Upset by his troubles with the Twitter board and the SEC, Elon Musk has recruited our CS4400 class to build the next great social media microblogging platform, and he's allowed use to make it U themed. You'll be implementing "Utter" (U-twitter) by building on the book's simple HTTP server program.

Getting started

Start by downloading and unpacking server_release.tar.gz Download server_release.tar.gz. When you have completed the assignment, hand in only the "utter.c" file, which contains your solution.

To build and run the initial server, which is a variant of the textbook's Tiny web server, type:

```
make
./utter <port>
```

where ‹port› is some number between 1024 and 65535. Note that only ports 2100 - 2120 are open to the outside world on the CADE machines, otherwise you will only be able to connect to it from the same subnet. After starting utter, you can connect to it using a web browser or curl on the same machine with:

```
http://localhost:‹port›/
```

The initial server replies to any request with "Hello World". The server also prints to standard output any query parameters that it received through the request, both in the URL and as POST data.

Server behavior

The server keeps track of a set of "utters" for each user and allow 4 operations:

listen (parameter: user): returns the utters for a given user

utter (parameters: user, message) posts an utter from the given user, returns a unique ID for that utter

shh (parameters: user, messageID) deletes a given post

sync (parameter: hostname, port, user) Makes an HTTP request to another server, gets all posts from that user on that server and adds them to the list of posts on this server.

You should store all the posts in memory, and will almost certainly want to make use of the provided dictionary_t type and associated functions (see dictionary.h)

Any number of clients should be able to connect to the server (up to typical load limits) and concurrently post/delete/read utters. The server starts with no posts.

Finally, your server must be highly available as described in the section "Availability and client constraints" below, which means that it is robust against slow or misbehaving clients. This requires concurrency in your server implementation. During grading, your server will be thrown a mixture of clients—fast and slow, behaving and misbehaving—all at the same time.

Your server's output to stdout and stderr is ignored, so you can use those streams for logging or debugging as you see fit.

Because printf is thread safe, it introduces synchronization, so calling printf can actually make your server appear to be threadsafe, even if it isn't. For that reason, and for speed, when we grade the assignments, we'll compile with flags that remove calls to printf by using a macro. **Look at the Makefile. There's 3 options for CFLAGS that will be helpful when developing/testing that you can switch between by uncommenting one of the CFLAGS lines.** You'll need to call make clean, and then make to switch between the various CFLAGS.

Server queries

Your server starts out with no posts.

Your server must support four kinds of HTTP GET/PUT requests that are suitable for automated clients:

- GET /listen?user=‹user› — Returns the posts made by <user>. For each post, the response body should contain the utter's ID, a space character, the text of the utter, and then a newline character. The order of the lines in the response doesn't matter.
- POST /utter?user=‹user›&utterance=‹the message› — This makes a new utter post for user. Your server needs to create a unique ID for the post and store it in memory. The ID needs to be unique across all servers, so I recommend using the server's hostname and port as part of the ID. The response body should contain the ID.

  Note that the tester scripts send the parameters in the body of the request as POST data, instead in the query string. The provided code already merges POST query data with URL query arguments for you, so you shouldn't need to do any special handling.
- POST /shh?user=‹user›&id=‹messageID› — Remove the post with the given ID from the user's posts.

  If no post with that ID exists you can return either success or error (the testing scripts don't intentionally remove posts that don't exist)

- POST /sync?user=‹user›&hostname=‹hostname›&port=‹port› — Contacts an utter server running on ‹host› at ‹port› to get all of the posts of ‹user›, and adds those posts to the user's posts on this server

  The given ‹host› and ‹port› should refer to some utter server (the test scripts run another server on localhost using a different port).

In the queries described above, ‹user›, and ‹message›should all be interpreted as UTF-8 text with the usual encoding within URLs. Basically, that means you do not have to worry about encodings as long as you use the provided parsing functions. The order of query parameters should not matter; for example, utterance might be supplied before user in a /utter request, and the server should treat that the same as user before utterance.

As an example, suppose that your server has just started running on localhost at port 8090:

```
$ curl -d "user=CS&utterance=4400" -X POST localhost:8090/utter
brocksamson.lan_8090_1

$ curl -d "user=CS&utterance=4400_000_000" -X POST localhost:8090/utter
brocksamson.lan_8090_2

$ curl 'localhost:8090/listen?user=CS'
brocksamson.lan_8090_1 4400
brocksamson.lan_8090_2 4400_000_000

$ curl -d "user=CS&id=brocksamson.lan_8090_1" -X POST localhost:8090/shh

$ curl 'localhost:8090/listen?user=CS'
brocksamson.lan_8090_2 4400_000_000
```

In the listen response above, the order of posts could have been in the other oder

Availability and client constraints

You must make no assumptions about clients of your server. It is allowed to limit the initial request line and header lines to MAXLINE characters. Otherwise, as long as clients follow the HTTP protocol, and as long as machine resources are not exhausted (including memory or allowed TCP connections), your server should continue to respond to new requests in a timely manner. Your server must not run out of resources as a result of failing to free unneeded memory or close finished connections.

You should make a good effort to report errors to clients, but it is allowed to simply to drop a client that makes an invalid request. It is allowed for communication errors cause the error-checking csapp.[ch] functions to print errors; the starting server includes exit_on_error(0) so that discovered errors are printed and the function returns, instead of exiting the server process.

As long as a client is well behaved, the server should not drop requests.

There is no a priori limit on the length of messages or time that the server must stay running. If your server runs out of memory because the given data is too large, that is allowed. If your server runs out of memory because it has a leak and cannot deal with thousands of sequential requests to access a user's posts, that is not allowed.

During grading, the efficiency of your server will not be probed by sending a bunch of simultaneous requests and making sure that posts are added and removed in a consistent manner (any post that was added remains on the server until it is deleted, after which point is no longer on returned by listen)

Your server is free to support additional queries other than the four listed in the "Server queries" section above.

Support libraries

In addition to csapp.[c/h], the provided code includes dictionary.[c/h] and parse.[c/h].

The dictionary.[c/h] library provides an implementation of dictionaries with case-sensitive or case-insensitive keys. When you add to the dictionary, the given string key is copied, but the given value pointer is added as-is. When creating a dictionary, you supply a function that is used to destroy values in the dictionary when they are no longer needed. For example, if data values are allocated with malloc, supply free to be used to destroy data values when free_dictionary is called or when the value is replaced with a different one. See dictionary.h for more information.

The more_string.[c/h] library provides string helpers and functions for some basic parsing and encoding functions. See more_string.h for details.

The provided code includes several tests:

- The simple.rkt script runs some basic checks and reports any issues that it detects. It tests utter, listen, shh by default. If you pass the --sync flag it will test sync as well. It will try to start a server on the given port, but will keep running even if that server fails to start. This is convenient if you want to start your server in another terminal to keep the output of the testing script separate from the output of your server.

  ```
  $ racket simple.rkt localhost 8090
  ```

  or

  ```
  $ racket simple.rkt --sync localhost 8090
  ```

  reports whether problems are found for the selected functionality.

- The trouble.rkt script helps check how well a server responds to misbehaved clients, and it requires the server to handle concurrent connections. Example:

```
$ racket trouble.rkt localhost 8090
```

  reports its status. If the script does not finish in 5-10 seconds, then something has gone wrong.

  The trouble.rkt script may cause your server to report many connection errors. That is expected, and you may want to redirect output to /dev/null when running this test, or use the CFLAGS in the Makefile that disable printf. There is only a problem with our server *only* if the trouble.rkt script itself reports errors. If trouble.rkt completes after printing “Making sure that trouble is done” and if your server is still running, then that test passed.

- The stress.rkt script throws many concurrent queries at a server:

```
 $ racket stress.rkt localhost 8090
```

  runs the stress test. The stress test prints some output about what it is currently doing (e.g. "posting"). It should not print any other output if it passes, and will loudly report errors.

  The stress.rkt script assumes that the server state is fresh (i.e., no posts). It will be slow if your server prints a lot of stuff. Use the CFLAGS with -fsanitize=thread to help debug synchronization issues, of the fast flags to test performance.

Evaluation

Your submission will be graded based on the following functionality thresholds:

- Your server must not have any memory leaks.
- Implementing the server well enough to handle the simple.rkt test is worth 40 points.
- Implementing the server well enough to handle the simple.rkt test with the --sync flag is worth 60 points.
- Passing the previous tests and adding concurrency well enough to handle the trouble.rkt test is worth 80 points.
- Passing the previous tests and passing the stress.rkt test is worth 100 points.

Tips

1. Pay attention to the ownership rules for values added to or extracted from a dictionary. See the "Support libraries" section and the comments in dictionary.h.
2. If you use a dictionary to map keys to NULL values, then the dictionary effectively implements a set. Note that you can specify NULL instead of a value-destroying function when creating a dictionary to indicate that values do not need to be destroyed.
3. You can have a dictionary of dictionaries, but you'll need to think carefully about what function to pass to make_dictionary to do appropriate cleanup.
4. If you get segmentation faults or if values seem to be changing out from under you, use the CFLAGS that have -fsanitize=address
5. Use functions like query_encode to ensure that an unusual user name or user string does not create trouble when embedded in a URL.
6. More generally, you do not want to be in the business of encoding, decoding, or parsing strings. If you find yourself having to parse, encode, or decode a string, check again whether functions in more_string.[c/h] could be used for the job—maybe with a slightly different approach to the communication pattern.
7. You need to use Pthread_create to enable your server to handle multiple clients concurrently, but it makes sense to add concurrency as a last step.
8. When you do add concurrency, you must make sure that uses of your shared global variables are synchronized. The stress.rkt script mostly checks your server's synchronization.