Server Design

The server will use threads in two ways:

- Incoming connections and message requests from clients will be put onto a *thread pool* that handles all such actions (with appropriate functions that respond to each, i.e. create a client fd and put into server structs for incoming connections, or send a reply with message requests); because compared to streaming data from each station, the workload from client requests is relatively light, we can use a thread pool to handle each request.
- For each station, a thread will be used to continuously stream music to each connected client's UDP port. The logic for this can be seen in `src/util/station.c` : we first read a chunk of data (here 1 KiB), send to each client, then sleep for `0.0625s` . This way, in one second, we send 16 chunks of data; equivalently, `16KiB/s` , as required.

[Question]: I originally wanted to incorporate stations streaming music into the thread pool (above); however, I reasoned that since each station needs to send data at a fixed bitrate (`16KiB/s`), putting the work onto a thread queue might delay the data, resulting in inconsistencies in the streaming bitrate. Is this actually a concern? Or would not including the work into the thread pool only lead to more inconsistencies?

Note that listening for `snowcast_control` TCP requests will *not* need a thread for every client; instead, the server will utilize the `poll(2)` system call on a single thread to monitor every socket file desriptor. This way, the server can asynchronously poll clients for messages, then handle each message by offloading onto the thread pool. This will (hopefully!) enable the server to rapidly respond to TCP requests.

- **[Question]**: how does `poll` fare with an arbitrarily large number of connections? Beej's Socket Programming guide notes that `poll` is incredibly slow with an "extremely large amount of sockets", but what is this number actually? Should I worry about it and choose a different approach (an external library? one thread per client?), or is `poll` okay?

Client connections will be represented by the following struct:

```
typedef struct {
  list_link_t link;      // for the doubly linked lists
  int client_fd;         // TCP connection
  struct sockaddr *addr; // UDP address [TODO: "localhost"?]
  socklen_t addr_len;    // length of address
  uint16_t udp_port;     // UDP port [TODO: is this necessary?]
} client_connection_t;
```

The `link` will allow the struct to be placed on a linked list (necessary for stations, as we'll see below). The `client_fd` will represent the socket file descriptor of the client. `addr` and `addr_len` will be necessary for the `sendto` call; and `udp_port` may be necessary to send data (actually, I don't think so, but might as well store it?). These client connections will be stored and controlled on the server in the following struct:

```
typedef struct {
  client_connection_t *clients; // currently connected clients
  struct pollfd *pfds;          // client file descriptors to poll
  uint16_t num_clients;         // keep track of number of clients
  uint16_t max_clients;         // record current max size of the array
  pthread_mutex_t clients_mtx;  // synchronize access to client control
} client_control_t;
```

A full description is in the `src/snowcast_server.h` file; I won't repeat it here (although please offer feedback! I'm not sure if this is a sufficient or necessary approach).

Stations will be represented by the following struct:

```
typedef struct {
  uint16_t station_number; // unique number for a station
  char *song_name;         // name of a song
  FILE *song_file;         // file to read the song
  char buf[CHUNK_SIZE];    // buffer to store processed song chunks
  sync_list_t client_list; // list to store clients connected to this station
  pthread_t streamer;      // streamer thread
} station_t;
```

The uses are explained in the comments; `streamer` will be responsible for streaming music indefinitely in a loop until the server quits. Note that `song_name` should be the file name of the appropriate song; no extra parsing is done currently. `client_list` stores a synchronized list (see `src/util/sync_list.h` for its entire implementation; it essentially wraps the `typedef` 'd doubly linked list macro definition in a `mutex`) of clients to which to stream. The client control structures and the station's clients are synchronized by mutexes, so multiple threads can attempt to access and modify them if necessary (i.e. to add or remove clients).

The server has a representative structure

```
typedef struct {
  station_t **stations;       // available stations
  uint16_t num_stations;      // keep track of number of stations
  pthread_mutex_t server_mtx; // synchronize access to server
  pthread_t server_repl;      // thread to run the server REPL
  uint8_t stopped; // flag for server condition: 0 -> running, 1 -> stopped
  // pthread_cond_t server_cond; // condition variable FOR EXTRA FUNCTIONALITY
  // thread_pool_t th_pool; // [TODO: implement a thread pool]
} snowcast_server_t;
```

This way, when a client sends a request to change stations to the server, the server can access this centralized struct in a synchronized manner (by acquiring the `server_mtx` lock), then switch the client from one server to another. Note that since a `sync_list_t` is used to store clients in each `station_t`, adding and removing from the client list is done in a synchronized manner, preventing data races. As long as this `snowcast_server_t` is the only point of access to modifying each individual `station_t` struct, any change station request must first lock the `snowcast_server_t` structure, which prevents any other thread from performing a modification to the existing `stations`' client lists.

In the future, I'm considering swapping from a single `server_mtx` to a dynamic array of `pthread_mutex_t station_mtxs[num_stations]`; this way, we have a finer-grained locking scheme that'll allow more threads to modify stations' client lists at once.

- **[Question]**: Does either implementation actually prevent data races? To my understanding, access to each station's client lists is synchronized, but it's also a rather naive and coarse-grained locking scheme; I may be missing some important concurrency principles.

Also, in general, my client, station, and server structures are rather rudimentarily designed; any feedback on anything that might be missing or anything that'd make it more efficient would be greatly appreciated!