# Abstract

Team 3 Members:

Nancy Carrier N00017868

Joseph Allen N01045721

Jasmine Jones N01379886

Joshua Martinez N01276494

Robert Futterman N00665333

Athlene Jones N00168025

Computer Networks & Distributed Systems

Spring 2018

Comparison of Distributed Programming Paradigms

Distributed computing enables applications to take advantage of multiple hosts on internetworks. The project compared latency for light load and heavy load requests using two distributed programming paradigms. All tests were conducted on the same hardware enabling the group to observe and compare latency. The first version is an iterative server, and the second is a concurrent server. Concurrency was implemented by spawning a thread for each client request. Both versions implement a simple TCP-based network client-server administration tool. Both versions offer a simple menu through which the user may request the host provide system time, uptime, memory use, netstat, current users, and running processes. The number of clients were increased from 1-100 in varying increments.

# 1. Introduction

Client-server computing (Figure 1) is a two-tiered architecture for distributing an application across hosts on a network. Distributed applications enable resources across a network or internetwork to be shared more effectively and efficiently. Java is a popular and capable language for development of distributed applications. Sockets are the endpoints for communication between processes on client and server using TCP.

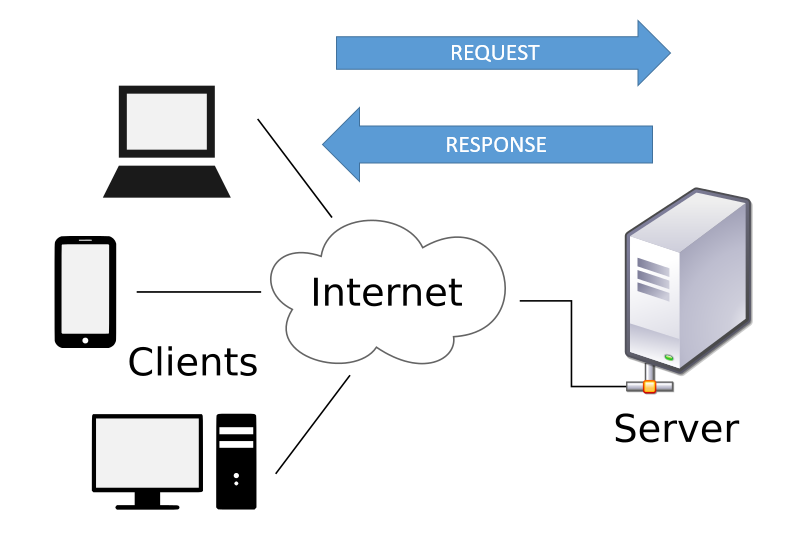


Figure 1 - Client Server Architecture

The project compared network response time for two server types: 1) Iterative model and 2) Concurrent model. The iterative method demonstrates one server process handling multiple client requests. A single iterative process serves the client requests in a queue as received. The concurrent method enables the server to spawn a new thread for each client request. This enables each thread to take advantage of CPU time not being used by another thread.

Both server models responded to multiple simultaneous client requests. The client program for both paradigms was a network administration tool. This program would be started on one of two machines with same configuration. User selected from six different functions.

1. Host Current Date and Time
2. Host Uptime
3. Host Memory Use
4. Host Netstat
5. Host Current Users
6. Host Running Processes

The current date/time function was used for light load testing. Netstat was used for heavy load testing. These functions can be executed for increasing number of clients using both versions of the server. The server runs the command and returns the output to the client.

The performance of light load request vs. heavy load request were captured, charted and analyzed. Results for all client requests were averaged. Multiple runs were used to remove impact of variability among runs.

# 2.0 Distributed Application Development

## Distributed Computing with Sockets

All programs for the project were developed in Java using Berkeley sockets. The diagram below provides a generalized view of socket functions. Berkeley sockets is a standard used across the internet. An internet socket has a local socket address containing the IP address and port number. In our project we used TCP sockets.

Generally, the client program creates a socket at its end and tries to connect to a server-socket on the server. The server accepts the connection and returns the socket address, establishing communication with the client.

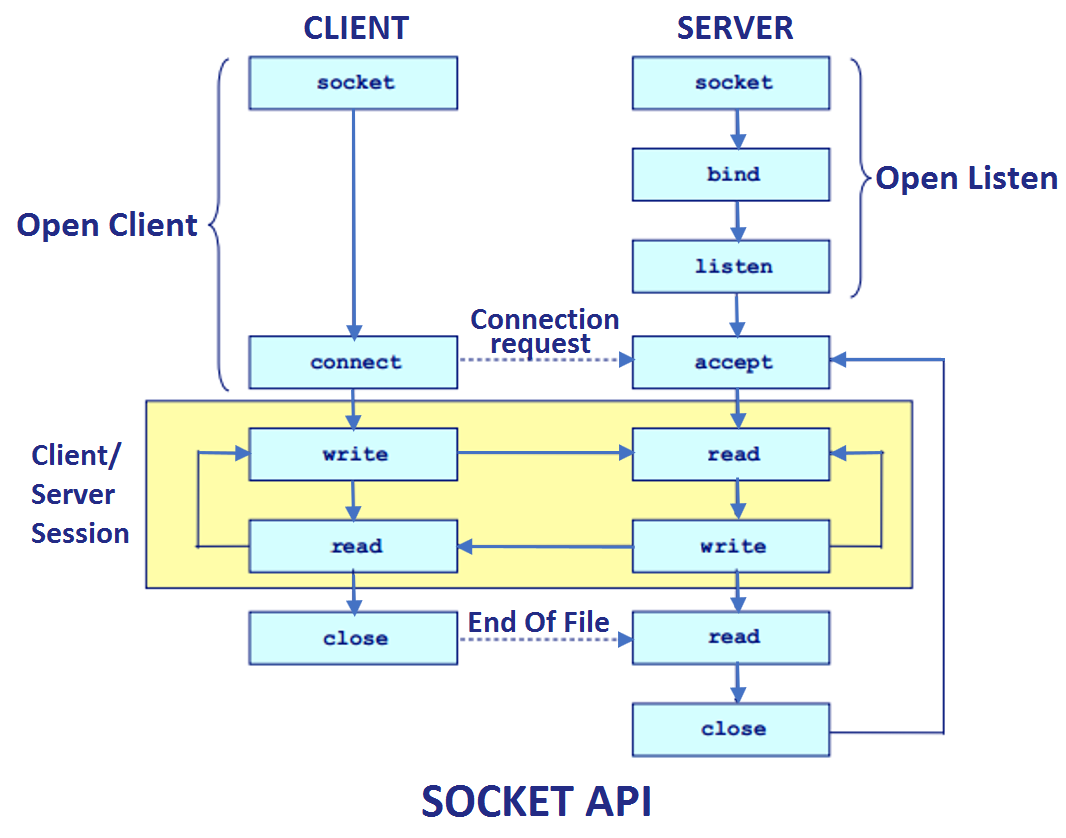


Figure 2 - Overview of Berkeley Sockets [6]

## 2.1 Iterative Approach

Iterative approach is a paradigm of programming which involves the breakdown of development of huge software to lesser manageable portions. The procedure involves marking out of statement blocks in computer programs for a specific count of repetitions.

The servers have a capability of communicating with various clients while a client can only establish communication with a single server at a certain period (Livingstone, 2018). The clients are required to understand the server address; however, the server does not have to know the client address before connecting.

Referenced from our ManyClients and IterativeServer.java code, iterative servers services a single client before establishing connection in the next port (e.g., the server has to display a menu before prompting the user for a command). However, there can be a buildup of queues.

Different clients can contact the server by use of server’s recognized protocol port

(if (args.length != 1){

System.err.println("Usage: java Server <port number>");

System.exit(1);

}

int portNumber = Integer.parseInt(args[0]);

try{

ServerSocket server = new ServerSocket(portNumber, 100);

System.out.println("(Type CTRL+C to end the server program)");

while(true){

try (

Socket client = server.accept(); )

The Transmission Control Protocol offers connection-oriented services, because it is founded on connection between the server and the client (Graba, 2007). When its client has sent data to the servers, it need to get acknowledgements accordingly. In case there is no acknowledgement, TCP retransmits the data automatically and delays for some time.

With our code, the structure of the socket address is at all times passed as argument by reference to any socket function. However, socket functions which take one pointer as argument deals with the socket address structure from any of the protocol families which are supported.

ServerSocket server = new ServerSocket(portNumber, 100);

System.out.println("(Type CTRL+C to end the server program)");

while(true){

try (

Socket client = server.accept();

The API to protocols such as the TCP/IP software is in the Operating System (OS). Thus, for the use of TCP/IP for communication, application programs require interactions with the OS (Luus, 2000).

The Transmission Control Protocol Socket Application Programming Interface function calls sequence for the server and clients in Transmission Control Protocol connections is as follows.

* Establishing TCP sockets on client’s side are
  + Creating of a socket by use of socket () function
  + Connecting it to the server address by use of connect() function
  + Sending and receiving data through read() and write() functions
* The server side involves
  + Creating a socket by use of socket() function
  + Binding the socket to an address by use of bind() function
  + Listening for connections by use of listen() function
  + Accepting connections by use of accept() functions system call. It blocks up to the time a client establishes connection with the server.
  + Sending and receiving data with send() and receive() functions

The socket() Function: Defining the form of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP) is the first step to calling the socket function.

The connect() function is utilized by TCP clients for establishing connections with TCP servers.

The bind() allocates to the socket a local protocol address.

The listen() function translate sockets which are not connected to passive sockets, showing that the kernel is supposed to accept incoming requests of connection requests in direct to the socket.

The send() Function: has similarity to the write() function, however, this lets some options specification. The extra argument flags are utilized in the specifying the way we need the data transmission, and returns the number of bytes after success, -1 on error.

The receive() Function: has similarity to read(), however, it allows specification of some options for controlling the way the data is received, returning in bytes the messages length, 0 when there are no messages and peers had performed an logical shutdown, or -1 on error.

The close() Function: The standard close() function closes a socket and terminates a TCP socket. Returning 0 after success, -1 on error. This process occurs repeatedly.

## 2.2 Concurrent Approach

Java threads allow servers to handle many clients at the same time by using the host resources more efficiently. "Using threads, a single application can work on several tasks concurrently, as if multiple copies of the Java Virtual Machine were running. (In reality, a single copy of the JVM is shared or multiplexed among the different threads.)"[2]

**Concurrent Approach**

To overcome the limitations of the iterative server, a more complex approach, the concurrent approach is a paradigm in programming that is designed for handling multiple clients at the same time. A separate thread is created to handle each new client session, allowing the server to deal with multiple clients simultaneously, making optimal use of available resources, all while the server continues to listen for additional clients. (JGuru, 2012) Concurrency is implemented by creating a Runnable interface that spawns a thread for each client request. Using a simple TCP-based network client-server administration tool, our program displays a simple menu requesting the number of clients to run. Once obtained, the client’s request is then executed.

ServerSocket server = new ServerSocket(portNumber, 100);

         System.out.println("(Type CTRL+C to end the server program)");

         // Server loop

         while(true) new ServerThread(server.accept());

Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a client thread that processes the request. That way, the thread listening for the next incoming requests spends as much time as possible in the server.accept() call thereby minimizing the risk for clients to be denied access to the server. Here, the client thread actually executes the request. (Net-informations.com)

The accept() method is used to get a connection request on the listening socket. If a connection is accepted the accept() method supplies a new connection socket (clientSocket) on which to perform the connection activity. If the server produces a thread after a successful accept(), passing the new clientSocket to it, then the original server thread can continue to loop waiting for a new client request on listenSocket. Meanwhile the spawned thread can carry out the connection request on the connection socket (clientSocket). (Cems.uwe.ac.uk)

The system must be able to identify each of the server processes so that incoming and outgoing messages reach their correct endpoints. Therefore, the transport layer of our layered system must be able to identify the correct process from the header information supplied with each packet of information (the source port and the destination port number). The port number together with the IP address of the client forms a unique pair to identify the thread. (Cems.uwe.ac.uk)

**Establishing TCP sockets on client’s side are:**

* Creating of a socket by use of socket() function; Defining the form of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP) is the first step to calling the socket function.
* The action of connecting a client socket to a server is handled at the time the client socket is created. The client socket is created with the server's IP address as a parameter, so connecting is handled there instead of with a separate connect() function.
* Sending and receiving data through read() and write() functions.

**Server-side calls involve:**

* Accepting connections by use of accept() functions system call. It blocks up to the time a client establishes connection with the server. The listening and binding is handled here in the server's accept() method and not by separate listen() and bind() functions.
* Creating a socket by use of socket() function; Defining the form of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP) is the first step to calling the socket function.
* kernel is supposed to accept incoming requests of connection requests in direct to the socket.
* The connect() function is utilized by TCP clients for establishing connections with TCP servers.
* The send() Function: has similarity to the write() function, however, this  lets some options specification. The extra argument flags are utilized in the specifying the way we need the data transmission, and returns the number of bytes after success, -1 on error.
* The receive() Function: has similarity to read(), however, it allows specification of some options for controlling the way the data is received, returning in bytes the messages length, 0 when there are no messages and peers had performed an logical shutdown, or -1 on error.                                                                                                                              The close() Function: The standard close() function closes a socket and terminates a TCP socket. Returning 0 after success, -1 on error. This process occurs repeatedly.

# 3.0 Results and Comparisons

## 3.1 Test Bed

The client and server programs were run on two UNF hosts.

1. IP 192.168.100.105
2. IP 192.168.100.106

The host and network configuration is below.

* Dell Optiplex 755
* Operating System: Ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-24-generic i686)
* Intel 82566DM-2 Gigabit Network Connection
* Netgear gigabit switch
* Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz
* 4 GB RAM
* Seagate Barracuda 7200.10 ST3160815AS 160GB 7200 RPM 8MB Cache SATA 3.0Gb/s 3.5" Hard Drive Bare Drive

Software Used

* Java Compiler
* Editor JGRASP as Editor
* File transfer (scp, win scp)
* SSH, Putty

## 3.2 Studies Carried Out

To evaluate latency, mean server response time was collected for both iterative and concurrent server programs. Time was measured in milliseconds. A programmatic timer measured from initiation of first client request to completion for all client requests. The times were averaged for up to 6 runs to help reduce impact of variability due to time/day of execution and other system workload.

Two levels of work were measured: light load and heavy load. A request for current system date and time was used for light load test. A request for netstats was used for heavy load testing. Client processes were increased as follows 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

## Graphs

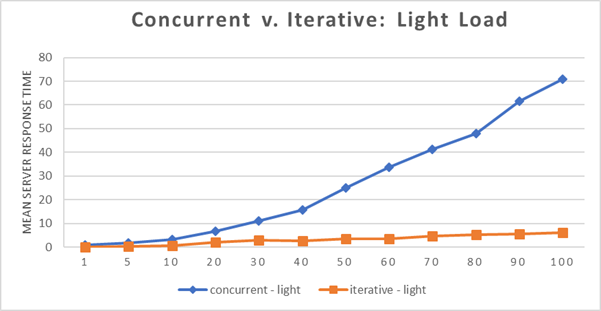


Figure 3 - Latency is better for light load using iterative model.

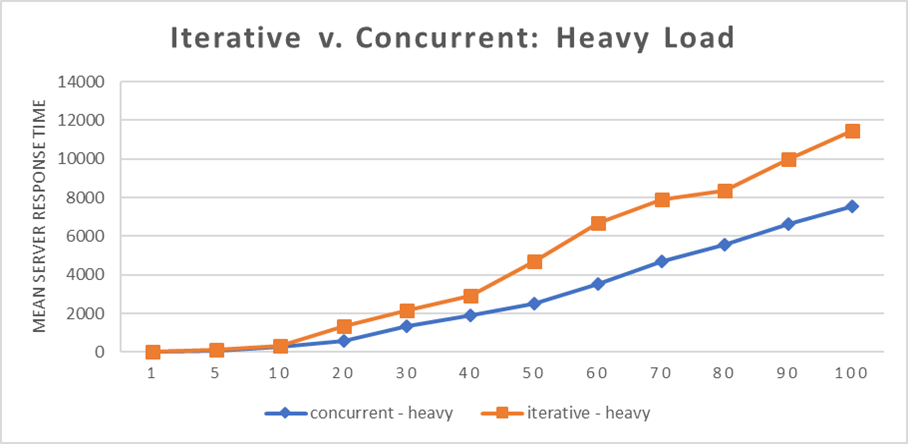


Figure 4 - Latency is better using concurrent model under heavy load.

## 3.3 Results

For light load, concurrent (multi-threaded) model was less efficient than iterative. This is likely due to operating system overhead of managing the multiple threads (switching active thread to CPU).

For heavy load, concurrent model was more efficient.

# 4.0 Conclusions

Our project demonstrated that iterative processing performs better than concurrent model for very light loads. Iterative servers handle clients sequentially, finishing with one client before servicing the next. When a process needs more than a very small amount of server connection time, the wait for subsequent clients may exceed requirements for the application.

For heavier loads, the concurrent model using thread-per-client was the better performer.

Some advantages of threads are [7]

* Threads minimize the context switching time.
* Use of threads provides concurrency within a process.
* Efficient communication.
* It is more economical to create and context switch threads.
* Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

The light load performs less successfully in the thread model because every new thread consumes system resources: spawning a thread takes CPU cycles and each thread has its own data structures (e.g., stacks) that consume system memory. [7] As the number of threads increases, more and more system resources are consumed by thread overhead. Eventually, the system is spending more time dealing with context switching and thread management than with servicing connections.

A recommendation to overcome this is to reuse threads from a thread “pool” that is created on start-up. Instead of spawning a new thread for each connection, the server creates a thread pool on start-up by spawning a fixed number of threads. When a new client connection arrives at the server, it is assigned to a thread from the pool. When the thread finishes with the client, it returns to the pool, ready to handle another request.

In related studies, Berkeley socket communication was shown to be a better performer than the more feature rich, abstracted options of Java RMI or CORBA. CORBA appears to no longer be used, although legacy applications exist. Both related studies reviewed showed performance degradation with CORBA. Java RMI added less latency than CORBA, but provides additional useful server features.

Java, Python and Scala: all can be used to implement socket-based distributed communications. A challenge going forward will be whether socket programming can or should be abstracted to enable higher level development and consistency across languages.

In summary, the trade-off between portability, ease and accuracy of coding, functionality and performance continue to be decisions best made per implementation requirements. However, few situations will require only light load communication; thus, it is our recommendation that multiple threads will perform better in most situations. Our research and testing both support this conclusion. Re-use of threads rather than spawning for each client will improve the light load performance.

# REFERENCES

[1] Ahuja, Sanjay, P., Quintao, R., and Basavanahally, S.,*An Empirical Analysis of the Performance of CORBA and Java RMI: Distributed Open Object Architectures for Internet Based Applications,*Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001), Orlando, FL, July 2001.

[2] Calvert, Kenneth L., and Michael J. Donahoo. *TCP/IP Sockets in Java: Practical Guide for Programmers*. Morgan Kaufmann/Elsevier, 2008.

[3] Graba, J. (2007). An introduction to network programming with Java. London: Springer.

[4] Livingstone, M. (2018). Java Network Programming - TCP/IP Socket Programming. Udemy. Retrieved 12 April 2018, from <https://www.udemy.com/java-network-programming/>

[5] Luus, R. (2000). Iterative dynamic programming. Boca Raton: Chapman & Hall/CRC.

[6] Pattamsetti, Raja Malleswara Rao. *Distributed Computing in Java 9,* PACKT Publishing Limited 2017, Chapter: Sockets and streams.

[7] Tutorials Point. “Java Multithreading.” *Www.tutorialspoint.com*, Tutorials Point, 24 Mar. 2018, [www.tutorialspoint.com/java/java\_multithreading.htm](http://www.tutorialspoint.com/java/java_multithreading.htm).

[8] Tutorials Point. “Operating System - Multi-Threading.” *Www.tutorialspoint.com*, Tutorials Point, 8 Jan. 2018, [www.tutorialspoint.com/operating\_system/os\_multi\_threading.htm](http://www.tutorialspoint.com/operating_system/os_multi_threading.htm).

[9] Vlassov, Vladimir, and Leif Lindback. “Network and Web Basics. Architectures of Distributed Applications. Java Platforms Editions .” *Lecture Notes from ID2212 Network Programming with Java* , [www.kth.se/social/files/56373503f276540e96aa0137/lec1.pdf](http://www.kth.se/social/files/56373503f276540e96aa0137/lec1.pdf).

Lecture notes from Swedish university found online that are nice summary of distributed computing, sockets and Java.

[10] “What Is a Socket?” *What Is a Socket? (The Java™ Tutorials > Custom Networking > All About Sockets)*, Oracle, 19 July 2016, docs.oracle.com/javase/tutorial/networking/sockets/definition.html.

Additional Link References

http://www.cems.uwe.ac.uk/~a2-lenz/n-gunton/worksheets/npJavaWs3.pdf

http://net-informations.com/java/net/multithreaded.htm

http://www.jguru.com/faq/view.jsp?EID=42986

Related Work References

[1] Ahuja, Sanjay, P., Quintao, R., and Basavanahally, S.,*An Empirical Analysis of the Performance of CORBA and Java RMI: Distributed Open Object Architectures for Internet Based Applications,*Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001), Orlando, FL, July 2001.

[2] Eggen, M. and Eggen, R. Efficiency of Distributed Parallel Processing using Jva RMI, Sockets and CORBA. In Proceedings of 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (June 2001).