Client Server Benchmark Program

Performance Measurement for Three Implementations of a Client Server Program

---

The authors of this paper and project members are University of Colorado undergraduates in Dr. Pavol Cerny's class CSCI 4830: Concurrent Programming:

Jessica Lynch
Andrew Arnopoulos

OVERVIEW

The goal of our project was to determine which of the following three contrasting methods below provide the fastest running time under various circumstances by testing the performance of messaging between client and server: (1) IPC shared memory client-server program, (2) TCP client-server program, and (3) client-server program using named pipes.

To vary the circumstances under which these implementations were run and tested, we modified the following settings: number of threads (concurrent clients), number of cores, and message load which is determined both by size of message data and frequency of messages sent. We also faced the following concurrency issues: dealing with multiple clients sending and receiving data, ensuring a fair and starvation free solution in terms of both readers and writers, and maintaining data integrity for multiple writers.

As our platform/environment for development and testing, we used a Windows platform 2-core laptop for initial testing, and both an open source Linux (4-core) server and a CSEL (8-core) server for performance measurement (i.e. elra-01.cs.colorado.edu). The programming languages, APIs, standards, etc. used in our project included Java, C, JNI (Java Native Interface), TCP and POSIX.

BASIC TRANSLATOR

In order to test the performance of our programs we needed a unified way to test our programs. We decided that a simple queue that applies a mapping once it's been dequeued was the best solution. For our benchmarking we decided that a simple translator would be a great way to test the performance of each.

There are a lot of intricacies that go into making a robust translator, however making such a translator would create additional runtime overhead and due to the additional overhead it would make benchmarking these programs harder. Our translator takes an English string and returns a Spanish string. It does this by, first, tokenizing the string, using spaces as delineators, and looking up the word in a HashMap. This one-to-one translation provides an average running time of O(n), which is due to the linear time it takes to tokenize the string because a HashMap lookup is performed in constant time.

SHARED MEMORY

For our first client server program implementation, we created IPC shared memory in C, which involves implementation of a shared memory library (written in C) linked to by Java Native Interface (JNI). It uses POSIX shared memory through queueing where memory is shared globally among all processes, and is locked using POSIX Pthread mutex locks. The shared memory program creates a shared memory segment or memory portion which other processes can access if given permission. It is known to be an efficient means of passing messages (data) between programs.

Java has no official API to create a shared memory segment. At times, it is even necessary to

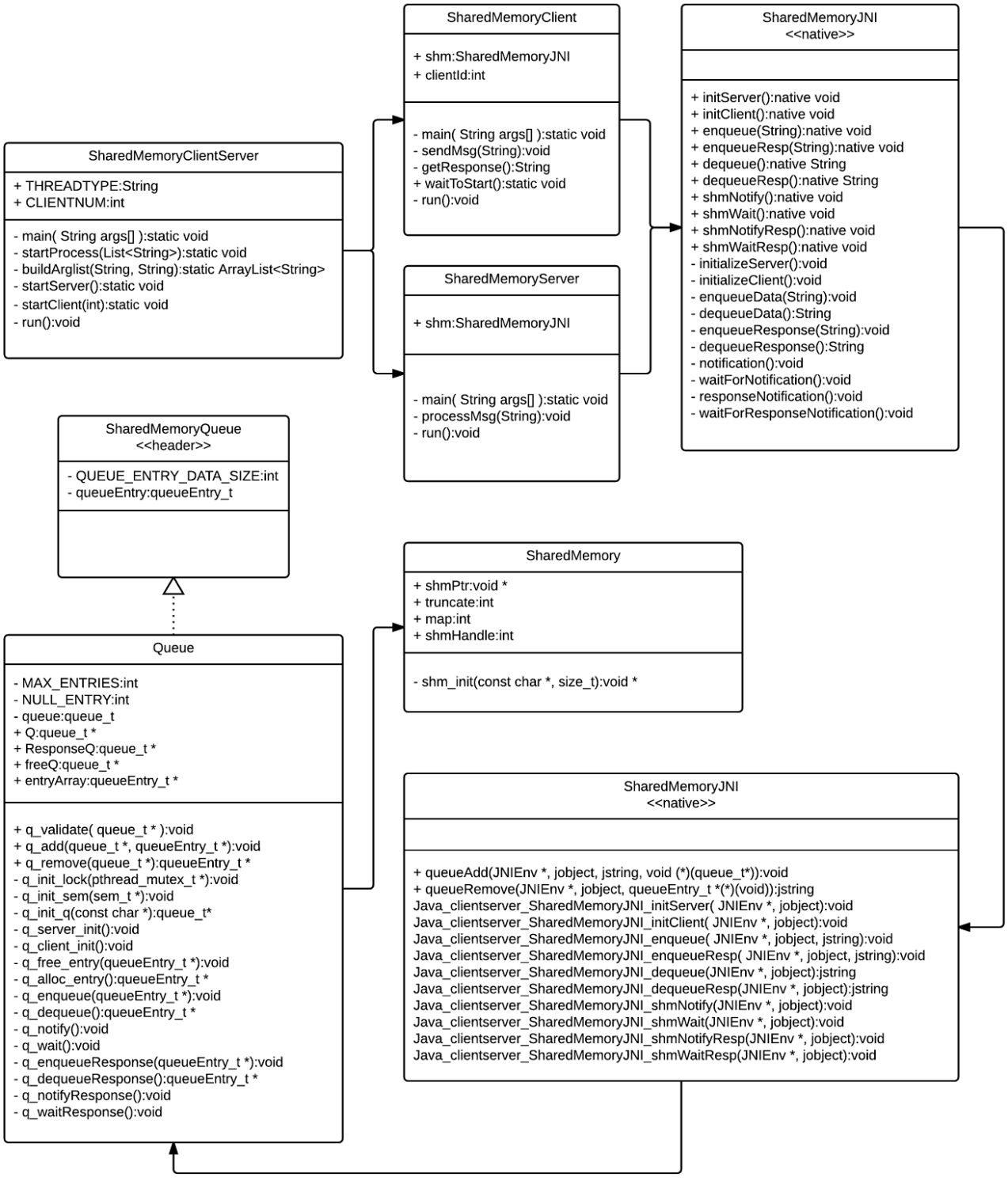


Fig.1.2. Class diagram of IPC shared memory client-server program in C using JNI.

use native codes (C/C++) to overcome the memory management and performance constraints in Java. Java supports native codes via the Java Native Interface (JNI). Therefore, we implemented a helper Shared Memory library in C using JNI to adapt it so that it can be accessed by the Java code for the Java server and multiple client processes to communicate via a shared memory segment.

The most efficient way to implement access to the shared memory is to rely on the mmap() function,

which is true of our implementation. This creates a pointer to the shared memory in the queue that can be used by the Java processes. Signals are used to wake up waiting processes when data is present in the queue. Since write access is allowed for more than one process, our shared memory implementation employs semaphores to prevent inconsistencies and collisions. JNI is known to be rather expensive and complex. However, if used correctly when implementing shared memory for Java processes, it can possibly provide the lowest latency in comparison to other implementations like TCP sockets and named pipes.

The challenges our team overcame while implementing shared memory using JNI were mainly associated with learning curve and system preferences. Understanding how to use JNI took more time than expected, and debugging errors proved complex due to the path of the processes made complicated by the JNI structure. Initially, we attempted to run our program on a Windows 8 platform, but after three days of trying, it proved too difficult and time-consuming. As a result, we moved to an open-source Linux platform and after adjusting the system preferences, i.e. jre version, we were able to run and more easily debug the program until it ran successfully.

One of the main debugging issues we faced was a timing issue where the client threads were starting before the server finished initializing the shared memory queue. Clients started using the shared memory queue while the server was still initializing it. Therefore, we created barrier code to force clients to wait until the shared memory queue was done being initialized. The barrier code requires the server after it is done initializing the queue to create an empty file called "/temp/sminitialized". If the file does not exist, clients repeatedly wait for a period of two milliseconds and check to see if the file exists. Once the file exists, clients can begin work on the queue.

TRANSMISSION CONTROL PROTOCOL (TCP)

For our second client server program implementation, we used Transmission Control Protocol (TCP) sockets for the transport of data and queueing for local shared memory among multiple threads, which uses Pthread mutex locks. TCP is the

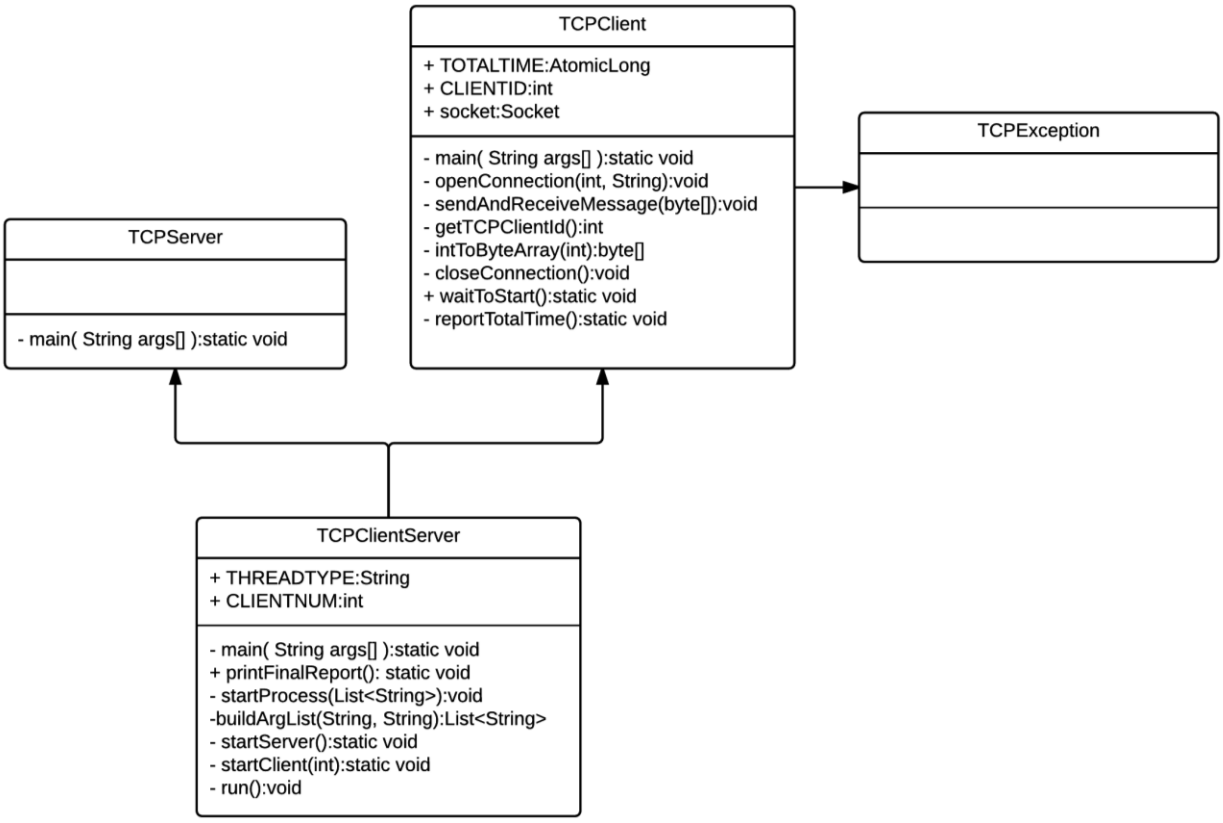


Fig.1.2. Class diagram of TCP client-server program.

most widely used of the main protocols in TCP/IP networks, and unlike UDP, it enables two hosts to establish a connection and exchange streams of data. Our implementation establishes a single server with multiple clients who send messages and receive responses to and from the server. Opening a connection via a TCP socket enables a message exchange between server and client. The socket is one end-point of a two way connection link. It must also be closed after the message exchange is complete.

NAMED PIPES

For our third client server program implementation, we used named pipes in Java to provide communication between the client and the server, which uses standard Java locks. Named pipes allow for cross application communication. Our implementation of pipes is done using an input and an output file (/tmp/input and /tmp/output, respectively) with buffered stream wrappers. This comes at a cost due to the slow read and write times of a standard hard drive. This had to be done due to the lack of named pipes in Java and the persistence of named pipes, after the programs has finished executing.

In our implementation when the client is initialized it creates a unique identifier and writes the unique identifier and its message to a global input file (/tmp/input). This global input file is read by the server and placed into the message queue. The message waits until it is dequeued by a message receiver. When the message is dequeued a mapping is applied to the message and the message is sent back to the client with the client's unique identifier attached. The client receives the message by continuously reading a global output file (/tmp/input), until it reads it's unique identifier.

DATA ANALYSIS

We ran each implementation of the client server program with the following adjustments made to two different settings: client thread count and message load or number of messages sent per client process. We adjusted each setting to values within a power of two. The number of client threads were adjusted to be 16, 128 and 256. For each of these thread count settings, we ran our program with different message load values, which were also within a power of two. For each of the client thread count settings, we experimented with 8, 64, and 128 messages sent per client process. Each message consisted of the sending client thread's id to simplify benchmarking. We, therefore, experimented with nine different combinations of settings.

Each of our programs return the average runtime, so we ran each combination five times, removed the lowest and highest times, and averaged the averages to find the best average runtime for the combination per implementation. For each of the combinations, the data for each implementation was recorded and calculated to show performance per process and per message sent. The data shows the performance for one-way messaging for the sake of simplicity.

Shared Memory

After running the shared memory client server benchmark program five times per settings combination, we analyzed the data and found that it performs best with 16 client threads and 8 messages sent per client, overall. Under this combination, the average runtime per process is 1.126 ms and average runtime per message sent is 0.141 ms. The best average time per message sent, however, was achieved by combining 16 client threads and 128 messages sent per client. The average runtime per message sent under this combination is 0.0949 ms. Each of the graphs below show the performance for one-way messaging for the sake of simplicity.

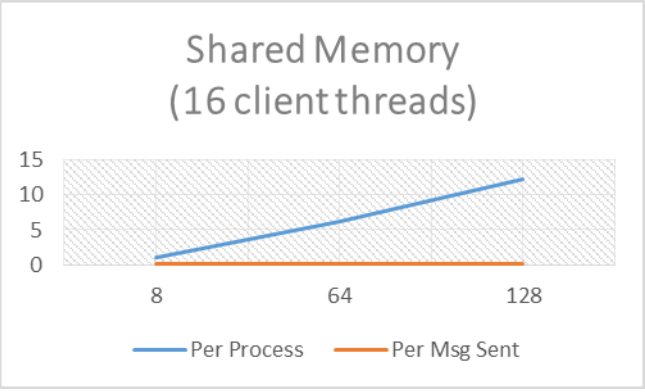


Fig.1.3. The above graph shows for 16 client threads the time per process and per msg sent in milliseconds (y-axis) each for 8, 64 and 128 messages sent per process (x-axis). The raw data supporting our graphical representations of performance for each settings combination are included below.

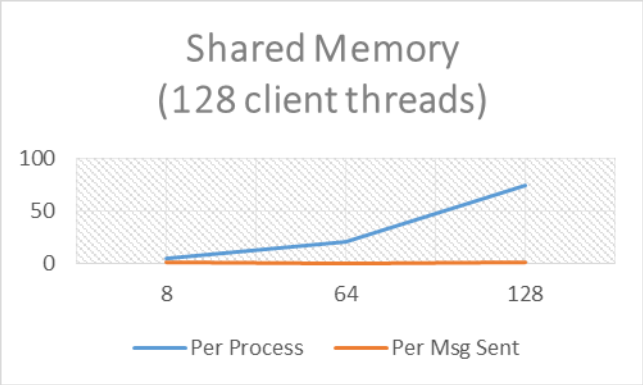


Fig.1.4. The above graph shows for 128 client threads the time per process and per msg sent in milliseconds (y-axis) each for 8, 64 and 128 messages sent per process (x-axis).

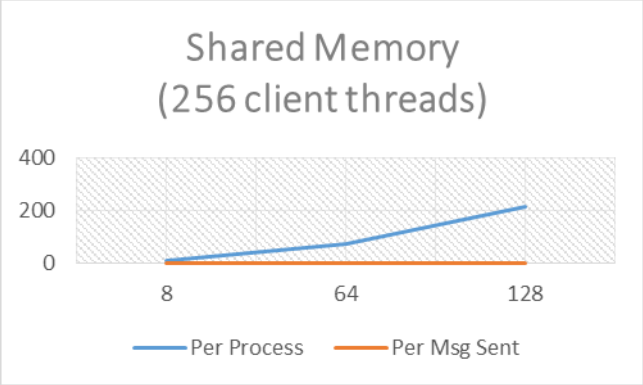


Fig.1.5. The above graphs shows for 256 client threads the time per process and per msg sent in milliseconds (y-axis) each for 8, 64 and 128 messages sent per process (x-axis).

Raw data for shared memory with 16 client threads		
Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	1.126	0.141
64	6.101	0.0953
128	12.15	0.095

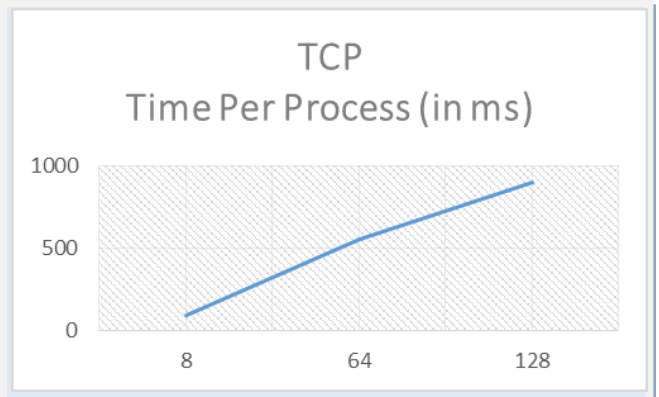
Raw data for shared memory with 128 client threads		
Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	4.777	0.597
64	20.717	0.324
128	74.45	0.582

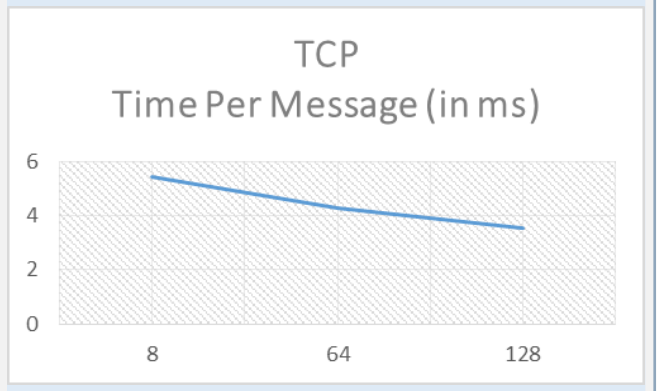
Raw data for shared memory with 256 client threads		
Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	11.74	1.468
64	73.522	1.149
128	211.537	1.653

Fig.1.5. The raw data supporting our graphical representations of performance for each settings combination are included in the above tables.

TCP

Our TCP client server benchmark program reached a threshold at 128 client threads and 8 messages sent. Beyond these settings, the program would run for a small amount of time and then it would deadlock. From our debugging efforts and investigation, we found that the server is waiting for the client to process the response it just sent, while the clients are waiting for the response which they do not see. Analyzing the logs we put in place, however,





Raw data for TCP with 16 client threads		
Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	87.048	5.441
64	547.864	4.28
128	899.81	3.515

Fig.1.5. The above graphs show for 16 client threads the time per process and per msg sent in milliseconds (y-axis) each for 8, 64 and 128 messages sent per process (x-axis). The data shows the performance for one-way messaging for the sake of simplicity. The raw data supporting our graphical representations of performance for each settings combination are included (bottom).

shows that the server sent the response, and that the client checking for the data is just not seeing it. We

were surprised to find out that the Java TCP is not guarding against this. We could not find the problem in our code, but this does not rule out the possibility that it could be rooted in our implementation of TCP.

We analyzed the data that we were able to procure and found that our TCP benchmark program performs best per process with 16 client threads and 8 messages. Under this combination, the average runtime per process under this combination is 87.048 ms and average runtime per message sent is 5.441 ms. The best average time per message sent, however, was achieved by combining 16 client threads and 128 messages sent per client. The average runtime per message sent under this combination is 3.515 ms. The performance of TCP, however, is surpassed by that of shared memory.

Named Pipes

shows that the server sent the response, and that the client checking for the data is just not seeing it. We were surprised to find out that the Java TCP is not guarding against this. We could not find the problem in our code, but this does not rule out the possibility that it could be rooted in our implementation of TCP.

We analyzed the data that we were able to procure and found that our TCP benchmark program performs best per process with 16 client threads and 8 messages. Under this combination, the average runtime per process under this combination is 87.048 ms and average runtime per message sent is 5.441 ms. The best average time per message sent, however, was achieved by combining 16 client threads and 128 messages sent per client. The average runtime per message sent under this combination is 3.515 ms. The performance of TCP, however, is surpassed by that of shared memory.

Fig.1.6. The above graphs shows for each client thread amount (16, 128 and 256) the time per process and per msg sent in milliseconds (y-axis) each for 8, 64 and 128 messages sent per process (x-axis). The data shows the performance for one-way messaging for the sake of simplicity.

Raw data for TCP with 16 client threads

Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	137.05	8.57
64	1402.09	10.95
128	4374.62	17.09

Raw data for TCP with 128 client threads

Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	1605.61	100.35
64	11476.39	89.66
128	28611.93	111.77

Raw data for TCP with 256 client threads

Number of Messages Sent Per Client	Avg Time Per Client Thread (in ms.)	Avg Time Per Msg Sent (in ms.)
8	87.048	5.441
64	547.864	4.28
128	899.81	3.515

Fig.1.4. The raw data supporting our graphical representations of performance for each settings combination are included in the above tables.

CONCLUSION

Our data shows that shared memory using JNI out performs TCP and named pipes under the context of a client server program. We anticipated shared memory and named pipes, however, to be a lot more comparable since they both make use of in-memory file systems. We were pleased to see that the overhead from JNI minimally affected the performance of the shared memory implementation. For future design choices client similar client server programs, we will gravitate to the use of shared memory over other options. Named pipes is worth taking a second look at, however. Further investigations into the design of our named pipes implementation is needed and into eliminating the overhead from multiple reads and writes.

REFERENCES

1. Java Programming Tutorial (JNI). https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html.
2. Beginning JNI with NetBeans IDE and C/C++ Plugin on Linux. https://netbeans.org/kb/docs/cnd/beginning-jni-linux.html.
3. *Using Memory Mapped Files and JNI to communicate between Java and C++ programs.* Stanley Wang. http://www.codeproject.com/Articles/2460/Using-Memory-Mapped-Files-and-JNI-to-communicate-b.
4. *Using TCP sockets in Java*. M Bateman, A Ruddle & C Allison. Created as part of the TCP View project. http://tcp.cs.st-andrews.ac.uk/lectures/tcp_injava.pdf.
5. Shared memory educational website. http://www.cs.cf.ac.uk/Dave/C/node27.html.
6. UML Diagram Tool. https://www.lucidchart.com.
7. Java™ Platform, Standard Edition 7 API Specification. http://docs.oracle.com/javase/7/docs/api/index.html?overview-summary.html.