The Server

In this assignment, you are required to implement a HTTP-like server that provides key-value store services. The server consists of two components. The first is a custom protocol simplified from HTTP 1.1. It retains the basic request-response mechanism and the persistent connection feature of HTTP 1.1, but uses a different header format. On top of this protocol, an interactive in-memory key-value store with a counter store are to be implemented under the path `/key/` and `/counter/` respectively. The key-value store supports insertion, update, retrieval, and deletion operations on key-value pairs. In addition, a record can be marked as temporary by adding a counter with the corresponding key in the counter store. The counter represents the remaining retrieval times of a temporary record in the key-value store. It will be decreased by 1 for each retrieval of the record in key-value store. A record in key-value store without a counter can be retrieved/updated with arbitrary times. In contrast, a temporary record is read-only and will be deleted after the corresponding counter reaches 0. Figure 1 shows the overall architecture of the web server.

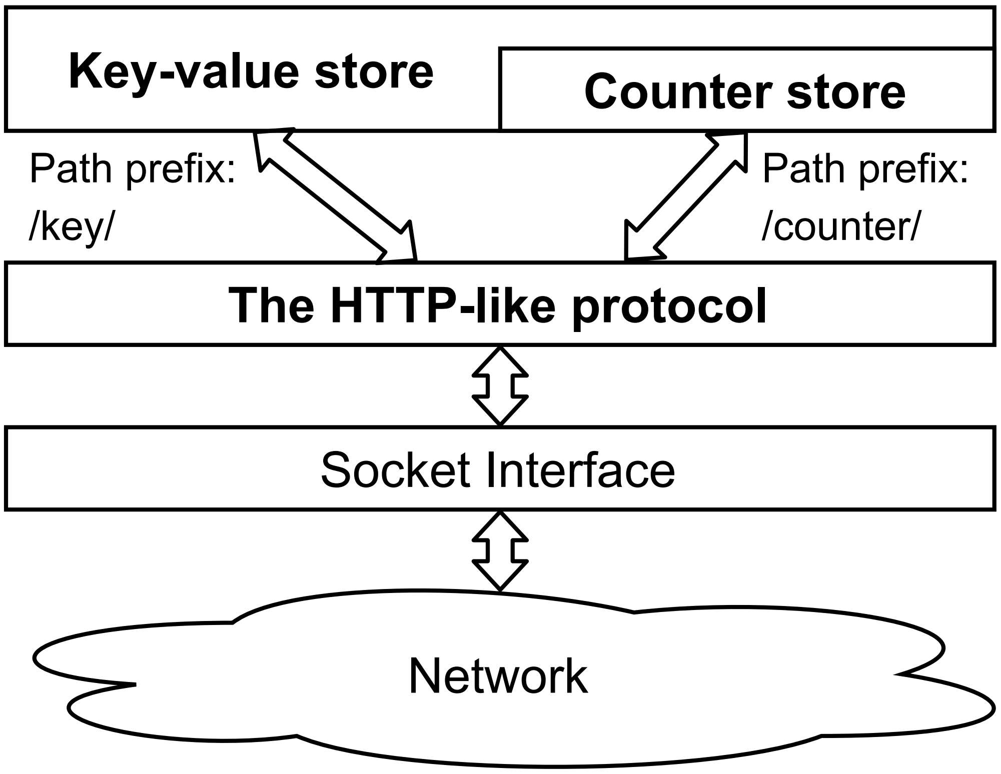

Figure 1: The layered architecture of the server. You need to implement the top two layers which are highlighted with bold fonts.

The TCP Socket

The lowest-level layer of this server is the TCP socket interface. Basically, the server should

1. `bind()` to a port

2. `listen()` to the socket and `accept()` a new connection
3. Parse and respond to client request(s) sequentially, by using `recv()` and `send()` for socket I/O.
4. Going back to 2 after the client disconnects

As discussed in the "Testing Your Program" section, port number is passed to your server as a command line argument. The grading script ensures that only one client connects to your server at any time so that no connection multiplexing or polling is required; your server can simply process connections sequentially.

It is necessary to detect disconnection events reliably, because your server is required to support the persistent connection feature originated from HTTP 1.1. If the `bytes` object returned by `recv()` is of zero length, then the connection has closed. Note that if the connection is still open, but the client has no additional data to send (eg. client is waiting for a response), `recv()` may appear to stall until the connection is closed, instead of returning a `bytes` object. The client is guaranteed to keep the connection open until either the server finishes sending all responses or a timeout occurs. Therefore, there will be no `send()` failures on the server side, unless it times out during this process.

The HTTP-like Protocol

After establishing a new TCP connection with a client, your server should read a customized HTTP request from the socket, which consists of request headers and an optional request body. Your server parses the request and sends the corresponding response as defined below. After such a request-response cycle, the connection is **persistent** (i.e. not closing immediately), and the server waits for another request from the same client until a notification of socket disconnection.

Since a real-world TCP connection could be intermittent, your server is also required to work properly when requests are delivered in chunks of random sizes with delays between chunks. Note that a TCP connection only guarantees ordering of data, but not segment sizes or delays. The test script simulates this type of intermittent connection in some test cases, while keeping the delays between two consecutive transfers below one second. To handle chunks that contain incomplete requests, it is sufficient for your server to handle data buffering well.

In addition, your server should be **responsive** in the sense that upon receiving a complete request, the server processes it and sends back the response immediately. The test script sets an one-second timeout after each request is **fully** sent to the server. If a single chunk contains multiple requests, your server is also expected to respond quickly to all complete requests inside.

To grasp the core ideas of HTTP, we omit other tedious aspects of the protocol. For this assignment, the server only needs to handle basic request and response information, as well as the `Content-Length` header field. The **header format** of this HTTP-like protocol is simplified as follows:

1. A header is a string consisting of **non-empty substrings** delimited with white-spaces (ASCII code 32). Two consecutive white-spaces mark the end of a header. You may assume that a substring does not contain white-spaces. Obviously, our header format is different from (and simpler than!) that of standard HTTP (e.g., no more `\r\n` or colons).
2. You may assume that a header consists of at least two substrings. The first two substrings (compulsory) contain information specific to requests or responses. Additional (and optional) substrings may follow. Every two substrings form a header field, where the first substring is the **case-insensitive** name of the header field, and the second is the value of this field.
3. For a request, the first two substrings (compulsory) are *HTTP method* and *path*, respectively. The HTTP method is **case-insensitive**, while path is **case-sensitive**. After these two substrings, optional header fields may follow, e.g. `Content-Length` and its value (a non-negative integer) if there is a content body in the request. The header finally ends with two white-spaces. This is followed by the content body, if any. Some sample requests are shown below:
   (a) `GET␣/key/CS2105␣␣`
   (b) `POST␣/key/CS2105␣Content-Length␣27␣␣Intro␣To␣Computer␣Networks!`
   (c) `POST␣/counter/CS2105␣Content-Length␣1␣␣3`
   (d) `GET␣/key/CS2105␣␣`
   (e) `GET␣/counter/CS2105␣␣`
   (f) `POST␣/counter/CS2105␣Content-Length␣1␣␣2`
   (g) `DELETE␣/key/CS2105␣␣`
   (h) `DELETE␣/counter/CS2105␣␣`
   (i) `GET␣/counter/CS2105␣␣`
   (j) `DELETE␣/key/CS2105␣␣`
4. For a response, the first two substrings (compulsory) are *HTTP status code* and *description of status*. The description has **no** real effects and is kept as a convention as in HTTP, and it should not have any spaces. If the response has a content body, a `Content-Length` header should similarly be included, if there's content body. The corresponding changes of state in the server are shown in Figure 2 and responses for the above 10 requests could be:
   (a) `404␣NotFound␣␣`
   (b) `200␣OK␣␣`
   (c) `200␣OK␣␣`
   (d) `200␣OK␣Content-Length␣27␣␣Intro␣To␣Computer␣Networks!`
   (e) `200␣OK␣Content-Length␣1␣␣2`
   (f) `200␣OK␣␣`

(g) `405␣MethodNotAllowed␣␣`

(h) `200␣OK␣Content-Length␣1␣␣4`

(i) `200␣OK␣Content-Length␣8␣␣Infinity`

(j) `200␣OK␣Content-Length␣27␣␣Intro␣To␣Computer␣Networks!`

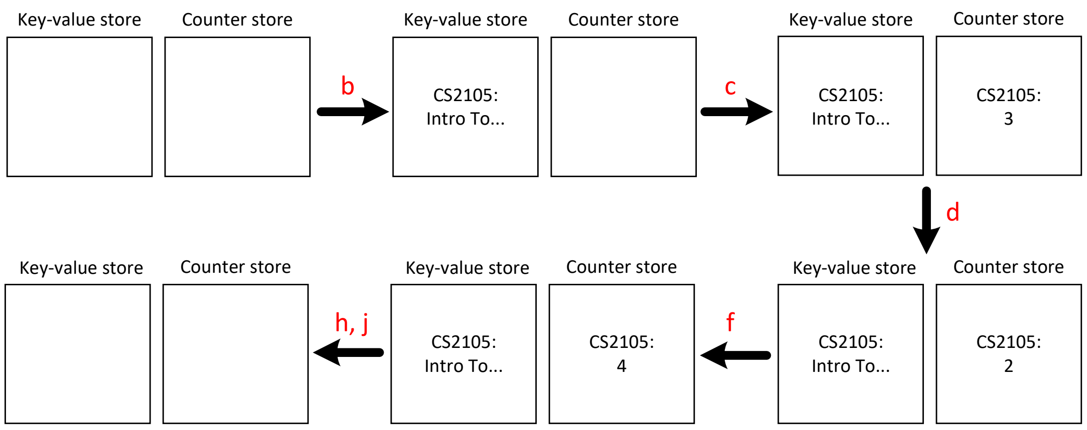

Figure 2: State of Key-value store and Counter store under the 9 consecutive requests.

The server can assume that **all requests are correctly formatted** as defined above. However, as in a real web server, a request may contain multiple header fields, and some of them may be unknown to your server (but with valid format). Therefore, `Content-Length` and its value, if exist, may not be the first header field after the HTTP method and path, and it may not be the last as well. In addition, it is guaranteed that there are no duplicate header fields. Your server should parse all header fields and ignore irrelevant ones. Moreover, the maximum size of a content body is 5MB for this assignment.

The Key-value Store and Counter Store

A key-value store can be understood as a dictionary data structure that supports insertion, update, retrieval, and deletion of key-value pairs, with keys functioning as pointers to the values. If the key does not exists in the counter store, all four operations are applicable with arbitrary times e.g. (a), (b), (j). Otherwise, the key is marked as temporary and only the retrieval operation is allowed, e.g. (g). For each retrieval, the counter with the associated key should decrease by 1, e.g. (d). For simplicity, we restrict keys to be case-sensitive ASCII strings in this assignment. Due to the use of `Content-Length` header, values can safely be any **binary string**. The server should keep all data in memory only and avoid accessing the disk.

The counter store is very similar to key-value store, except that it stores positive remaining times for the retrieval, instead of bytes value, of a particular key in key-value store. It supports operations that are highly correlated with the key-value store: **Insertion** - which inserts a key to mark as temporary and the counter which indicates the number of retrieval times that this temporary record has remaining, e.g. (c), **Update** -

which increases the existing counter value by certain positive integer, e.g. (f) **Retrieval** - which allows the client to retrieve the type of the key by getting it from the counter store which returns either the key is not found, positive counter of the temporary key, or a string value `Infinity` for a permanent key, e.g. (e) (i), and **Deletion** - which changes the temporary record back to permanent by deleting the counter in the counter store, e.g. (h). Counter keys are a subset of keys in key-value store i.e. only temporary keys should appear in the counter store. The content-length header and a positive integer content should be present in a POST.

Specifications

The key-value store should be accessible to the client through paths with prefix `/key/`. To operate on a specific key, the complete HTTP path is `/key/` appended with the actual key string. For example, a request to operate on key `CS2105` specifies `/key/CS2105` as the HTTP path. The server should support the following key-value operations over the HTTP-like protocol:

1. Insertion and update
   - (a) The HTTP request method should be `POST` and the value to be inserted (or updated) constitutes the content body. There should also be a `Content-Length` header indicating the number of bytes in the content body.
   - (b) For insertions, the server should always respond with a `200 OK` status code after successfully inserting the key-value pair. For updates, one of two possibilities will occur.
     - If there is a positive number of retrieval times in the counter store for the associated key, then the update request is rejected and will return a `405 MethodNotAllowed` code.
     - Otherwise, server should always respond with a `200 OK` status code after updating the value.
2. Retrieval
   - (a) The HTTP request method should be `GET`, and there is no content body.
   - (b) One of three possibilities will occur.
     - If the key does not exist in the key-value store, the server returns a `404 NotFound` status code.
     - If the key already exists and there is a positive number of retrieval times in the counter store for the associated key, the server should return a `200 OK` code, the correct `Content-Length` header and the value data in the content body. Then, the server should decrease the remaining retrieval times of that key in the counter store by 1. Once the number of retrieval times reaches 0, the server should delete the key from both stores.

    - Otherwise, if the key exists and is not in the counter store, the server should return a `200 OK` code, the correct `Content-Length` header and the value data in the content body.

3. Deletion
    (a) The HTTP request method is `DELETE`, and there is no content body.
    (b) One of three possibilities will occur.
        - If the key does not exist, the server returns a `404 NotFound` code.
        - If the key exists, but there is a positive number of retrieval times in the counter store for the associated key, then the delete request is rejected and will return a `405 MethodNotAllowed` code.
        - Otherwise, it should delete the key-value pair from the store and respond with a `200 OK` code and the deleted value string as the content body.

The counter store should be accessible to the client through paths with prefix `/counter/`. To operate on a specific key, the complete HTTP path is `/counter/` appended with the actual key string. For example, a request to operate on key `CS2105` specifies `/counter/CS2105` as the HTTP path. Note the keys in counter store are a subset of the ones in key-value store, i.e. temporary keys should appear in both stores. The server should support the following counter key operations over the HTTP-like protocol for a counter store:

1. Insertion and Update
    (a) The HTTP request method should be `POST` with a non-negative integer in content body. The retrieval times of a corresponding key should be inserted (or incremented). For simplicity, you may assume that the testing script will only increment up to a maximum retrieval times of **9** for any key in the counter store.
    (b) For insertion, one of two possibilities will occur.
        - If the key does not exist in the key-value store, the server returns a `405 MethodNotAllowed` code.
        - Otherwise, the server should respond with a `200 OK` status code after updating the counter.
    (c) For updates, the server should increase the existing counter value by a positive integer specified in the request, and the client always expects success status.
2. Retrieval
    (a) The HTTP request method should be `GET`, and there is no content body.
    (b) One of three possibilities will occur.
        - If the key does not exist, and the key is not found in key-value store, the server returns a `404 NotFound` code.

- If the key exists , the server should return a `200 OK` code, the correct `Content-Length` header and the remaining retrieval times for the corresponding key.
- Otherwise, if the key is not found in the counter store, but exists in the key-value store, the server should return a `200 OK` code, the correct `Content-Length` header and a string `Infinity`.

3. Deletion
   (a) The HTTP request method is `DELETE`, and there is no content body.
   (b) If the key does not exist, the server returns a `404 NotFound` code. Otherwise, it should delete the key-value pair from the store and respond with a `200 OK` code and the deleted counter integer as the content body. The `Content-Length` header should also be sent accordingly.

Grading Rubric

1. The server program is free from syntax errors and can be successfully executed (or compiled, for Java/C/C++). (1 mark)
2. Test cases (7 marks)
   (a) You will receive marks based off the test cases passed.
   (b) The starting number of marks is (-0.2) marks, before any test cases are considered. The actual marks awarded cannot fall below 0 marks.
   (c) Each numbered test case from 1 to 18 is worth 0.4 marks, for a total of 7.2 marks.
   (d) Therefore, in the event that you pass all test cases, you will receive 7 marks.
   (e) For test case 1-17, each of test cases a, b, c are worth 0.1 mark, and an additional 0.1 mark is awarded for passing all 3.
   (f) For test case 18, this is simply worth 0.4 marks.