Chase Howard
2021-06-06
University of Washington
CSS 503B

# Program 4: Sockets

**Purpose**

The purpose of this assignment is to gain practice with system calls relating to socket creation and message passing as well as a refresher on multithreaded operations. Additionally, the evaluation of varying write methods on performance using point-to-point communication over a network.

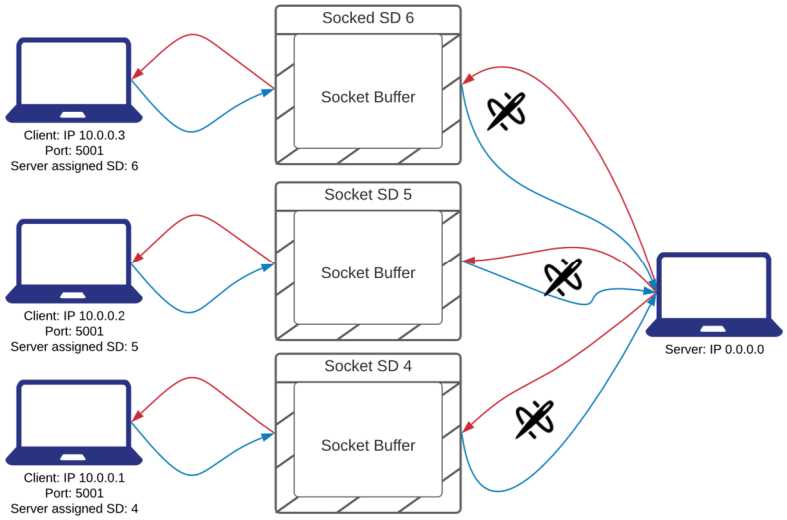
**Program Description**

This program uses a multithreaded client-server model where a client process establishes a connection to a server, sends data or requests, and closes the connection. The server accepts the connection and creates a thread to service each individual request and then wait for another connection on the main thread. Servicing the request consists of (1) reading the number of iterations the client will perform, (2) reading the data sent by the client, and (3) sending the number of reads which the server performed.

Write a Client.cpp and Server.cpp that establishes a TCP connection and sends buffers of data from the client to server. To start, the client sends a message to the server which contains the number of iterations it will perform (each iteration sends 1500 bytes of data). When the server has read the full set of data from the client it will send an acknowledgment message back which includes the number of socket **read()** calls performed.

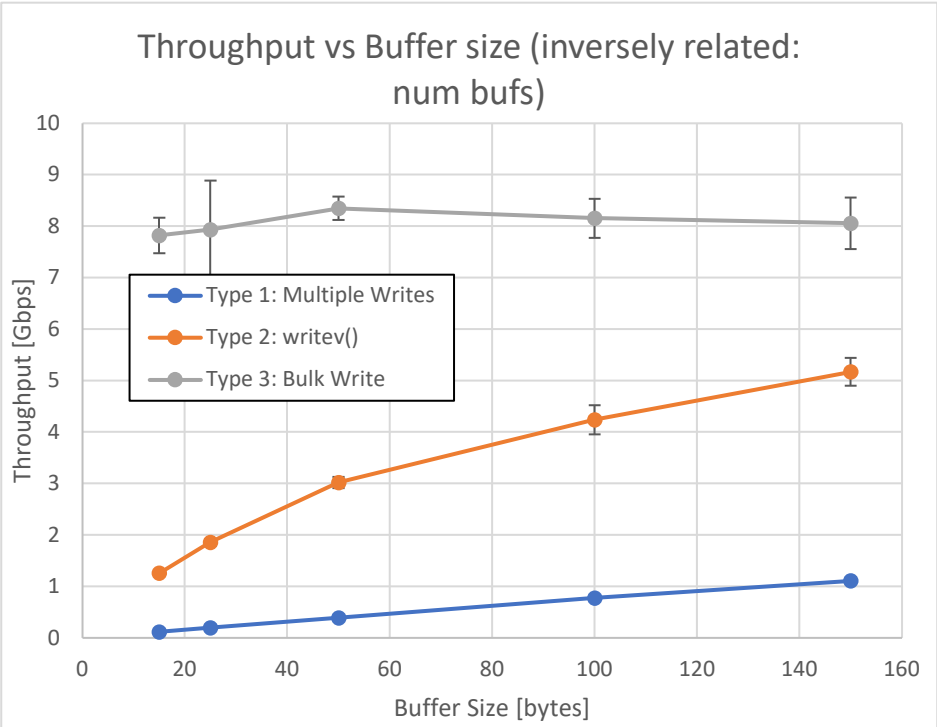
The client will send the data over in three possible ways depending on the type of test being performed (see below for details of the three tests).

- Difference in write types explained in program specifications.

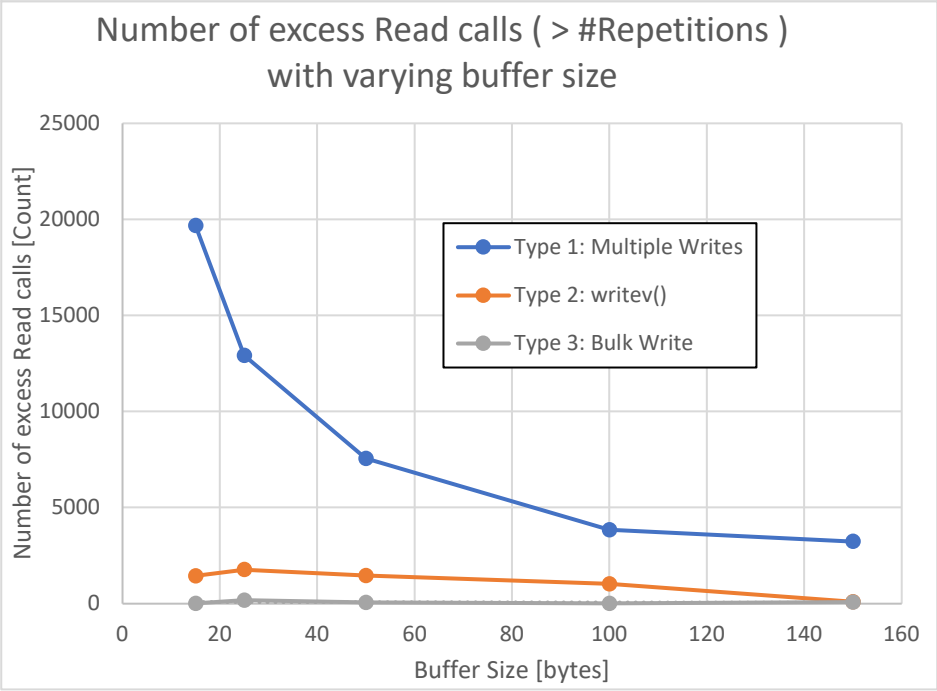


# Discussion

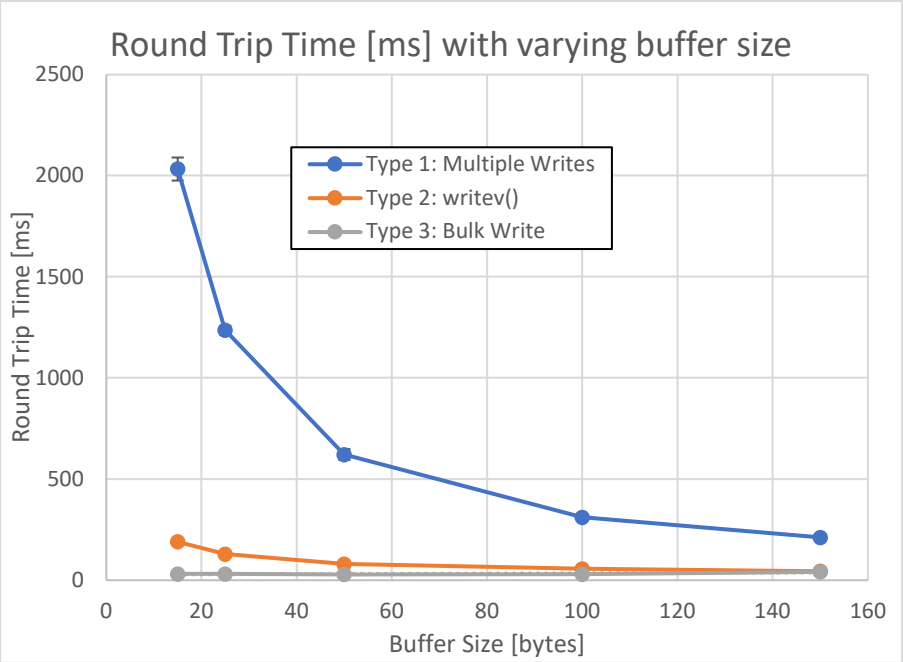
Tests were run with repetition values of 20,000 and 5 runs were recorded for each test case displayed below




| Throughput [Gbps] | | | | |
|---|---|---|---|---|
| | | Type | | |
| *nbufs* | *bufsize* | *1* | *2* | *3* |
| *10* | *150* | *1.11* | *5.17* | *8.056* |
| *15* | *100* | 0.772 | 4.238 | 8.154 |
| *30* | *50* | 0.388 | 3.018 | 8.348 |
| *60* | *25* | 0.194 | 1.858 | 7.932 |
| *100* | *15* | 0.118 | 1.258 | 7.822 |




| Excess No. Reads [Count] (>20,000) | | | | |
|---|---|---|---|---|
| | | Type | | |
| *nbufs* | *bufsize* | *1* | *2* | *3* |
| *10* | *150* | *3232* | *93* | *65* |
| *15* | *100* | 3834 | 1014 | 8 |
| *30* | *50* | 7561 | 1460 | 57 |
| *60* | *25* | 12924 | 1766 | 175 |
| *100* | *15* | 19688 | 1441 | 10 |




| Round Trip Times [ms] | | | | |
|---|---|---|---|---|
| | | Type | | |
| *nbufs* | *bufsize* | *1* | *2* | *3* |
| *10* | *150* | 212 | 45 | 41 |
| *15* | *100* | 311 | 57 | 29 |
| *30* | *50* | 622 | 80 | 29 |
| *60* | *25* | 1236 | 129 | 31 |
| *100* | *15* | 2033 | 191 | 31 |

## Performance observations:

*Type 1:*

- Poor performance regardless of buffer size, especially poor at smaller buffer sizes
- Generally, a linear trend with the worst throughput of the lot
- Additional overhead needed for repeatedly calling write (calls nbuf * repetition write() calls)
- The large number of excess read calls is due to the large number of individual writes taking place on the client side. There are a total of nbuf * repetition write calls made by the client, and while the server is expecting BUFSIZE (1500) bytes, it will read as soon as data is placed in the buffer. With many small read and write calls, there is a significant amount of overhead.

*Type 2:*

- Poor performance with small buffer sizes
- May be best for varying buffer sizes or unknown buffer sizes beforehand.
- Requires use of vector operations to add nbuf's to **iovec** fields prior to calling expensive write calls. This limits the number of write calls made but still requires some additional overhead of vector operations for manipulating the **iovec** structure
- Requires fewer read calls than Type 1, with significantly fewer write calls from the client

*Type 3:*

- Fairly constant throughput around 8 Gbps (Gigabits / second)
- Best performance regardless of buffer size with few excess read calls from server

- For known number of repetitions with constant buffer sizes, this exhibits best performance

*Performance in relation to bandwidth of lab machines.*

- Assumptions: 1 Gbps (Giga**byte** / second) allowable bandwidth on network machines ( in which case Type 3 is effectively limited by network bandwidth) – If network bandwidth is 1 Giga**bits** per second, I don't anyway the throughput can exceed the network bandwidth for a single client. There may be multiple clients who are serviced in individual threads but that would be total throughput when the method used for calculating throughput is for a single client. Alternatively, my calculations may be incorrect.
- Note, there may be some jitter in the use of the timer causing slight inaccuracies in timing measurements
- Type 1 will increase linearly until nbuf = 1 (only a single write call made) in which the throughput will be limited by the network connection.
- Type 2 appears to increase throughput as a function of either log(bufsize) or sqrt(bufsize), asymptotically approaching limit of network capacity as nbuf decreases
- Type 3 will remain constant, only limited by network bandwidth

*Main advantage of using an asynchronous read rather than a blocking read at the server*

- Breaking out handling of clients to a new thread for asynchronous response allows for improved server performance as new clients do not need to wait on a blocking read call from another client who may be trying to read at the same time. This provides enhanced responsiveness from the server to service more clients, more quickly while improving throughput.

## Program Limitations and Improvements

*Limitations*

- Program hangs with sending nbufs = 1500 and bufsize = 1
- Possible optimization of write method selection have a way to select the best transfer method based on application for highest throughput, or fewest number of reads.
- Increase infrastructure capabilities.

*Improvements*

- More object-oriented approach – have client and server inherit a base class with messaging standards – have common headers etc.
- Modularize client and server with functions broken out.
- Make a more robust / extensive TcpSocket class with messaging (send / recv)
- TcpSocket class to allow for customization of flags other than IPv4 etc.