Description of Implementation

This lab required the creation of a read and write application hosted on a TCP server using the C p-threads library to manage concurrent clients. Essentially, a client can connect to the server through a TCP socket, and request to either read from an index in a string array, or write to that index a specified string. This server was implemented four different times:

1.) Mutex on the entire string array

2.) Mutex per node

3.) Read and write lock on the entire string array

4.) Read and write lock per node

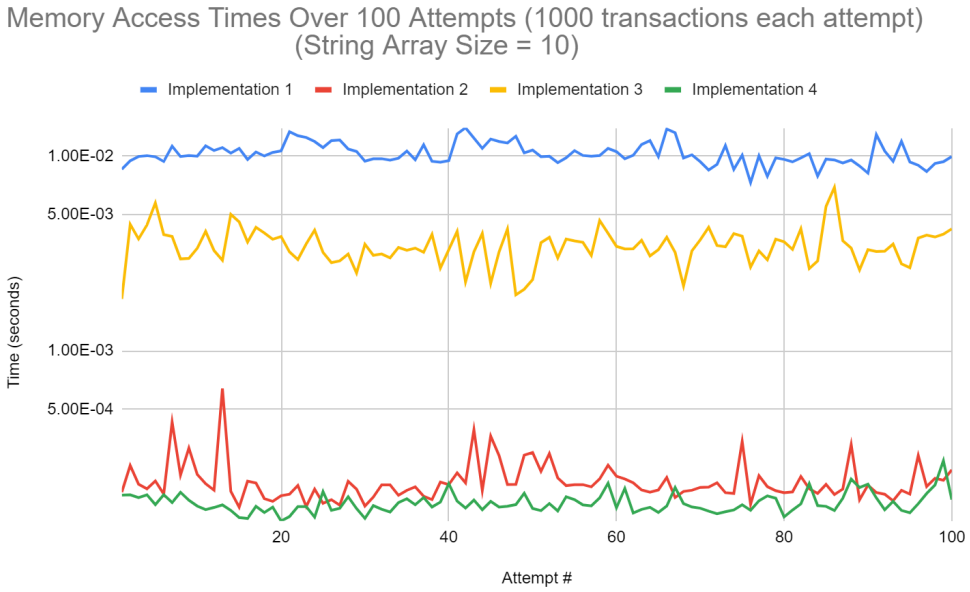
For implementations 2 and 4, we used a global strings array for reading and writing. For implementations 1 and 3, an arguments struct was used to pass a reference to a strings array declared in the main function. The time overhead created from these implementation differences was considered to be negligible, since only memory access time is considered in this report.

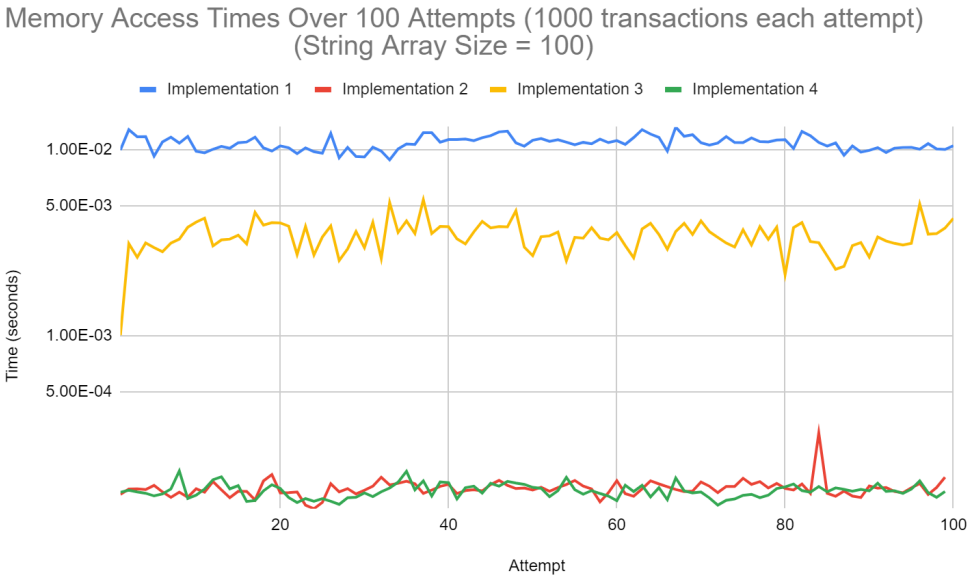
Performance Discussion

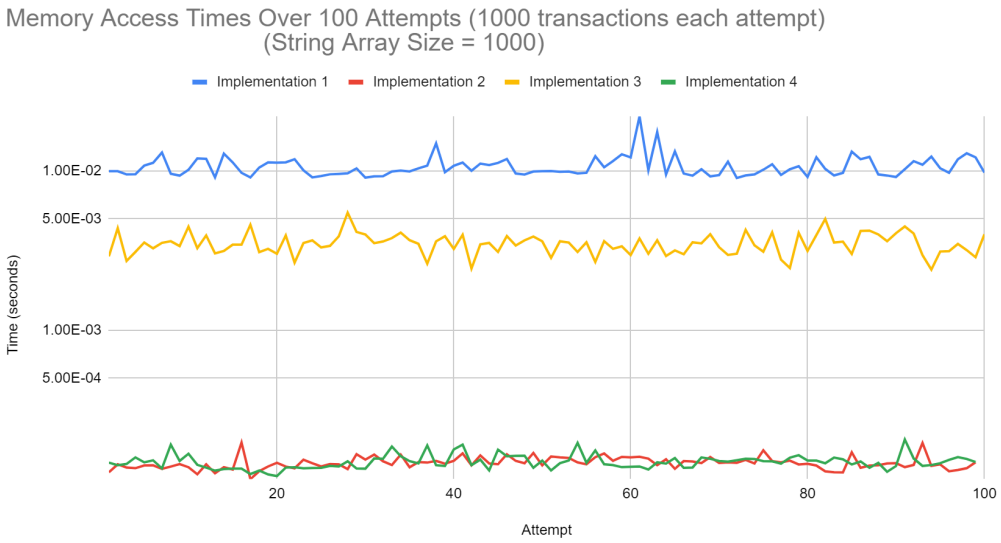
Average time over 100 attempts, each with 1000 transactions. Size of strings array versus implementation

Size Of Strings Array	Implementation 1	Implementation 2	Implementation 3	Implementation 4
10	1.05E-02	2.18E-04	3.49E-03	1.67E-04
100	1.08E-02	1.52E-04	3.44E-03	1.48E-04
1000	1.07E-02	1.48E-04	3.50E-03	1.50E-04

The fastest speed achieved during the measured attempts is 1.17E-04 seconds.


Memory Access Times Over 100 Attempts (1000 transactions each attempt)
(String Array Size = 10)
Implementation 1
Implementation 2
Implementation 3
Implementation 4
1.00E-02
5.00E-03
1.00E-03
5.00E-04
Time (seconds)
20
40
60
80
100
Attempt #


Memory Access Times Over 100 Attempts (1000 transactions each attempt)
(String Array Size = 100)
Implementation 1
Implementation 2
Implementation 3
Implementation 4
1.00E-02
5.00E-03
1.00E-03
5.00E-04
Time (seconds)
20
40
60
80
100
Attempt



This lab examined the implementation of a web socket server which updates an element in a string array based on the client's request. During this lab we explored the performance of four different multithreaded server approaches. The key discrepancies between implementations are the use of read/write lock versus mutex for concurrency, and the usage of these locks on a per element basis or a per array basis. The performance of each approach was measured on a varying array size of 10, 100, and 1000. Also each approach was run 100 times, each run consisting of 1000 clients, each performing one request.

The results of this experiment will be examined in two different manners, which implementation performed best at each array size and, how much did each implementation's performance vary based on the array size.

First of all, Main4, and Main2 were the two best performing programs at each array size with Main4 being considerably better at array size 10. It is worth noting Main3 did outperform Main1 in all cases, as a result of the read/write lock allowing multiple readers to access the critical section at once. Both of the top performing programs used locks on a per item basis, instead of on a per array basis. While having a lock

per element comes with more memory usage in comparison to only having one lock, if large amounts of memory are readily available as in our case, it is clearly the more time efficient solution. As for why the difference between Main2 and Main4 is not as noticeable at large array sizes, this is a result of less conflicting requests. When the array is only of size 10, there will often be threads attempting to read, and write to the same element, hence why the data structure which can allow multiple readers at once will realize better performance. However, as the array size increases, the probability of multiple threads accessing the same element decreases, as does the performance difference. To summarize, using our experiment's parameters it is always better to use concurrency locks on a per element basis, rather than per array. Furthermore, it is better to use read/write locks than mutex locks when the size of the array is small (less than 100), after this the performance benefits are marginal.

Secondly, when looking at the performance of each program per array size the difference was negligible except for Main2, and Main4. In the case of Main1, and Main3, they utilized a lock on a per array basis thus increasing the amount of elements in the array did not affect the amount of threads which could access the critical section. On the other hand, Main2, and Main4 used locks on a per element basis, in this case, increasing the number of elements in the array increases the probability that more threads will be able to access the critical section at once. Both Main2, and Main4 witnessed noticeable improvement as the array increased from size 10 to size 100, however, they did not experience the same improvement from size 100 to size 1000. The diminishing improvement from 100 to 1000 is likely due to the fact that the operating environment does not have enough resources to sustain 1000 threads all acting at the same time. Conclusively, Main2 and Main4 observed noticeable time improvements from array size 10 to 100, and no programs observed noticeable time improvement from size 100 to 1000.