from the client on your terminal.
2. Next, have your server simply send to the client a “static” HTTP response no matter what request it sends. For example, an HTTP 200 OK response with a Content-Type of “text/html” and an appropriate Content-Length, where the body of the response is a very simple HTML page (I had one in my slides during class, though examples of HTML documents are easily found on the web)
3. Now, instead of sending a static HTML page back to the client, generate an HTTP 200 OK response that sends the headers back as the body of a “text/html” document. In a sense, at this point you’ve implemented something very similar to server3.go!
4. Now that you have a server that simply replies to the client with the request that the client sent, you can start to work on your parsing and framing code. Have you server modify the HTML you send back to, for example, highlight the URL, or some of the headers you’re supposed to handle.
5. Once you’ve identified the URL, you can return (as an HTML document) to the client information about the file, such as whether it exists or not, or what its size is
6. At this point, you can fully implement code for 200 and 404 return types, and complete the project.

Again, this isn’t the only way to implement the project but one that some might find helpful.

###