**Project Overview**

For our project, we made a simple implementation of a distributed shared memory system (DSM) with the use of networks and synchronization. Distributed Shared Memory (DSM) is a method that allows for memory spaces on different computer systems to be addressed as one memory space. The purpose of our DSM system is to give the illusion that multiple computers are using the same memory so that each computer can read and write to a shared data structure.Initially, we believed that our work would involve the utilization of virtual memory, but due to time constraints, we did not end up using this problem solving strategy. Our system integrates a central server that all clients are connected to. This central server is necessary since it enables each client to push the local updated copy to every other client and pull another clients changes to the array for every other client. Directly accessing memory addresses on other systems poses its own challenges and this implementation reduces complexity.

Our evaluation of correctness has two major components we needed to verify. For the first component, each locally updated array must be reliably distributed to all n clients, where n is the number of connected clients. Next, each client must then receive each character sent and update their local copy. It was also essential to ensure all data was not corrupted. For our second component, we had to ensure that the system would work for not just local hosts, but across other computers as well. Another point of evaluation that we used was that we confirmed we could modify the size of the shared array, in which the size is the same and constant for all clients. Our system successfully performed both of these components.

**Design and Implementation**

The two main concerns for our system are hosting data and distributing it to all computers in the system. For hosting the data, we used the simple method of having a global array of characters with a globally defined size that is consistent across all clients. We decided to use this method since it decreases the complexity of testing the distribution process for the push and pull operations. Our distribution can be further broken down to two main structures: server and client. Both of these structures have their own global array, but the corresponding entries entries for each client array remains consistent.

In our implementation, we decided to use a central server (server.c) that runs on one computer and have other computers connect to that central server as clients (client.c) where both utilize standard networking libraries. By using a central server we can easily distribute our data and accept updates since the updates would all be taken by that server as opposed to trying to make multiple clients distribute the data to each other. Our server first waits to connect to a set number of clients by looping until that number of clients has connected. The server stores the client sockets and ports in a global array. Once our server has the required number of clients, it distributes the initial value of the array all clients will have, which is just an array of all lowercase a’s.

Once the server’s copy of the array has been initialized, server.c runs in an infinite while loop. Within this while loop, the server first listens for changes in each of the clients’ local array copies. The server accomplishes this listening tack by utilizing the poll function and the poll\_fd struct from the poll.h library. Poll takes a poll\_fd struct array, the number of fds to listen for (number of clients), and a number of milliseconds to listen before timing out. The poll\_fd struct has three fields: the file descriptor for each client (int), the requested events (we used POLLIN | POLLWRBAND ) (short), and the returned events (short).

When a client attempts to push updates, the poll function changes the revents field in the corresponding array entry from 0 to a short value greater than 0. The server.c function then enters a for loop that identifies which clients have revent values that indicate push requests. This process allows us to determine which client to listen to for changes to the array. The server will fill the global array in the server with the new array information from the client. Next, server.c will loop over all clients and send the updated array. We iteratively continue this process until the user stops running the server or a client disconnects.

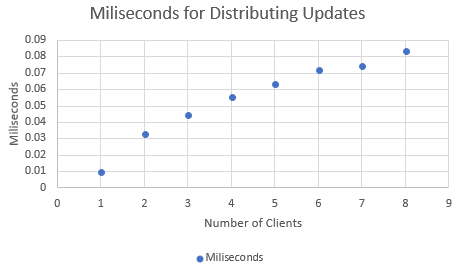
In client.c, we used concurrency by utilizing two seperate threads that run concurrently. The threads both use the pthread.h library. One thread (pull\_thread) listens for pull requests from the server and modifies the local copy with the array sent from the server. The other thread (push\_thread) receives user input through fgets that modifies the local copy and then pushes these changes to the server. Each update to the array completely overwrites the existing contents.

**Evaluation**

We were able to confirm that the server and each client received the correct array values by comparing the array values before they were pushed, when the server received the values, and the values that each client received.

To evaluate the performance of our implementation, we measured how long it takes it to distribute updates to a varying number of clients. This allows us to estimate how well our system scales as the number of clients increases. Measuring the amount of time to distribute updates is an important task for our system, because the point of our system is to give the illusion of shared memory. If the system is too slow, then it will defeat the purpose of our implementation.

We used the time function from the time.h library to measure how many milliseconds it takes to update the array and distribute it to all other clients in the network. Our eight data point were one client, two clients, etc. up to our last run which was eight clients. Each time we sent an array of size ten that consisted of ten lowercase b’s. All clients were connected through localhost so that our measurements would not be confounded by switching between localhost and multiple computers. Another point of consistency was using the first client to connect each time.



From our results we see that there is a direct positive linear correlation between number of clients and amount of time to distribute updates. For each additional client the time increased by roughly 0.01 milliseconds in most cases. This is not an issue for a small number of clients, but if that number grew too large distributing data in a reasonable time may prove difficult. Therefore, depending on how this system will be used, performance optimization may be an important issue to address.