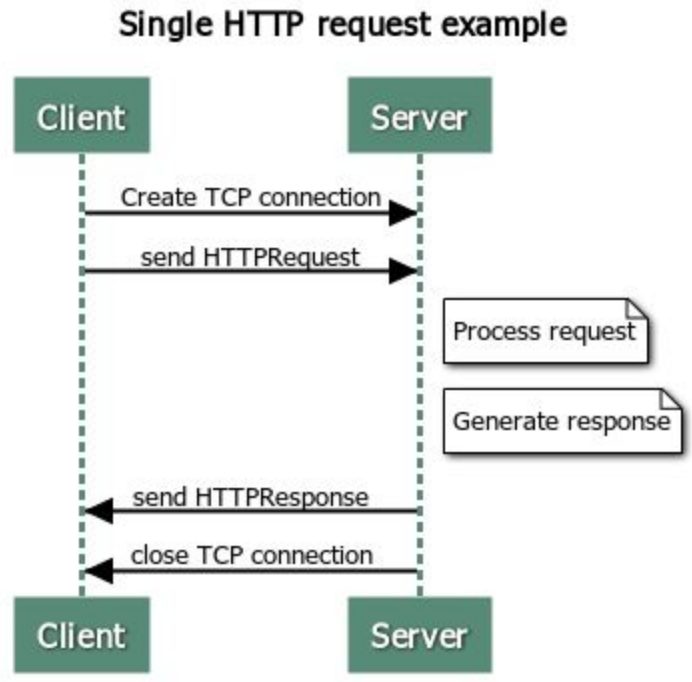
# Overview

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, 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 our course's Canvas site (canvas.ucsd.edu). 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/

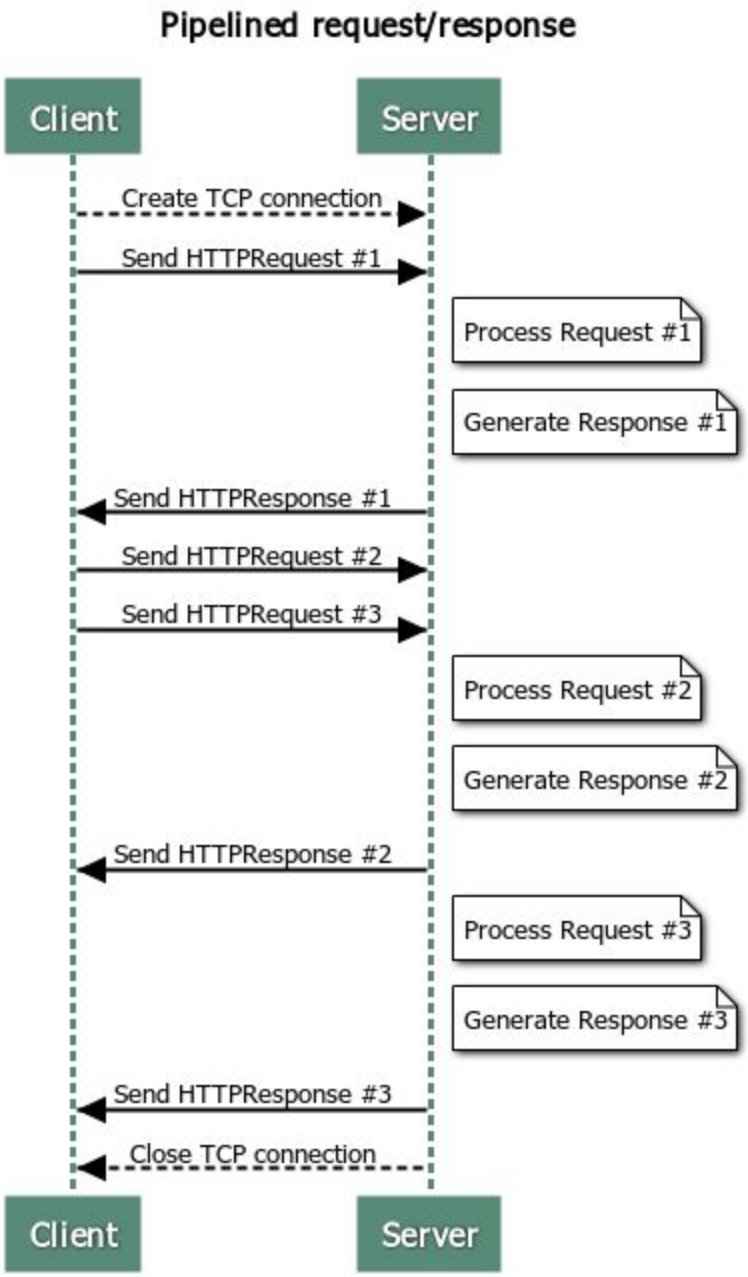
## Client/server protocol

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:

```
<request initial line>[CRLF]
Key1: Value1[CRLF]
Key2: Value2[CRLF]
...
KeyN: Value3[CRLF]
[CRLF]
```

A response message has this form:

```
<response initial line>[CRLF]
Key1: Value1[CRLF]
Key2: Value2[CRLF]
...
KeyN: Value3[CRLF]
[CRLF]
<optional_body>
```

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:

```
GET /images/myimg.jpg HTTP/1.1
```

**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 either:

`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 if return type is 200; otherwise if you create a custom error page, you can set this to ‘text/html’)
- Content-Length (required if return type is 200; otherwise if you create a custom error page, you can set this to the length of that response)
- Connection (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>. This is the same format at 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.

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.

A custom error page is simply a human-friendly message (formatted as HTML) explaining what went wrong in the case of an error; custom error pages are optional, however if you use one, the Content-Type and Content-Length headers have to be set correctly.

###