Design Notes

Some Key Requirements

What are some key requirements? Let's focus on requirements that might be tricky to implement. So we'll talk about requirements like retransmitting responses that are lost, but not about parsing messages, which should be easy. I give each requirement a name in italics. We start by referring to RFC 2616.

**Forwarding**: As stated in Section 1.4, Overall Operation, “A proxy is a forwarding agent, receiving requests for a URI ... rewriting all or part of the message, and forwarding the reformatted request toward the server identified by the URI.” I think the tricky part here is tracking the relationships between the original request and the messages that relate to it: the reformatted request, the response from the server, and the response that is sent back to the client.

**Chunking**: Section 3.6.1 Chunked Transfer Coding says, “The chunked encoding modifies the body of a message in order to transfer it as a series of chunks”. Reading and writing chunks shouldn't be too difficult, but it could get tricky to reliably identify the end of chunks being read, and report errors.

**Message length**: Several different rules determine a message's length. These include chunked encoding, a Content-Length header field and the server closing the connection. We need to carefully analyze input bytes so we can identify the ends of messages.

**FIFO sockets**: Each socket is associated with an input and output stream. As stated in 8.1.2.2 Pipelining, on a given socket responses must be sent (and received) in the same order as requests.

**Closing connections**: Persistent connections are the default (8.1.2). A connection must be closed if the server sends a close header (8.1.2), or after a period of inactivity (8.1.4).

**Reliability**: As further stated in 8.1.4

A client, server, ... MAY close the transport connection at any time. For example, a client might have started to send a new request at the same time that the server has decided to close the "idle" connection. From the server's point of view, the connection is being closed while it was idle, but from the client's point of view, a request is in progress.

This means that clients, servers, and proxies MUST be able to recover from asynchronous close events. Client software SHOULD reopen the transport connection and retransmit the aborted sequence of requests without user interaction ...

Thus, we need to buffer requests, identify those that have been handled, and retransmit those that haven't been handled when a connection closes.
Status codes are discussed in Section 10:

**Bad Request**: 10.4.1 400 Bad Request: If we receive a bad request, then we should respond with a 400 status code. If we receive a response with a 400 status code then we should discard our request and print a debug message for ourselves.

**Request Timeout**: 10.4.9 408 Request Timeout: If a client takes too long to send a request, then we should respond with a 408 status code. If we receive a response with a 408 status code then we should retransmit our request.

**Bad Gateway**: 10.5.3 502 Bad Gateway: If we receive a bad response from a server, then we should send a 502 to the client and eventually discard the request.

**Gateway Timeout**: 10.5.5 504 Gateway Timeout: If a server takes too long to send a response, then we should send a 504 to the client and eventually discard the request.

In my opinion, these are the tricky requirements in RFC 2616.

The tricky requirements that I've added include:

- **Concurrent IO**: Concurrently read from and write to sockets. In particular, a response that is received from a server should be concurrently forwarded back to the requesting client. More precisely, the proxy should start forwarding a response to the client that requested it as soon as possible after the proxy starts receiving the response from a server.

- **Concurrency and reuse**: implement both [concurrent] approaches; your proxy should take a command line (or GUI or configuration file) argument that determines which concurrency approach to use. Only the concurrency code should change between the approaches, that is, only write one implementation of the rest of the code.

Now lets think about system architecture. That is, given the requirements above, what pieces should we build this proxy from?
What are some major components?

Given the requirements above, what pieces should we build this proxy from?

**Gross flow of information**

Let's start with requirements that influence the gross flow of information in the proxy, and its relationship to the network and other systems, as these will determine its major architectural structure.

In my view, these kinds of requirements are *FIFO sockets, Forwarding, Reliability* and *Concurrent IO*. Since different requests from the client may be handled by different servers, responses may not come back to the proxy in the same order as requests were received. But since sockets are FIFO, we need to send responses back in the same order as the requests. This leads to my first proposed architectural component:

*FIFO message queues at each socket*: A pair of queues at each socket store's the socket's requests and responses, and tracks their order.

The *Forwarding* requirement involves a mapping between a request received from a client and other related messages. Since we already have queues of requests and responses, another component can map the relationships between messages:

*Forwarding mapping*: map associations between messages. In particular, maps between a request received from a client and its forwarded version.

The message queues are perfect to help implement *Reliability*, because, especially, queues containing request messages that have been sent to a server contain exactly the messages that need to be retransmitted.

Let's skip *Concurrent IO* for now, because it seems too complex.

**Input data processing**

Next let's consider the requirements that relate to the detailed processing of input data streams, as we should be able to isolate them fairly well.

One such requirement would be *Gateway Timeout*, I think. Which other requirements would you include in this set?

What data and/or components would you propose meet these requirements? At a minimum, to implement *Gateway Timeout* we need to track the time when a request is sent to a server, and note when it has take too long (against some input value) to receive the response.

Now, lets return to *Concurrent IO*. How would that fit into the architecture? Because we need to implement 2 kinds of concurrency, we need 2 sets of answers.

Also, what about *reuse*? How should we structure the concurrency to maximize reuse?