CS 7345 Sorting Library- Client/Network API Documentation:

a. **Overview:** This library allows for the creation of a user-specified vector and the running of several sorting algorithms on any vector of numbers passed into it. The vector can be specified to be in reverse order (from how it was created), completely random-shuffled (if the specified array was not already in that order), and duplicated (80% duplicated, which tests edge cases of many sorting algorithms and is thus also typically used as a benchmark).
    i. **User Input:** The array can be of any length (at least, any before the hardware/browser completely runs out of memory to supply) and **must contain only integers.**
    ii. **Sorting Algorithms:** Bubble, Merge, Insertion

b. **Endpoints/Callable Functions:**
    i. The networking version of this API simplifies functionality to passing messages between two types of clients- one to perform sorting algorithms, and one to return and post results- and a central server that coordinates work requested by the "posting" client to the "performing" clients. **No functions of the server are explicitly callable- it will only respond to messages sent from either the RequestingClient or the SortingClient.** *Only the methods of the Requesting Client should/can be explicitly called or are otherwise exposed to the end user. Sorting Clients can only be created- and will then work on their own beyond that.*
    ii. *Requesting Client:*
        1. The Requesting Client is used to begin networked sorting jobs. It holds a container of arrays/vectors to be sorted and, as the server asks for work, it provides from this container until it runs out of work to give.
        2. RequestingClient(url urlOfSocket);
            a. Instantiates the Requesting Client connected to a given socket. Client will be internally remembered as "Requester" because it will open a given job to send to the server. **This must be called before other methods can be used.**
            b. *Example Call:*
                i.
```
RequestingClient
visualizer("ws://localhost:8080/");
```
            c. *Sample Output:*
                i. Creates a new "RequestingClient" object with the given url as the url for its socket. In this case, the url is ws://localhost and the port is 8080.
        3. void sendMessage(vector arrayToSort, string method)
            a. Sends a vector to be sorted to the server (on the previously instantiated socket). This vector will be sorted using the method defined in the method parameter.
            b. *Example Call:*
                i.
```
visualizer.sendMessage(array, "Bubble");
```
            c. *Sample Output:*

        i. This method sends a json message of the following form to the server connected to the same socket:
            1. type: “ToSort”
            2. method: “Bubble”
            3. array: array
            4. size: 10

    4. void addJob(vector new_job, string method);
        a. This method adds a new sorting job to the internal job “queue” contained within the RequestingClient object.
        b. *Example Call:*

```
visualizer.addJob(array, "Bubble");
```

        c. *Sample Output:*
            i. This job is added to the queue as a vector of length 10.
            ii. It will eventually be sorted with the sorting method “Method”

iii. *Sorting Client:*
    1. SortingClient(url urlOfSocket);
        a. Instantiates the Sorting Client connected to a given socket. Client will be internally remembered as “Client” because it will receive a sorting job from the server and sort it.
        b. *Example Call:*

```
SortingClient
client("ws://localhost:8080/");
```

        c. *Sample Output:*
            i. Creates a new “SortingClient” object with the given url as the url for its socket. In this case, the url is ws://localhost and the port is 8080.

c. **Design Patterns/Philosophies:**
    i. **Strategy Design Pattern:** This library utilizes the Strategy Design pattern, allowing the main Requesting Client, through the use of messages that it passes, to seamlessly select the algorithm that is to be run at runtime (as opposed to hard-coding it). This is done by assigning the message a “method” parameter that tells the server which type of client to run for work.
        1. **Pros:**
            a. This strategy pattern allows the library to be easily expanded and extended (another emscripten-bound c++ sorting method could very easily extended to Javascript by simply adding another message-listener (which checks the “type” parameter of the message passed to it) in the Sorting Client and pointing it to the new type of sort to be run, as all sorting algorithms (strategies) take the same input (a single vector) and yield the same output (a single, sorted vector).
            b. All algorithms are bound to the client objects, so the only instantiations required are the Requesting Client (to send

messages with), the correct number of Sorting Clients, and the vector object that will be sorted.

2. **Cons:**
    a. Like all Strategy patterns, the user must be aware of each message-listening block of code to use it (to use a Strategy, the user must know it exists).

d. **Server/Client Functionality:**
   i. The Server and Client(s) communicate with each other through messages of specific types with specific parameters. A sorting job is internally parsed as the following:
      1. The server, when it has enough clients, sends a “ready” message to the Requesting Client, which is sent as the following, as a JSON object:
         a. type: “ClientsReady”
      2. The Requesting Client sends a message to the server, which is currently listening on a given socket for jobs. This message is sent as the following, as a json object:
         a. type: “ToSort”
         b. method: The sorting method that will be used on the passed vector
            i. This can be “Bubble”, “Merge”, or “Insertion”
         c. array: The actual vector that is to be sorted
         d. size: The size of the vector that is passed
      3. The server, upon receiving the message, will check the number of currently-available “Client” clients connected to the socket. If there are more than 0 clients, it will divide the vector into similarly-sized chunks and distribute it to each Client. The message that is passed to each Client is sent as:
         a. type: The sorting method that the client is instructed to use. It will be intrinsically saved from the “method” parameter of the message it received from the Requesting Client.
            i. This can be “Bubble”, “Merge”, or “Insertion.”
         b. array: The actual vector that is to be sorted, passed directly from the message it received from the Requesting Client.
         c. size: The size of the vector that is to be sorted, passed directly from the message it received from the Requesting Client.
      4. Each Client, when finished running the correct algorithm, will then send their piece of the vector back to the server in the following format:
         a. type: “Sorted”
         b. array: The now-sorted vector.
         c. method: the sorting method that was used (Bubble, Merge, or Insertion)
         d. size: The size of the vector that is to be sorted, passed directly from the message it received from the Server.

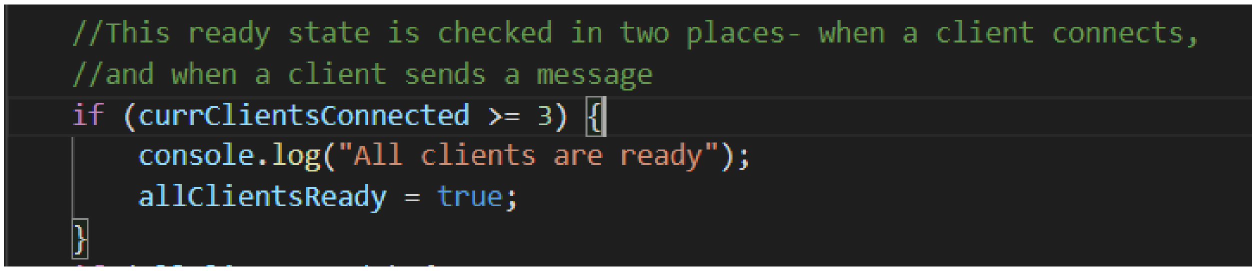
5. Finally, the server will stitch together all the client's pieces of the array and send it back to the Requesting Client. This message will be in the following form:
    a. type: "Sorted"
    b. method: the sorting method that was used
    c. array: The now-sorted vector.
    d. size: The size of the vector that is to be sorted, passed directly from the message it received from the Requesting Client.
6. The Requesting Client will then display the time that it took for the server to finish sorting the array- calculated by the difference in time between when Message #2 (sending a job to the server) was sent and Message #5 (receiving the sorted array from the server) was received.

ii. This means that to effectively run the library, each object must be created in the following order.
1. The server- it must exist for the RequestingClient and SortingClient(s) to connect to.
2. The requesting client- it must be waiting with a task to give, when the server tells it that it is ready.
3. The sorting clients- they will be notified of work by the server after enough clients (3, in this case) are online and connected.
    a. It is advised to wait until the requesting client finishes creating all of its jobs and prints an "All jobs added to queue." Message to screen, as connecting clients before this state can cause undefined behavior in the Requesting Client, as it will be attempting to read from a non-fully-initialized job queue.
    b. **Note:** that the two clients are served up using a Python script.

e. **Other Design Decisions:**

i. **Use of Vectors:** Unlike arrays, which must be manually resized to change their size, vectors are able to be quickly imported to any language and are commonly used. Because the runtimes of all a vector's factory operations are universally understood, it can very easily serve as a base for numeric operations. As vectors are also a templated type, the library, itself, can also be very easily templated (even for completely-user-defined classes, provided operations such as the equality operator are also defined for those classes).

ii. **Networking Architecture:**
1. The server-client architecture was created to allow jobs to be adaptable and scalable across a variable number of clients, as on certain types of networks, it is infeasible to expect that *three* clients would always be available. Thus, when the server decides how to split an array, it uses the number of currently connected clients as the metric for splitting up work. To amend this for different numbers of clients, this line can be changed from 3 to any number of desired clients:



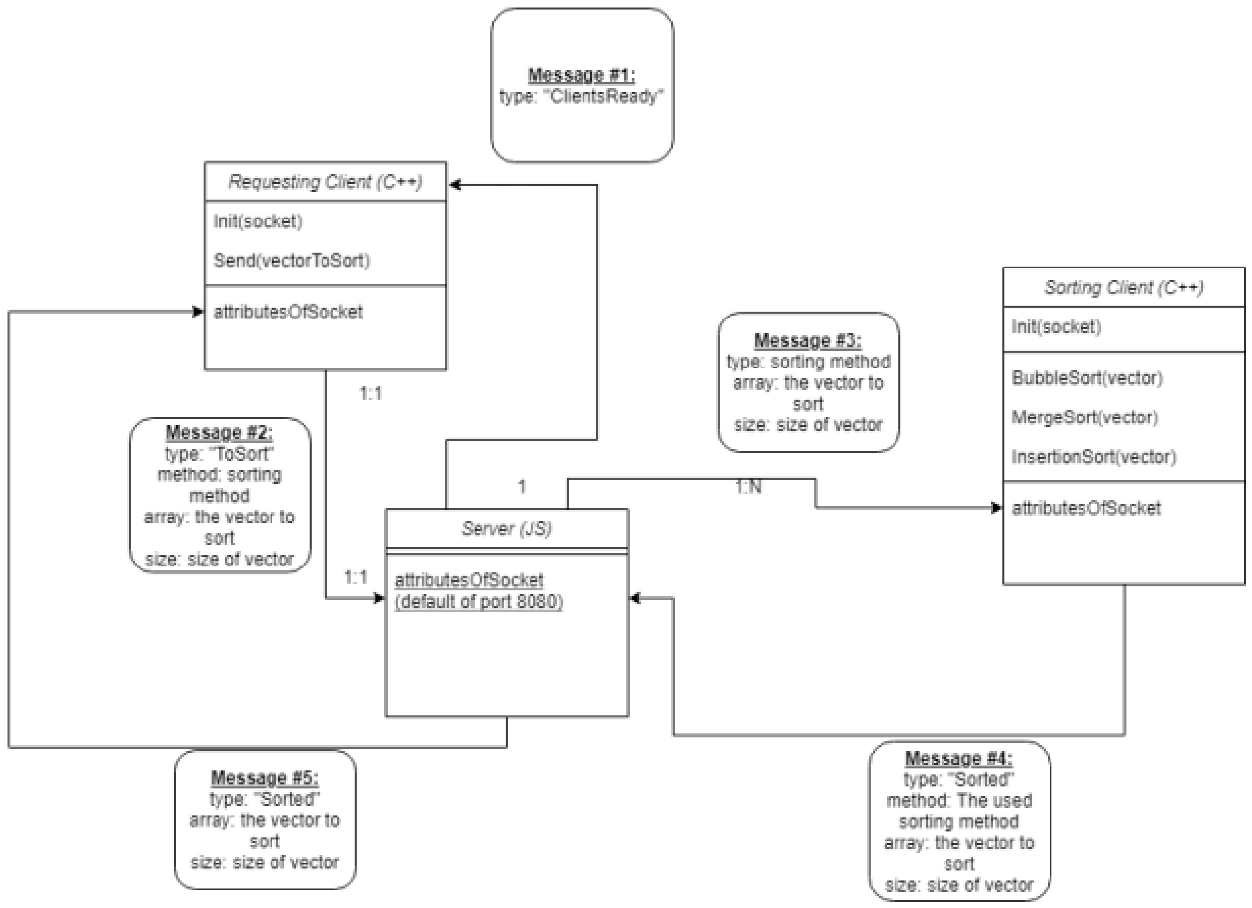
```
//This ready state is checked in two places- when a client connects,
//and when a client sends a message
if (currClientsConnected >= 3) {
    console.log("All clients are ready");
    allClientsReady = true;
}
```

2. For demonstrating purposes, the test application is always run with one Requesting Client and three Sorting Clients. This is to directly mimic Lab 3's Multithreaded architecture, which used the same number (three) of pthread workers to complete any given sorting job.

iii. **Testing Environment/Metrics:**

1. The network uses a server built with JS and clients built with Emscripten-compiled C++ that can then be run in a browser using a python script that services HTML files. This is to comply with the examples done in class.
2. The metric measured, based on the problem to be solved (sorting numbers), is elapsed time until a given set of numbers is sorted. Algorithms are typically measured in Big O (f(N)) notation (and algorithms are considered "better" regarding strictly time-based performance when they can solve the same problem more efficiently- even if this comes at the cost of extra memory usage), and thus, timing was chosen. The goal of this networked architecture will be to compare the time of one server, one visualizing client, and three processing clients with the time of one main thread and three worker threads on the same set of sorting jobs.

f. **Class Diagram/UML:**



g. **Dependencies:**

   i. *sorting_functions.cpp:* The file containing the C++ functions that will be bound by Embind to callable attributes that can be accessed using Javascript.

   ii. *socket_client.cpp:* The file containing the C++ declaration and definition of the SortingClient class- used to process sort requests.

      1. This file is transpiled into *client.html* by default

   iii. *socket_client_opener.h/.cpp:* The files containing the C++ declaration and definition of the RequestingClient class methods- used to initiate sort requests.

      1. This file is transpiled into *requester.html* by default

   iv. *socket_server.js:* The file containing the JS server that both SortingClient and RequestingClients communicate with while initiating and processing sort requests.

   v. *nickspythonserverscript.py*: The file that services any HTML file in the same directory on a specified port. This is used to run the transpiled RequestingClient and SortingClients.