Non Blocking Networking

Note: *do not use threads* for this task.

Threads start wearing out

One of the common ways of handling multiple clients at a time is to use threads. Sounds simple enough. A client connects, we spawn a new thread to handle it, and then that thread can clean up after itself once it's done. What's the catch?

Well, that usually works okay up until a point. After that, your server won't scale as fast. And in the modern world, you have to do things web scale (TM).

Non blocking I/O

Well, what can we do about this? Maybe we could keep a thread pool, that is, have a fixed number of threads, and have them service the connections. However, it's an M:N mapping this time (M connections, N threads). But wait, how do I multiplex all these different connections, and handle all those threads?

Non-blocking I/O is your friend here. Remember how regular calls to read(), accept() etc. block until there's data or a new connection available? (If you don't, try it out! Trace how your server code blocks in read() until your client sends some data!). Well, non-blocking I/O does exactly what it says, you structure your application in such a way that works with the data that's already present (in the TCP buffers), not block for data that may or may not arrive! Functions that help you with this are `select()`, `poll()`, and `epoll()`.

Think of it as an event driven system. At a high level, you maintain a set of file descriptors (could map to files, pipes, network sockets, etc.) that you're interested in, and call the appropriate wait() function on that set of descriptors. Your program waits until one (or more) of those descriptors have some data available (in a server scenario, when a client has actually sent data). When data is available, it's like an 'event' occurred, so your

program exits the wait() call, and can iterate over the descriptors that have data, process each of them, and then go back to wait()-ing for additional data to arrive.

Epoll basics

`epoll()` arose out of the inefficiencies of `select()` and `poll()` (O(N) waiting is so 20th century) (Check out the C10k problem for more information). It provides two modes of operation, edge triggered (ET) and level triggered (LT). Think of it as follows, you have a tank (the descriptor) that you want a notification for whenever there's water (data) in it. Edge triggered mode would wake up your program once and expect you to empty out the entire tank (process all the data). If you only process half of it and call `epoll_wait()` again, your process will block (that's not good - there is data waiting to be processed and other connections to handle).

On the other hand, level triggered will wake up your `epoll_wait()` call any time there is any data in the descriptor. In this case, if you process half, and then call `epoll_wait()` again, it'll immediately return with a notification about that descriptor.

So why would we ever want to use edge triggered behavior? Well, consider what happens when there are multiple threads blocking on the same epoll descriptor (yes, we can do that; yes, people do that). Some data arrives on a socket, a thread wakes up and starts processing it. But there's still data, so another thread might accidentally get woken up and start processing data for the same descriptor. That's bad, very bad.

Edge triggered mode (together with the `EPOLLONESHOT` flag) guarantees that a single thread will handle all the data that arrived on that given socket, so (although with some additional code complexity) it's not possible that two threads accidentally 'steal' the file descriptor data from each other.

Note: You **must** use `epoll()` for this task.

The Problem

You'll be writing the client and server for a simplified file sharing application. TCP is used for everything here, so reliability is taken care of. The server uses non-blocking I/O (with epoll) to handle concurrent requests. The application supports four basic operations - `GET`, `PUT`, `LIST` and `DELETE`. Their functions are as follows:

`GET` - Client downloads (GETs) a file from the server `PUT` - Client uploads (PUTs) a file to the server `LIST` - Client receives a list of files from the server `DELETE` - Client deletes a file from the server

For simplicity, you can assume that there are no conflicting requests (that is, nobody will upload a file while someone else is downloading or deleting it, etc.)

The Protocol

This is a text-based protocol (similar to HTTP and FTP). The client sends plaintext requests along with some binary data (depending on the operation), and then the server responds with plaintext containing either error messages or optional binary data. The binary data in this case is the file being transferred, if it is a `GET` or `PUT` request. The maximum header length (header is part before data) for both the request and response is 1024 bytes. The format for the protocol is as follows:

Client request

```
VERB [filename]\n
[File size][Binary Data]
```

- VERB can be any one of '`GET`', '`PUT`', '`DELETE`' or '`LIST`' (*Case sensitive* - VERB must be capitalized).
- '\n' is the newline character (a single byte).
- There is a space character in between VERB and [filename].
- [filename] is limited to 255 bytes.
- File size and binary data are only present for a PUT operation (since the client is trying to upload a file). File size is a `size_t` which indicates how many bytes are present in the binary data in the request. For example, if file size is 32, then the server should expect 32 bytes of binary data to

be in the request from the client. For this binary data, we will be using the Little Endian form of byte ordering (the system used by Intel hardware) while sending size_t over the network. Because of this, you do not need to convert the byte ordering in either the client or server.
- On PUT, if local file does not exist, simply exiting is okay.

If VERB is “LIST”, then only the newline after will be present (no space, file size, or data).

Server response

```
RESPONSE\n
[Error Message]\n
[File size][Binary Data]
```

RESPONSE can be either OK or ERROR, depending on how the request went (details on error handling are in a later section). Error message and the newline after it are only present if and only if the RESPONSE is ERROR.

File size and binary data are only present in GET or LIST responses, and refer to the number of bytes (size_t, same way the client sent size_t in a PUT request) of binary data that follows. If it’s a GET request, the binary data is the data in the file being requested. If it’s a LIST request, the binary data is a series of filenames, separated by newlines, referring to files currently stored on the server. For example, if a server is hosting files ‘you.txt’, ‘gonna.log’, ‘give.avi’, ‘never.mp3’, and ‘up.mov’, the response to a LIST might look like this-

```
OK\n\<10 + 10 + 9 + 8 + 6, expressed as a size_t\>
never.mp3\n
gonna.log\n
give.avi\n
you.txt\n
up.mov
```

The large size_t referred to comes from the length of the filenames, plus the newlines between filenames (there is no newline after the last file, or before the first one) - the value is broken down into a sum on a per line basis for ease of understanding. In that example, the actual value that would be sent would be 43.

Specifics: Examples

In all four examples, the first line represents how we call the client in the command line, followed by the client's request to the server, and finally followed by the server's response. Notice how we **always** send all `sizeof(size_t)` as raw bytes. That is, in each below, `[size]` is 8 bytes that represent the number of bytes the binary data that follows should be. You did something similar to this in chatroom. While you're looking at these examples, think about what parts of the request and response you want to print (if any), depending on request and response types.

- GET

Here, the client is GET'ing the file "The.Social.Network.2010.1080p.BluRay.x265.10bit-z97.mp4" and saving it locally as "social_network.mp4".

```
$ ./client server:port GET The.Social.Network.2010.1080p.BluRay.x265.10bit-z97.mp4 social_network.mp4

GET The.Social.Network.2010.1080p.BluRay.x265.10bit-z97.mp4\n

OK\n
[size]...
```

- PUT

In this example, the client is PUT'ing the file "Prison.Break.S05E01.WEB-DL.x264-FUM[ettv].mp4" (local to the client) on the server as "prison_break_s05_e01.mp4".

```
$ ./client server:port PUT prison_break_s05_e01.mp4 Prison.Break.S05E01.WEB-DL.x264-FUM[ettv].mp4

PUT prison_break_s05_e01.mp4\n
[size]some call it prison break others call it privilege escalation ...

OK\n
```

- DELETE

In this case, the client DELETE's the file "prison_break_s05_e01.mp4" from the server.

```
$ ./client server:port DELETE prison_break_s05_e01.mp4

DELETE prison_break_s05_e01.mp4\n

OK\n
```

- LIST

In LIST requests, the client LIST's all available files on the server.

```
$ ./client server:port LIST

LIST\n

OK\n[size]
logan.mp3\n
laura.log\n
live.avi\n
man.txt\n
is.mov
```

Notice there is no new line at the end of the list.

Specifics: The Client

The client's job is simple: execute a single request. The usage is as follows:

`./client <server IP>:<server port> VERB [remote] [local]`

Where remote is the filename used in the request and local is the filename that the client uses while uploading/downloading. For example:

`./client 127.0.0.1:9001 GET remotefile localfile`

That would download the file 'remotefile' from the server and store it as 'localfile'.

The client runs a single command (GET, PUT, LIST or DELETE) with the chosen arguments, makes sure the file it's trying to upload (if it is uploading one) actually exists, connects to the server, sends the request (and file data, if needed), and finally prints out any error messages to STDOUT. Once the client has sent all the data to the server, it should perform a 'half close' by closing the write half of the socket (hint: `shutdown()`). This ensures that the server will eventually realize that the client has stopped sending data, and can move forward with processing the request.

For LIST, binary data from the server should be printed to STDOUT, each file on a separate line. For GET, binary data should be written to the `[local]` file specified when the user ran the command. If not created, create the file. If it exists, truncate the file. You should create the file with all permissions set (r-w-x for all users groups).

Your client is allowed to use blocking I/O, since clients don't really care about scaling. However, there are a few important things to keep in mind:

1) Parse the response carefully. When you're reading, you may accidentally read into the application binary data without realizing it (since you don't have fixed size fields in the responses).

2) `write(fd, buffer, n)` may not always write n bytes for various reasons. Buffers may become full, signals may arrive, the Skynet revolution might begin. None of which are excuses for not sending the correct amount of data to the server. Good practice is to wrap your read/write calls in a loop that runs until the specified number of bytes are read, the connection is closed, or an error (with the exception of EINTR) occurs.

3) Your client needs to be able to handle large files (more than physical RAM) and should do so efficiently.

Error Handling

Your client should handle the following errors and use the appropriate function in `format.h`:

- Received too much or too little data from server.
- Invalid response from server (malformed or nonexistent STATUS).
- Print any ERROR message from the server.

Specifics: The Server

The real fun lies here. As discussed, you'll be using epoll to allow non-blocking I/O. As you know, epoll allows you to add various descriptors to the epoll set to be 'monitored' for events. After that, when you call epoll_wait(), it will block until there are events on one or more epoll descriptors (either indicating data is available or that data can be written to the socket).

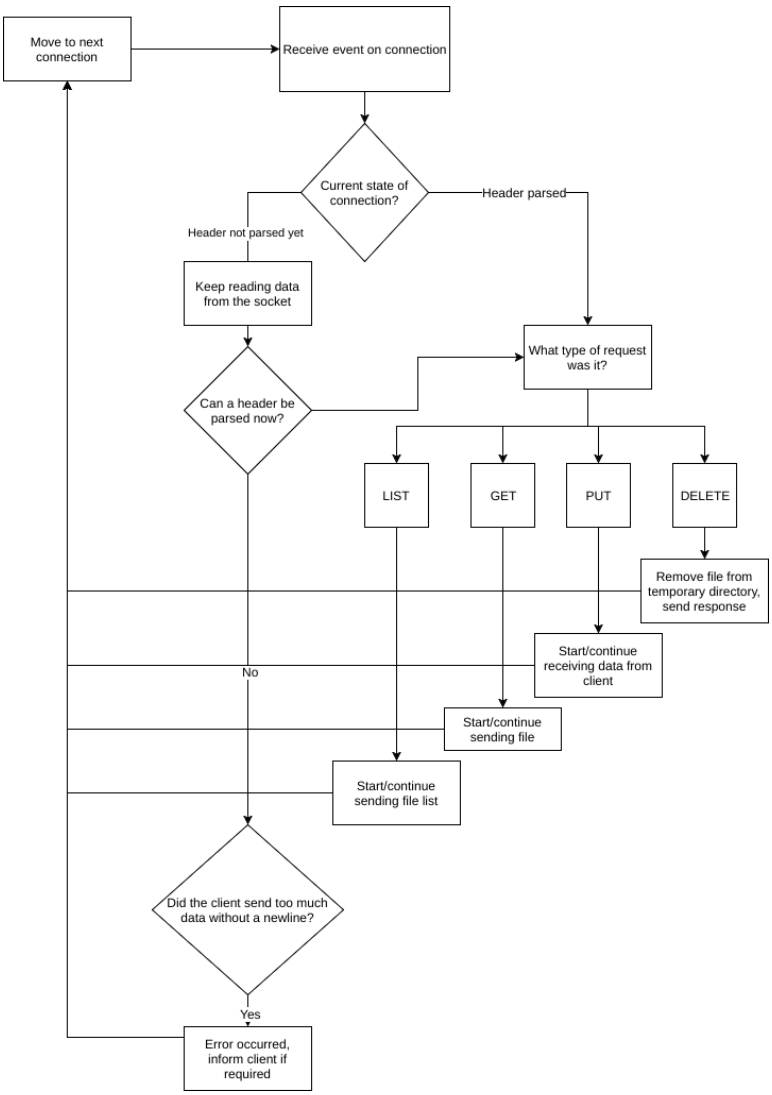
The server usage is as follows:

```
./server <port>
```

Request states

One way to reason about connections in a nonblocking server is to visualize each one as a traversal of a finite state machine. That is, there is some initial state (probably when the connection was created), and you transition between the different states depending on what action occurred (the type of request, whether there was an error or not, etc.).

A suggested flowchart for the server automata:



Maintaining persistent connection state

New connections arrive and old connections close all the time. Your server needs to know what the current status of the command it's serving is. One suggested way is to maintain a mapping from socket handle to connection

state (you are provided a dictionary data structure for this purpose). When a connection arrives, this state is allocated on the heap and added to the dictionary. When the server is handling epoll events, it can check the descriptor of the event that occurred, and quickly find out the underlying state of the request. Information that might go into a connection state could be:

- What state in the DFA you're in
- The request VERB
- Various buffers and offsets
- Filenames
- Anything else you'd like to put in there, really, this is your design decision

Global data structures

You should maintain the file list (server-side) with a global vector that gets appended to every time a file is added (using push_back()). Entries are removed one file at a time, by using vector_erase() with the appropriate index.

You might also want to maintain a map from file descriptor to connection state, as discussed above. Remember to clean up any state when you're done serving a single connection!

Memory limits

Since your server is expected to be able to serve a large number of clients multiple (possibly large) files concurrently, you cannot assume that the files will entirely fit in memory (that is, you cannot read the entire file data into one giant buffer). Instead, you should maintain some (reasonably) fixed size buffers (say, 1024 bytes), and reuse these buffers as you send or receive data over time. You may use different buffers for different kinds of data (for instance, the request header, the file list, the file being sent/received) if you wish. It is also okay to keep different buffers for different connections.

File Storage

You should create a temporary directory using the `mkdtemp()` function (make sure you follow this convention exactly!). Your server will store all uploaded files in this directory. **Immediately** after creating your directory, you **must** print it out using `print_temp_directory` (found in format.h) from the current directory (do not cd into another directory and then call `print_temp_directory` - if you don't follow this rule, the tests will fail.

When your server exits, it should clean up any files stored in this directory, and then delete the directory itself. `unlink()` and `rmdir()` might be helpful here.

Note: Be sure to use the directory name that `mkdtemp(char \*template)` gives you. Additionally, make sure that your template is *exactly* 6 X's, as in `xxxxxx`.

Exiting the server

Your server should exit on receiving SIGINT. You might find `sigaction` useful.

Note: Do not store the newlines in your filenames. There will be no whitespace or slashes in filenames at all.

Error handling

Your server is expected to be able to handle misbehaving/stark raving mad clients. That means you can never assume the request is formatted the way the protocol says it should be. While handling a request, as you read data from the connected socket into your local buffers, you should parse the command to make sure it is well-formed (the legal verb, the number of arguments is as expected, etc.)

Another thing to keep in mind is that if you try writing data to a client that has disconnected for any reason (they close()d the socket, for instance), your server might receive a SIGPIPE - you should setup your program to ignore that signal. This may also result in write() calls returning -1 with errno set to EPIPE. If this happens, your server should also close the connection and clean up any associated state.

Your server should handle these errors:

- Bad request (malformed or nonexistent verb)
- Bad file size (too much or too little data from client)
- No such file (GET/DELETE on nonexistent file)

Notes:

- If a PUT request fails, delete the file.
- If a PUT request is called with an existing file. overwrite the file.
- You should use the error messaged defined in `format.h`

Writing your server code

Keep things modular! Write functions for everything. This has multiple advantages. First, it lets you debug your code in small, incremental units, rather than writing a huge monolith of code and trying to figure out which part of it is broken through trial and error. Secondly, you'd be surprised how much code you can end up reusing if you design your application appropriately. Third, it'll be helpful while debugging or discussing your approach with course staff - it's really hard to tell what your code is supposed to be doing, otherwise.

Reusable Addresses and Ports

Make sure you use `SO_REUSEADDR` and `SO_REUSEPORT` to ensure `bind()` doesn't fail in the event that your server or client crashes. This will enable faster debugging for you (otherwise, you would have to wait for the kernel to reopen the source address and port). We will be making sure that your socket is set up with these options (look into `setsockopt`) so please make sure you use both options! If you don't, you will not pass this assignment.

See this StackOverflow question for more information on the differences between the two and why they are necessary.

Logging

This assignment is challenging enough, and debugging it is even more so. We recommend that you constantly log state as your client/server program

executes. You might want to log at the beginning/end of function calls, entry and exit of loops etc. and maybe log the values of key variables, pointers, file descriptors etc. to sanity check what's going on with your code.

We provide a simple `LOG()` macro, if you're interested in that. Or, you could play around with writing one of your own (ours is just a wrapper around `fprintf()`). If you decide not to use this macro, ensure you log only to stderr. The stdout of your client and server implementation are used for testing purposes so writing to stdout should be deliberate. Any unwarranted new line characters or extraneous logging within stdout could result in a test failure.

Testing

Testing networking programs can always be challenging. There are a few ways we suggest going about this:

Client - You could write a toy server in any language of your choice that logs the adheres to the protocol above (even if it doesn't do multithreading, nonblocking I/O etc.)

Server - By the time you start your server, you will (hopefully) have a working client implementation. Use that to test your server! Another way to simulate multiple clients could be to write a program that fork()s a bunch of times, each child calling exec() on your client executable and sending a request (no fork bombing, please!).

Alternatively, if higher level languages are more your thing, you could try writing a script in some other language (say, Python or Ruby). As long as you strictly adhere to the specified protocol, it should work fine (be careful about the width and byte ordering of types in other languages, though!). The catch is, you have to be sure your mock client/server actually works as expected, since you'll end up debugging programs in different languages at this point, which is never fun. On the bright side, this lets you practice multilingualism.

We will also be providing a reference client and server. These print out helpful logging messages that you do not need to mirror in your code. These might also not be perfect, so please report things to us (they do pass our tests though).