Interprocess Communication with Java in a Microsoft Windows Environment - Heading Towards a Generic IPC Design

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Dylan Gregory Smith

Grahamstown, South Africa
28 October 2016

Abstract

Your abstract goes here .....

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2013) [16]:
D.3.3 [Language Constructs and Features]: Frameworks
I.2.9 [Robotics]: Autonomous vehicles

General-Terms: Autonomous Robot, Framework

Acknowledgements

Firstly, I would like to thank my mother, Jennifer Smith, for her unrelenting support, kindness and guidance; not only in my Honours year, but for the entire duration of my time spent at Rhodes University.

I would also like to thank my uncle, David Preston, without whom I would never have been able to attend this prestigious institution. I am privileged and proud to be an alumnus of Rhodes University's Department of Computer Science.

Thank you to my supervisor, Prof. George Wells, for the immensely insightful counsel and guidance throughout the duration of this research.

I would also like to express my utmost gratitude to Nicole Upfold for her undying support and belief in me during the tough parts of this project. Her motivation and her own outstanding work in her scientific field inspired me to conduct my research with much methodical enthusiasm.

I acknowledge and say thank you to my fellow Honours students for a great year. It was an absolute pleasure to share a laboratory with you.

In addition, I would like to acknowledge the financial and technical support of Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107) through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Java is a widely used general-purpose programming language that supports an object-oriented programming model as well as features such as multithreading, portability and simplicity (Oracle, 2010). Java currently lacks support for interprocess communication (IPC) but instead relies on a distributed network programming mechanism (Wells, 2010). This means that Java uses socket communication to communicate with other Java processes (i.e. Java programs with their own distinct address spaces). This approach can be significantly inefficient, as it forces communication to traverse the layers of the protocol stack (Wells, 2009).

The lack of IPC features is problematic due to the ubiquity of modern parallel, multicore computing systems. Most machines no longer rely on a uniprocessor model (Hayes, 2007). The ability to perform IPC as well as process synchronisation is fundamental to the design of effective parallel and distributed systems.

Java, however, does provide a mechanism in which it can access native code. This is known as the Java Native Interface (JNI). This framework allows Java programs to access the application programming interface (API) or system calls of the operating system that it is executing on. This is typically done in the native code of C or C++ (Liang, 1999). This means that low-level OS features (memory, I/O, IPC mechanisms and so forth) can be accessed using the JNI framework by making use of native C/C++ code (Dawson *et al.*, 2009). As such, this framework can be used to develop an alternative to Java socket IPC due to it allowing programmers to access low-level OS features.

The Linux and Solaris IPC implementations developed by Wells (2009) make use of the

JNI framework to implement IPC for the Java programming language, but no such implementation exists for Microsoft Windows or any other OS. The original implementation showed promising results and an attempt to replicate its performance boost on a Microsoft Windows platform served as a large portion of this research. As a result, this required significant research into the internal Windows IPC structures, including communication and synchronisation mechanisms as well as various implementation methodologies. WindowsIPC is the name I have given to the class library I have designed.

The final goal is to describe any potential generics that could serve as an implementation guide to a design that is operating system independent, for example a single design that could be executed on Microsoft-based OSes, various Linux flavours as well as Apple's Mac OS X.

1.1 Research Objectives

This research aims to achieve the following objectives:

1. Understand Microsoft Windows' IPC mechanisms and present how they can be implemented to provide a communication and synchronisation features for Java processes executing in their own distinct Java Virtual Machines (JVMs).

2. Understand the Java Native Interface and present how this can be interfaced as efficiently as possible with Windows' APIs.

3. Design and implement a viable Java library using JNI that will allow Java programmers to write applications that are able to communicate without the use of socket programming. This essentially will provide the language with concurrency mechanisms. In addition, the library will need to demonstrate that it is a more efficient and effective alternative to Java Sockets. As such statistical proof needs to be demonstrated.

4. Since JNI makes use of native code, this presents a significant portability issue. As such recommendations need to be made in terms of a generic IPC design for the Java platform. This is essentially an analysis of generic IPC mechanisms that can be implemented on Linux-based operating systems, Microsoft Windows, MacOS X and so forth.

1.2 Research Approach

The first phase involved the investigation and understanding of Windows internals and how the operating system implements IPC. This involved examining the Windows API, or more informally WINAPI. It was found that the APIs were immensely complicated in comparison to that of the Unix world and as such, much time was spent on gaining an understanding on how each mechanism could be implemented. Their viability were assessed as well as performance. Concurrent to this, I examined the literature available relevant to multiprocessing in Java as well as any other alternatives that may be possible.

Next, an understanding of how JNI worked was conducted. This involved the creation of simple test programs that made use of native code to perform simple computations such as string and array manipulation. This involved the revision of my C coding ability as well as understanding the Linux implementation of Java IPC. This cemented my understanding of how JNI and native C code interfaces with Java Virtual Machines.

I then initiated the development process once I felt comfortable working with JNI. I assessed each WINAPI and ranked them in order of difficulty in terms of implementation. I started with the lowest rank and proceeded with development. This involved the testing process which ensured that messages were sending correctly between Java programs. I used a test-driven development approach, using junit as a unit testing environment in simple Java programs that used the WindowsIPC class library to ensure that the expected results were obtained.

I then proceeded with performance assessment. Java sockets was used as a benchmark to see if WindowsIPC could produce better message sending times. This involved running each mechanism 10 times and calculating the mean execution time for a specific message size (40, 400, 4 000 and 40 000 bytes). Simple graphical representations of this data was produced to illustrate performance gains.

I then proceeded to investigate the viability of OS independent solutions, since the use of native code invalidates Java's property of portability.

1.3 Document Outline

THIS IS COMPLETED ONCE REST OF DOCUMENT HAS BEEN DONE

Chapter 2

Literature Review

2.1 Introduction

The purpose of this chapter is to outline methods with which Java concurrency can be implemented in a Microsoft Windows OS environment. The Linux version will be discussed later in this chapter. In addition, it examines literature that is available that may aid the in the development of an enhancement to the Java programming language that will allow access to IPC features on a Windows OS. This chapter will also examine how the JNI can be used as an intermediary between Java and native C code to access low-level OS IPC features. It also investigates the internal IPC mechanisms of Windows (WINAPI). In addition, it will provide a review of IPC mechanisms of other OSes to aid in the design of a generic Java class library to provide the language with IPC functionality, regardless of platform or operating system.

Note: The term "concurrency" within this text refers to both communication and synchronisation among processes and/or threads that belong to a particular program. When referring to Oracle and Microsoft's API, I simply cite the top-level link to avoid a large number of references to the respective APIs.

It is necessary to include some general background information regarding processes, threads and accepted concurrency mechanisms prior to the discussion of the techniques currently provided by the Java.

2.2 Processes and Interprocess Communication

A process is an abstraction that is created upon program execution - it is effectively an instance of a program that is currently running within its own distinct address space (Tanenbaum & Bos, 2015). All processes consist of three sections, namely code, data and a stack. The code section contains executable code (generally machine instructions) that the program is currently dealing with. The data section contains global variables and data that may be dynamically allocated in the heap as the program progresses through its execution stages. The stack contains any local variables that are in use. Programs that consist of a single process are known as *sequential* whereas programs that consist of multiple processes (and possibly threads) are known as *concurrent*. The underlying operating system (OS) is responsible for managing program processes and their subsequent threads (Garg, 2005). Processes may also consist of implicit variables, for example the program counter as well as the contents of data registers. The state of the process may change as it executes statements (Magee & Kramer, 2006).

Different OSs generally create new processes and implement IPC in different ways, but some form of commonalities are present, such as process identifiers and synchronisation techniques. Unix-based OSs generally make use of the system call `fork()` to create child processes. Then additional system calls, such as `shmget()`, can be used to access the native IPC mechanisms provided by the OS. In addition, Unix provides a comprehensive threading mechanism known as POSIX Threads which aid in multithreading programming (Bovet & Cesati, 2003). An OS such as Microsoft Windows uses a function called `CreateProc()` which handles process creation as well as features such as the process status, security and data structure access (Tanenbaum & Bos, 2015).

Parallel programming includes the concept of IPC. All programs that deal with multiple processes may require a method with which to communicate to achieve the desired results. IPC can then be defined as the method with which processes (either belonging to a single program or multiple programs) pass messages to one another - i.e. process to process. This is an important concept that allows the flow of information between programs running on multicore machines (Tanenbaum & Bos, 2015). IPC can introduce significant problems in terms of synchronisation, including race conditions and deadlock.

The Java programming model currently does not provide comprehensive support for an efficient IPC mechanism which appears to be a significant design flaw. This is particularly apparent due to the fact that multicore processing and the presence of parallel programming is almost ubiquitous in modern software engineering. In addition, Java is a popular

language that is used on many industry development jobs and this coupled with multicore processing's ubiquity adds to the problem.

2.3 Process Synchronisation

Correct synchronisation is a concept that is fundamental to concurrent program execution. It is an inherent property of IPC and is often blanketed under this term. Various languages implement synchronisation in different ways. This text will present a generic discussion of synchronisation and then proceed to mechanisms provided by Java, particularly within a multithreading environment.

When designing programs that access data concurrently, it is important to design them in such a way that data is consistently synchronised. This will avoid data corruption if multiple processes or threads are trying to access data that is shared (Garg, 2005). Race conditions are anomalies that can arise when multiple processes are attempting to read and write data that is shared among them. This is a significant problem if the final result of a process depends on this data, but it was previously modified by some other arbitrary process - effectively data has been corrupted and is now not relevant to the current process (Tanenbaum & Bos, 2015). Data or a piece of code that is shared among processes, and more specifically threads, is called the *critical region* or *critical section*. These can include aspects such as shared resources, memory locations and other related data. In order to avoid race conditions, some form of mutual exclusion needs to be implemented in order to protect data from rogue processes that may change it in the critical region and hence alter the results of another process. It should lock the data when the program is inside of it and then release it for any other process that requires it at that point in time (Beveridge & Wiener, 1997). Various software-implemented synchronisation techniques exist in addition to hardware-dependent techniques.

2.3.1 Hardware Synchronisation

In terms of hardware, synchronisation may be achieved by disabling interrupts on the CPU as soon as the process enters a critical region. This means that no process can switch into the OS's kernel mode (i.e. context switch) and alter the region's values. When the process exits the critical region, interrupts are then re-enabled (Garg, 2005). This is not a feasible solution because additional problems arise. In addition, this is not possible in multicore

CPU designs, as one would have to disable multiple CPU cores. Another problem that may arise by giving the program the ability to disable interrupts is that there is no way of telling if the program will re-enable interrupts at a later stage which poses significant problems to system functionality (Tanenbaum & Bos, 2015).

2.3.2 Software Synchronisation

Software mutual exclusion can be implemented relatively easily with *busy-waiting* techniques. A simple implementation makes use of lock variable that initially has a value of 0. A process subsequently tests this value to determine if a lock is in place, protecting a critical region. If the lock value is 0, a process will enter the region and set the lock to 1. An additional process can test this lock variable to determine the region's status. If the value is set, said process will wait until the region is available. This technique still has a race condition on the lock value. This can be prevented by making use of *strict alternation*. This is known as a spin lock where an integer value is used to determine which process can enter the critical region. Process A will enter a critical region whilst process B continuously monitors the integer `turn` value (Tanenbaum & Bos, 2015). This creates a number of significant problems as more overhead is introduced as the number of competing processes increases. In addition, other processes can be delayed by a slower process that is using the lock (Wisniewski *et al.*, 1994).

To overcome this, it is better to put competing processes to sleep to preserve CPU time. This can be illustrated by means of the Producer-Consumer problem. If a shared memory buffer exists (which can be abstracted as an array), a producer will deposit information and a producer will extract information from the buffer. Problems can arise if the consumer attempts to extract an item from an empty buffer. Conversely, a producer cannot deposit an item if the buffer is full. This can be circumvented by putting the producer and consumer to sleep until these conditions have been satisfied. As a result, `wakeup` and `sleep` are called when necessary. Problems can arise if a `wakeup` is made to a process that is not currently sleeping as this call will be lost.

As such, a better approach to synchronisation is to make use of semaphores, which was pioneered by Dijkstra (2002). Semaphores introduce the concept of an integer `s` that stores a non-negative value. Once `s` has been given a value, only two atomic operations may be performed on them: `up(s)` and `down(s)`. `up` effectively means "to release" the process whereas `down` means allow the process "to pass" (Magee & Kramer, 2006).

Semaphores need to ensure that the checking and subsequent changing of `s` is an atomic operation, therefore it is necessary to introduce a lock until the operation has been completed. It is also necessary in a multicore system (Tanenbaum & Bos, 2015). Semaphores are not without disadvantages including the fact that they are a low-level concept that can be prone to bugs and can be difficult to implement. If the programmer does not keep track of calls to `up` and `down`, deadlock can occur. In addition, semaphore usage can become more complicated as the complexity of accompanying algorithms increases (Schauble, 2003).

2.4 Java Concurrency Mechanisms

The Java programming language, at present, provides a significantly comprehensive support structure for multithreading programming by means of the `java.util.concurrent` package and `Thread` class. Version 5.0 of the Java platform introduced high-level APIs for concurrency in the language (Oracle, 2016). As such, the language provides multithreading programming facilities through means of `java.lang.Thread`, methods declared as `synchronised` which utilise monitor concepts such as wait, notify and signal. These are described substantially in Oracle's Java API documentation, particularly within `java.util.concurrent`. In terms of IPC, Java only supports RMI which is classified as distributed computing, however it can be used locally using localhost (127.0.0.1) (Wells, 2009).

2.4.1 java.lang.Thread

Java provides `java.lang.Thread` that allows the explicit creation of thread objects (Garg, 2005). There are two alternative ways in which this can be utilised. Firstly, classes can be written that inherit from the `Thread` class, which allow the programmer to override the provided `run` method. Subsequently `start` can be invoked to execute the thread (Garg, 2005). The following code adapted from Oracle (2016) (the Java API) illustrates thread creation:

```
class MyThread extends Thread {
   // this method overrides the run method in Thread
   public void run() {
      // do something
   }
   public static void main (String[] args) {
        MyThread t = new MyThread(); // create thread object
        t.start(); // launch thread
   }
}
```

The alternative method with which to make use of Java's threading mechanism is to implement the `Runnable` interface. This class can then be used to implement the `run` method. An instance of this class is then passed into the constructor of the `Thread` class and then started by calling `start` directly, as illustrated below:

```
MyClass c = new MyClass();
new Thread(c).start();
```

Java threads have their local variables organised as a stack, and can access shared variables and are generally used as lightweight processes running within the context of a single JVM (Magee & Kramer, 2006). Java threads support priority and have a minimum and maximum priority value and contain get and set methods. This can heuristically affect OS schedulers (Lea, 1999). According to Hyde (1999), making use of Java threads can have some benefits as well as drawbacks. Benefits can include better interaction with the user, the simulation of simultaneous activities, use of multiple processors as well as performing other tasks whilst waiting for slow IO operations. Drawbacks exist, such as each `Thread` instantiation, which introduces overhead and use of memory resources. They also require processor resources such as starting, scheduling, stopping and killing of `Thread` objects. According to Oracle (2016), Java concurrency mostly concerns itself with threads as opposed to multiple processes. In addition, most instances of the JVM run as a single process with associated child threads as units of execution. This is a distinct concept from that of IPC between Java processes belonging to separate JVMs.

2.4.2 java.util.concurrent

The package `java.util.concurrent` (which is affiliated with `Thread`) provides concurrency classes that include synchronisation mechanisms that contain operations that set and inspect thread state, various mutual exclusion solutions as well as barriers and queues and so forth (Lea, 2005). These facilities are sufficient for developing good multithreaded applications which aim to eliminate problems that can arise such as deadlock, race conditions and unintentional thread interaction (Wells & Anderson, 2013).

Oracle (2016) provides tools such as concurrent data structures like `ConcurrentLinkedQueue` which contains methods such as `peek`, `poll` and so forth which allows threads to access concurrent data. The `Executor` interface is a framework that essentially allows the creation of custom thread-like systems and for defining lightweight task frameworks.

In terms of synchronisation, the package offers `Semaphore`, `CountDownLatch`, `CyclicBarrier`, `Phaser` and `Exchanger`.

2.4.2.1 Semaphore

The `Semaphore` class in Java is implemented in the form of a counting semaphore which keeps track of a set of permits. The method `acquire` tries to gain access to a resource but blocks if a permit is unavailable. Conversely the method `release` adds a permit to a resource. Permits are kept tracked of in the form of a simple integer counter. The constructor accepts a fairness parameter which ensures that no thread starvation occurs if a thread tries to acquire a resource but is continually blocked (Oracle, 2016).

2.4.2.2 CountDownLatch

This essentially introduces a lock until another thread completes its operations. The object is given a count and the method `await` blocks until the count reaches zero. `countDown()` decrements the count on each invocation. Subsequently threads are released. This class can be used as an on or off latch by giving it a count of one (Oracle, 2016).

2.4.2.3 CyclicBarrier

This class implements barrier functionality that allows for groups of threads to catch up to a certain point before proceeding and is cyclic in nature since the barrier can be reused once the waiting threads are released (Oracle, 2016).

2.4.2.4 Phaser

This barrier technique is very similar to `CyclicBarrier` but allows a flexible addition of threads to be added at any time (unlike `CyclicBarrier`). A simple method called `register()` can be invoked to add a task. A phase number is initially generated and increases to a maximum number previously defined once all tasks reach the phaser number. It is wrapped around once the count reaches the maximum number. Phasers support a concept known as *Tiering* that allows subphasers to be created, which can reduce the amount of contention if there are a large number of tasks Oracle (2016).

2.4.2.5 Exchanger

This defines a point at which threads can pair together and exchange elements. On entry, threads pair and exchange their partner's object. This is performed within the `exchange()` method (Oracle, 2016).

The mechanisms provided by `java.util.concurrent` ultimately require a shared address space for JVM threads to synchronise and have common access to shared objects (Wells, 2010).

2.4.3 IPC

Current Java IPC mechanisms appear to make use of distributed programming features such as the "loopback" mechanism, remote method invocation (RMI) and the Java Message Service (JMS).

2.4.3.1 Loopback Mechanism

According to Wells (2009), Java provides a robust mechanism for communication in terms of distributed computing using the Internet Protocol's (IP) loopback system using 127.0.0.1. This means that communication can take place between multiple processes on a single machine. This takes the form of socket programming using TCP/UDP by using Java's standard socket API (Taboada *et al.*, 2013). Messages can be encapsulated into packets and passed into the IP stack as necessary. Messages can then passed to programs running in separate JVMs using this loopback mechanism, thereby providing a form of IPC. Taboada *et al.* (2013) states that Java's network support over TCP/UDP tends to be an ineffient communication mechanism. In addition, research conducted by Wells (2010) also found it to be inefficient. This was due to messages having to traverse the protocol stack.

2.4.3.2 Remote Method Invocation

Remote Method Invocation (RMI) allows methods to be called in an object that are running within the context of another JVM. Arguments of the called method are packetised and sent over a network to a JVM. They are then passed into the remote method as necessary. The object to which the remote method belongs should provide safety mechanisms as the caller does not know about the callee's state (Goetz *et al.*, 2006). RMI can be used in conjunction with the loopback method discussed above. A similar mechanism known as Common Object Request Broker Architecture (CORBA) also involves the remote calling of methods to provide a distributed solution (Wells, 2010). Yet another form of distributed communication is the Java Message Service (JMS) which provides communication between applications (Oracle, 2016). RMI provides a solution to IPC but according to Taboada *et al.* (2013), at a cost of poor performance.

It should be emphasised that the current facilities in Java are inherently thread-based as opposed to process-based. The current network communication mechanism through the IP stack has significant overhead.

2.5 Java Native Interface

The Java Native Interface (JNI) is a framework, released in 1997 in JDK 1.1, that allows the integration of native code into standard Java applications. Currently JNI supports

C and C++ within its framework. It can be used to integrate legacy code with Java applications as well as allow the language to invoke native code and the other way around. JNI can invoke native methods that are written as functions in a native language, such as C or C++. These native functions reside in native libraries (i.e. on a Windows platform as .dll files). JNI can also embed a JVM into applications written in a native language. There are two significant implications of using JNI. Firstly, since native code is being used, it breaks Java's property of "write once, run anywhere". This means that code written will be host dependent. Therefore a recompile would be necessary if this code was to be ported to another host environment. Secondly, Java is type safe and protects the programmer whereas C and C++ does not (Liang, 1999). JNI allows access to low-level machine features such as memory, IO and so forth, including IPC facilities (Dawson *et al.*, 2009). Liang (1999) states that prior to using JNI, other mechanisms should be considered such as other IPC mechanisms, legacy databases and other Java distributed object technologies.

2.5.1 How It Works

Methods are declared as `native` in Java. This means that their implementation is then written in a native language such as C/C++. Once this code is compiled and fed into `javah`, a C header file is generated with the native method's function prototype. This function can then be written in C/C++ and compiled using any standard C compiler (e.g. gcc, cc, cl, Borland C++ and so on). In a Windows environment, a .dll shared library file is created and linked into the Java program that is calling the native code. The function generated by `javah` may resemble the following:

```
JNIEXPORT jstring JNICALL Java_Sample1_stringMethod
  (JNIEnv *, jobject, jstring);
```

`JNIEXPORT` and `JNICALL` are macros that ensure that the function is exported from `jni.h` which is a native library provided by the Java platform. `jstring` means that the function is typed and returns a Java string. The parameter `JNIEnv *` points to a pointer that in turn points to a function table (an array-like structure). The function table contains pointers that point to native functions that have been written. The image below is taken from Liang (1999).

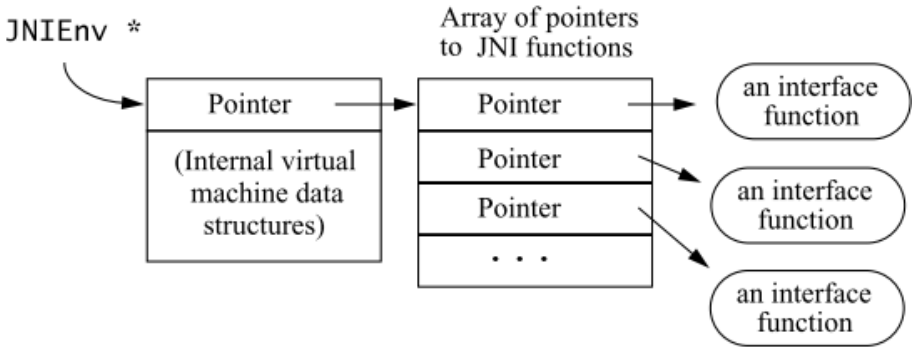

Figure 2.1: JNI function table

`jobject` refers to the object that the method belongs. If it is a static method, it is a reference to the class that it belongs to. The `jstring` parameter is an argument given to the function in the Java code. In this case, it was a string passed into the method. There can be any arbitrary number of arguments passed into the function and will be reflected here when the C/C++ function types are generated (Liang, 1999).

2.5.2 Performance

According to Dawson *et al.* (2009) and Wells (2010), use of the JNI can incur performance penalties. Performance is somewhat restricted when calls to native code are made or when a JVM has to move up a class hierarchy. This is true if a native function attempts to call a method written in Java.

2.6 Linux and Solaris IPC Implementation

2.6.1 Initial Implementation

A Linux and Solaris-based implementation was created by Wells (2010) using Unix System V calls. The approach taken was to develop Java classes using JNI to access these system calls. The Unix IPC features used included message queues, semaphores, shared memory and piping. The implementation aimed to make the native Java methods as close to possible as the system calls. This includes names and parameter lists.

2.6.1.1 Problems Encountered

Problems encountered by Wells (2010) was the difference in data representation between Java and C. Data representation issues arrised in aspects such as error handling (i.e. exceptions vs -1 returns), control system calls, semaphore operations and shared memory.

2.6.1.2 Results

Simple communication was peformed and and compared against the network loopback connection. It was found that all the Unix System V IPC mechanisms were significantly faster than using the loopback method. Named pipes were found to be fastest since it made minimal use of JNI calls. This was the case for tests running on Ubuntu 8.05.1 and Solaris.

2.6.2 Shared Memory Object Framework Model (SMOF)

The intial implementation by Wells (2010) contained some portability issues since calls to native code were being made. A Shared Memory Object Framework (SMOF) was developed by Wells & Anderson (2013) that aimed to create a more abstract solution to Java's IPC problems; thereby simplifying its porting to an OS such as Windows.

The SMOF provides the programmer with classes that are used within shared memory. The objects of these classes act as an intermediary between data and the shared memory. Mutual exclusion is provided by means of Java monitor concepts. The primary class `SharedObject` is a parent class from which child classes are built. Two processes can access the shared object by using a key, which is essentially a file name that acts a point of reference (Wells & Anderson, 2013).

2.6.2.1 Results

Testing involved assessing the latency of sending simple messages between processes. The loopback approach and named pipes were used as benchmarks. It was found to be 80% faster than named pipes and 87% faster than the loopback network (Wells & Anderson, 2013).

2.7 Windows IPC and Concurrency Mechanisms

Modern Microsoft Windows Operating Systems appear to be highly modular in nature, with emphasis on object-oriented design which is illustrated by the following diagram taken from Russinovich & Solomon (2009).

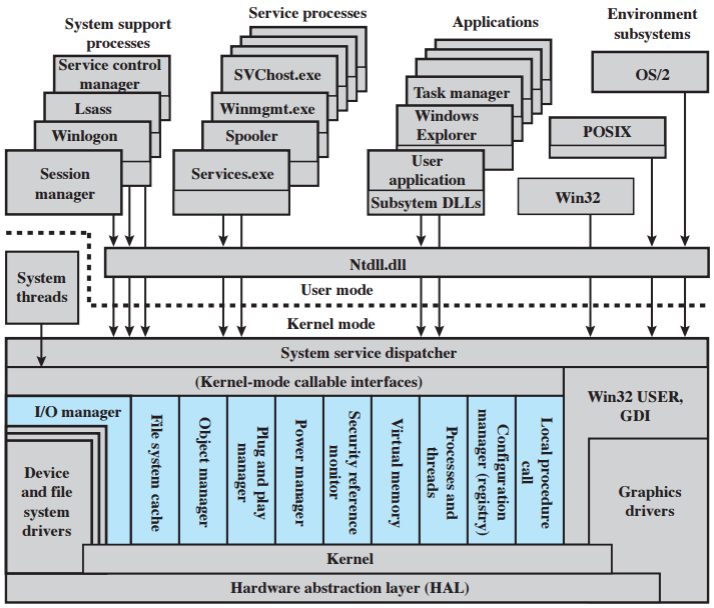

Figure 2.2: Windows System Architecture

According to Stallings (2009), the Windows kernel mode is divided into the following blocks: executive, kernel, the hardware abstraction layer, device drivers and the windowing and graphics system as depicted in figure 2.2 above.

2.7.1 Windows Executive

The Executive block contains the base services such as memory, thread management and IPC. The kernel block handles process switching and thread scheduling. It provides an application programming interface (API) for user mode software (Stallings, 2009). Within the Executive block, several other modules are built in. The module specific to IPC is the Process/thread manager. This module is responsible for creating, managing, and deletion of processes and associated threads. The Executive also contains a module called Local Procedure Call (LPC) that allows local processes to communicate. It is similar to remote procedure calls (RPC) (Stallings, 2009). The executive contains functions that are callable from user-mode through the Windows API (Russinovich *et al.*, 2012).

2.7.2 The Windows Process

Windows processes are defined as objects that have a number of attributes and functions (shown in figure 2.3) and contain at least one thread. These threads may execute in parallel on multicore systems. (Stallings, 2009).

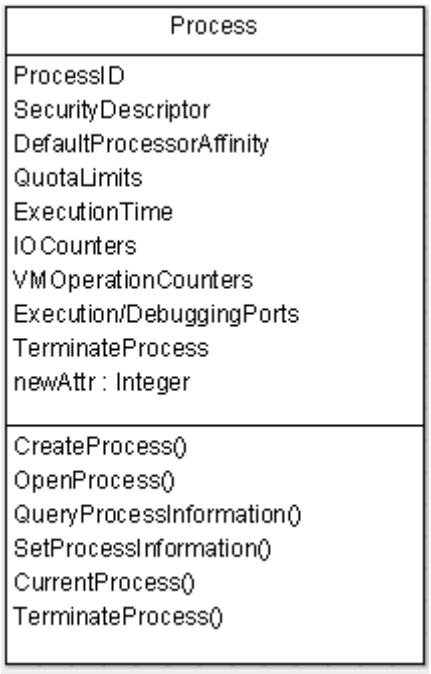

Figure 2.3: A Windows process object. Taken from Stallings (2009) p. 187

2.7.2.1 CreateProcess()

The `CreateProcess()` function creates a process and its primary thread. The function takes the following parameters: the application name, command line to be executed, process attributes, thread attributes, inherit handles, creation flags, a pointer to the environment block, the current directory, start up information and current process information. The resulting process is assigned an identifier and is valid until termination. `TerminateProcess()` can be used to the kill the process by passing in the correct handle to the process that needs to be exited. Additional functions exist in the Windows API that complement this function and include functionality such as returning handles, opening processes, returning process information and so forth (MSDN, 2016a).

2.7.3 Windows' Client/Server Model

Windows' services, subsystems and IPC mechanisms are structured according to a client-server model of computing which aims to simplify the Executive, improve reliability and provide a base for distributed computing (Stallings, 2009). Windows' IPC mechanisms are abstracted by means of this model where a client is a process that requests data from another process. A server responds to such requests (Lee, 2016). The following subsection discusses how applications can communicate.

2.7.4 Process Communication Mechanisms

Windows provides the following IPC mechanisms as per the APIs provided by Microsoft (Windows API).

2.7.4.1 Clipboard

The clipboard acts as a depository which allows applications to share data. An application deposits data into the clipboard which another application can later retrieve in a format it understands. Applications can be running on the same machine or across a network. Clipboards are user driven and should only send data upon a user command and not in the background (i.e. not without the user's knowledge). The clipboard represents a single central depository which multiple other applications can access, as such a process cannot create its own clipboard. (Lee, 2016).

Key functions such as `OpenClipboard`, `EmptyClipboard`, `SetClipboardData`, `CloseClipboard` and `GetClipboardData` are used (Vinoth, 2014). `SetClipboardData` deposits data on the clipboard in a particular format. `GetClipboardData` retrieves data from the clipboard in the format specified (MSDN, 2016a).

2.7.4.2 Component Object Model (COM)

The Component Object Model can be used to create software components that can interact. According to MSDN (2016a), it is a standard that defines how COM objects interact with other objects. Objects can reside in a single process or in multiple other processes. COM objects can access data and an object's data through a set of functions

known as interfaces. The functions that are part of the interface are known as methods. Interface methods are accessed through a pointer to that interface. Common functions exist between all COM objects; i.e. they are mandatory functions that all components require.

2.7.4.3 Data Copy

By making use of `WMCOPYDATA` that is part of Windows Messaging, an application can send data to another application. In order to use data copying, the receiving process needs to be aware of the format of the data being sent as well as identity of the sending process. Data can be encapsulated within a private data structure. It can be sent to the receiving application along with the a pointer to the data structure using `WMCOPYDATA` (MSDN, 2016a).

2.7.4.4 Dynamic Data Exchange (DDE)

MSDN (2016a) defines DDE as an extention of the Clipboard as it makes use of the same formats. Data can be exchanged at an ongoing rate, or as new data becomes available. MSDN (2016a) desribes DDE as not as efficient as newer IPC mechanisms. Its important to note that the DDE makes use of shared memory to exchange data between appilcations.

DDE contains a .dll file called Dynamic Data Exchange Management Library (DDEML) that processes can use to share data. The library provides functions and messages that adds DDE to applications. DDEML makes use of a conversation concept that makes it easier for the programmer to manage messages; pointers and atomic access to shared memory are replaced by making use of string handles. In addition, DDEML forces applications to implement DDE in a consistent fashion. Programs should have the DDEML header file within its source code (MSDN, 2016a).

A DDE client can start a conversation with a DDE server and can participate in multiple conversations simultaneously. A process can be a client or a server as necessary. DDE conversations are identified by the process/application name and a “topic”. The topic represents the data that is to be exchanged between processes (MSDN, 2016a).

2.7.4.5 File Mapping

File mapping allows a process to see the contents of a file as a block of memory within its own address space. This is essentially a form of shared memory. Pointer operations are used to access and modify contents of the mapped file with some form of synchronisation mechanism (such as a semaphore) used to prevent corruption and maintain the file's consistency. This IPC technique can only work on a single machine and the file cannot exist on a remote computer (MSDN, 2016a).

Vinoth (2014) states that a file mapping can be created by using the function `CreateFileMapping`. The file map view is then put into address space of the current process by using `MapViewOfFile`. `CopyMemory` is then used to write messages to the view. File mappings are closed using `UnmapViewOfFile` and `CloseHandle`. A client process can examine the contents of the mapping using `OpenFileMapping` and put the view into its address space by calling `MapViewOfFile`. It can also close mappings as necessary.

2.7.4.6 Mailslots

Windows Mailslots follow the client-server model with one-way process communication. A server process will create a Mailslot and the client can write messages into the Mailslot as required. The messages are saved within the Mailslot until read. Messages can be sent on a local host or across a network. Message size is only limited by what the server specifies when the slot is created (MSDN, 2016a).

Mailslot messages follow a datagram format and as such, no guarantees are made in terms of receipt. Mailslots receive a handle when created which is used when the process that created it accesses data within it. A process that is aware of a Mailslot's existence can insert a message there. Mailslots also have a broadcasting capability where each process can create a slot and all participating processes can deposit their messages to the respective Mailslots that exist. This means that a Mailslot can be a server and client which can allow two-way communication (MSDN, 2016a).

Mailslots can be created using `CreateMailslot()` and its contents examined using `ReadMailslot()`. A number of auxiliary functions exist such as `GetMailslotInfo()` and so forth. Handles can be closed using `CloseHandle()`. Clients can create a file and write to the slot using `CreateFile()` and `WriteFile()`/`WriteMailslot()` (Vinoth, 2014).

2.7.4.7 Pipes

Windows uses two types of pipes for IPC: anonymous and named pipes.

Anonymous pipes only allow related processes to exchange data and cannot be used over a network. They are typically used to exchange data between parent and child processes by using read and write handles respectively. `CreatePipe()` returns read and write handles for an anonymous pipe and specifies a buffer size. These handles are passed to another process to communicate, usually by means of inheritance where the handle is passed from the parent (server) process to the child (client) process. The relevant handle is sent depending on whether a read or write operation must occur. `ReadFile()` is used to read from the pipe and conversely `WriteFile()` is used to write to the pipe. `CloseHandle()` closes a process's pipe 'connection' (MSDN, 2016a). Asynchrony is not supported by anonymous pipes (Lewandowski, 1997).

According to MSDN (2016a), named pipes can be used between unrelated processes and across a network. Processes can act as a server or client without having to create multiple anonymous pipes. `CreateNamedPipe()` and `ConnectNamedPipe()` are used by the server and client respectively. Standard `ReadFile()` and `WriteFile()` functions read and write to the pipe as necessary.

2.7.4.8 Remote Procedure Call (RPC)

Similar to RMI discussed above, RPC allows procedures to be called remotely, either on a local machine or across a network. MSDN (2016a) states RPC conforms to the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) which allows RPC to work on other OSs that support DCE.

2.7.4.9 Windows Sockets (Winsock 2)

Winsock creates a channel between communicating processes and is protocol independent with asynchronous communication capabilities (Lewandowski, 1997). Socket handles can be treated as a file with usual I/O operations (MSDN, 2016a).

Data communicated by processes can be stream-based in the form of bytes or packetised in the form datagrams but no quality of service is guaranteed (Vinoth, 2014). In terms of local IPC, localhost can be used to send and receive data, as long as the correct IP addresses are specified.

2.7.5 Windows Interprocess Synchronisation

The Windows API provides functions that allow processes to synchronise as necessary. This is particularly important with regards to shared memory when there is competition for resources. Functions such `CreateEvent()`, `CreateSemaphore()`, `CreateMutex()` among others return handles that processes can use for the same event. These are generally used in conjunction with the security parameter defined in `CreateProcess()` (MSDN, 2016a).

2.8 IPC in Mac OS X and Generic Considerations

2.8.1 Mac OS X

Mac OS X generally treats other processes as hostile to enforce security mechanisms in its IPC design. IPC mechanisms used are Mach Messaging, Distributed Objects, Shared Memory and Signals (Mac Developer Library, 2016).

2.8.2 Generic Considerations

Microsoft Windows, Linux-based OSs with many flavours and Mac OS X are the most popular and most well-known OSs. As such generic considerations should be made. In terms of IPC, a common form of IPC would be shared memory. From the review conducted above, shared memory is common to the three and can provide a reasonably efficient method in which to implement IPC (as illustrated by SMOF). A generic version could be developed that utilises hese concepts and suitably abstracts away OS dependencies.

Since a Linux version of Java IPC has been developed by Wells (2009), and subsequent to the windows implementation proposed, future work could be completed in achieving a Mac OS version. A generic design then could be implemented that would aim to be OS independent.

2.9 Chapter Summary

The current trend of multicore computing emphasises the significance of parallelism, multithreading and communication in general (Taboada *et al.*, 2013). It is important to utilise these concepts to make software scalable (Slinn, 2012).

Investigation of Java's concurrency mechanisms found that the language is heavily thread oriented and that it provides a good mechanism for multithreading programming. Opposed to this is the language's lack of communication between separate JVM processes.

The existing Linux and Solaris implementation that used JNI to provide Java with IPC was successful and proved that a more efficient method of IPC could be developed without using the loopback method and "clunky" socket programming. The review of the use of JNI highlighted a few efficiency problems with calls to native code, therefore calls to JNI should be as limited as possible. In addition, the SMOF proved that portability issues can be addressed, but it may limit IPC communication to shared memory. As such, use of common IPC features among different OSs should be taken into account when developing a generic version of the Java class library.

Windows provides a fairly comprehensive set of IPC mechanisms that can be integrated with Java processes, albeit at the cost of complexity when compared to Unix system calls.

Mac OS X provides mechanisms that are common to Windows and Linux; namely shared memory. This can definitely be considered as a goal to aim for in terms of a generic solution to this problem.

Chapter 3

Project Approach

The following chapter provides a broad, high-level view of the methodologies used during the design and implementation phase of the project.

Section 3.1 discusses the hardware that was used for the project. Section 3.2 discusses the software and other related technologies that were used during development as well as the results phase of the project. This includes critical software required by the system as well as other non-critical software that supplemented the development process. Section 3.3 discusses broadly how I designed and implemented the actual system. It outlines the order in which I implemented each Windows IPC mechanism, why this was the case and any benefits this may have yielded. Finally, section 3.4 discusses the methodology used to collect and analyse results.

3.1 Hardware

I used one primary desktop machine for the majority of the project's development. It had the following specifications:

- Intel Core i5-6400 Quad-Core CPU @ 2.70 GHz.
- 8 GB DDR3 RAM.
- WDC 400 GB HDD.
- Intel HD Graphics 530 with 4 116 MB VRAM.

- Windows 10 Enterprise 64 bit Insider Preview Build 14393.

In addition, a secondary machine was used for development and testing purposes:

- Intel Core i3-2350 Quad-Core Sandy Bridge CPU @ 2.30 GHz.
- 4 GB DDR3 RAM @ 655 MHz.
- WD 1 TB HDD
- NVIDIA GeForce GT 520MX GPU with 1024 MB VRAM.
- Windows 10 Home 64-bit Build 10856.

3.2 Software

Various software packages were used for the project, including critical and non-critical support packages.

3.2.1 Critical Software

Since the project's focus is on IPC mechanisms on a Microsoft Windows platform, it is palpable that a Microsoft Windows operating system was used. As described in Section 3.1, two versions of 64 bit Windows 10 were used on two separate machines during the implementation and testing phases of the project. I consciously made the decision to develop on Windows 10 because it generally provides consistent backwards compatibility (Scanlon, 2015). The Windows API is also fully supported by Windows 10 OSes (MSDN, 2016a). In addition, I thought using Windows 10 made the most sense since it is Microsoft's latest OS product. In addition, I wanted to install the Windows 10 Insider Preview to test Bash On Ubuntu On Windows [1] (which will be discussed later in this text).

Another fundamental software component of this project is the Java Development Kit. I made use of the Java SE Development Kit 7 [2] or JDK 7 as a 32 bit Java Virtual Machine (JVM). This includes the standard Java APIs as well as the required header files needed

[1] https://msdn.microsoft.com/en-us/commandline/wsl/about
[2] http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

to make use of the Java Native Interface. I initially installed a 64 bit version of the JDK and immediately encountered problems in terms of code compilation with programs that used native C code. Upon compilation, my test programs were unable to load the native library and a `UnsatisfiedLinkError` exception occurred, outlining that it could not load a 32 bit .dll on a 64 bit Java platform. The solution to this was to install a 32 bit JDK (and hence 32 bit JVM) and recompile.

In order to compile the native C code that accesses Windows' IPC mechanisms, a C or C++ compiler was needed. Since the project is on a Windows platform, the clear choice was the Microsoft Visual C++ compiler [3]. It integrates with Microsoft's Visual Studio Integrated Development Environment (IDE) products but can also be called from the command line by executing `cl.exe` in the Visual C++ directory, located in Windows' Program Files file structure.

3.2.2 Non-critical Software

The fundamental support package used throughout this project was GitHub. This served as my version control throughout the implementation of each IPC feature. I made use of a Git GUI client called GitKraken[4].

JUNIT[5] is a Java library that provides a unit testing framework. This was used to ensure that Java methods performed as expected.

The programming language R[6] was used in the analysis phase of the project, along with Microsoft Excel 2016. The R source code I developed is available with WindowsIPC on the JavaIPC GitHub repository.

Atom was the code editor used throughout the development and testing process.

[3]https://www.microsoft.com/en-us/download/details.aspx?id=41151
[4]https://www.gitkraken.com/
[5]http://junit.org/junit4/
[6]https://www.r-project.org/

3.3 Design Approach and System Overview

The first phase prior to implementation was to become familiar with the JNI framework. This involved designing simple JNI programs to become competent in terms of how primitive data types are handled, how parameters are passed, how memory is allocated and freed and how values are passed between the JVM and native C code. For example, since Java uses UTF-16 strings that are passed down into the JNI, they have to be fetched using `GetStringUTFChars` which allocates them to a pointer and converts them to UTF-8. They then have to be released using `ReleaseUTFChars` since the memory is not freed upon function return (AndroidDevelopers, 2016). Throughout this process, I followed Liang (1999) which provided well structured and easy to understand JNI examples and tutorials, which greatly aided in the understanding of JNI. This was fundamental during the implementation of WindowsIPC since a thorough understanding of how JNI works was needed to ensure the stability and usability of the system. Concurrent to this, I cemented my C coding ability since a significant majority of time was spent developing in C. I also ensured I knew how to compile and build Java programs that used JNI. This involved understanding the various flags and include files that needed to be added to the command line. As such, I decided to build the system using the command line and not an IDE. As such I made use of Windows Batch scripts that specified system dependencies and files that were needed to compile and execute the code. Listing 3.1 is a snippet of a batch script that generates files required by the system. I preferred to develop in a simple code editor as I found that initialising an IDE to use JNI was rather tedious and not worth the time. In addition, complex debuggers would not have been beneficial since parallel development is notoriously difficult to debug.

Listing 3.1: Batch Script Snippet

```
REM create the header file
"C:\Program Files (x86)\Java\jdk1.7.0_79\bin\Javah" -jni -classpath
"C:\Users\g13s0714\Desktop\CS Honours\GitHub\JavaIPC\Implementation"
WindowsIPC
```

Once I had gained suitable confidence with using JNI to access native C code, I ranked each Windows IPC mechanism in terms of ease of implementation based on research conducted in terms of their use in other systems. This involved understanding the related WINAPI functions. I believe this was a logical method to follow that made the development process significantly easier. I started developing Java Sockets since this is the only

method of IPC that Java offers (as mentioned in Chapter 2). In addition, I wanted to use this as a benchmark to determine if the other mechanisms perform better. This involved timing message sending between two Java programs that made use of the socket.

Once I had a suitable benchmark from developing Java Sockets, I proceeded with the development of the Windows IPC mechanisms. Windows typically implements IPC using a client-server model which I conformed to during my implementation. I first implemented Named Pipes, followed by Mailslots, Windows Sockets, Windows File Mapping (a shared memory equivalent) and finally Data Copy. This entailed analysing and understanding the relevant WINAPI functions that were necessary for their implementation. I found that most of Microsoft Windows' APIs were significantly complex and difficult to understand. This was apparent in the many function parameters and flags that are present in WINAPI functions. I also found that most of the function examples on the Microsoft Developer Network were given in C++ code, which was problematic since WindowsIPC was written in C. As a result, C and C++ syntax is used interchangeably. Ultimately, this did not pose any significant problem, since Visual C++ could handle it's compilation.

Figure 3.1 presents a graphical, high-level overview of the system. It illustrates the problem that two Java programs cannot communicate within the "Java World" but can in the "native C/C++ World". through the use of JNI. The actual communication is done by making use of WINAPI functions which can call the relevant IPC function(s). This allows the system to feed messages back and forth between the Java and C/C++ levels. Note: the term "coordination" refers to both synchronisation and communication.

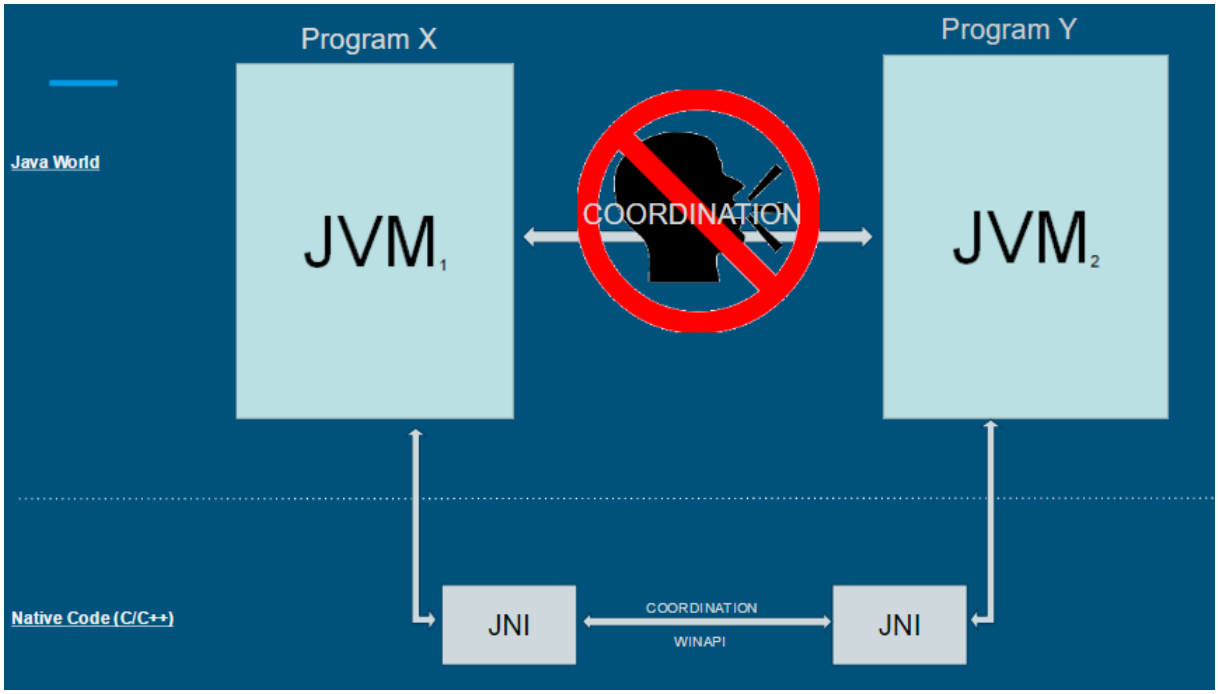

Figure 3.1: High-level Overview of WindowsIPC

During the implementation of each feature, I tested it by writing small Java test programs that used the native code to send messages to each other. Messages sent were simple byte arrays. The size (in terms of the number of elements) of the array sent were simply hardcoded as 40, 400, 4 000 and 40 0000. For example, a byte array with 400 elements represents a message size of 400 bytes. Two Java test programs (for each mechanism) were used to ensure that messages were sent between two distinct Java processes. Following this, I wrote a single Java program for each mechanism that used `java.util.concurrent` to create a thread that communicated with the program's main thread. This was achieved by implementing the `run` method of the `Runnable` interface. The `run` method uses an object of WindowsIPC to call method to send a message to the program's main thread. This was done to ensure true communication was possible (i.e. between separate processes as well as threads executing within a single JVM instance).

3.4 Results Analysis

In order to time the execution of each native method as well as the Java Sockets execution time, I used the following Java code:

Listing 3.2: Java Timing Code

```
long time = System.nanoTime();
  //Method being timed goes here...
System.out.println("Time to get message from file mapping: "+
    ((System.nanoTime() - time))+ "ns");
System.out.println("Message in Java: " + x);
```

For each message size, I timed the execution of the method that sent the actual message 10 times and then calculated a mean execution time. I used this figure to determine a speedup in comparison to Java Sockets to illustrate the performance gain that was achieved. Each timing value was copied into an R script that calculated the mean for the relevant IPC mechanism. I also made use of Microsoft Excel to provide a convenient method in which to view results without having to scan through an R script file. Once the mean execution time for each mechanism was calculated, the R script generated various graphs, including line and bar graphs. This provided a graphical way in which to display each mechanism's performance relative to Java Sockets. Listing 3.3 shows R code that calculates the mean execution time of Java Sockets in micro-seconds for each byte size and then generates a line graph illustrating the relationship.

Listing 3.3: Snippet of R Script that Generates Stats and Graphs

```
JS_VECTOR <- c(
JSOCKETS_40_BYTES / 1000,
JSOCKETS_400_BYTES / 1000,
JSOCKETS_4000_BYTES / 1000,
JSOCKETS_40000_BYTES / 1000
)
png(file = "JavaSocketsLineChart.png")
plot(
  JS_VECTOR,
  col="red",
  xlab="Size of Message in Bytes",
  ylab="Time to Send in MICRO-SECONDS",
  main="Java Sockets Performance",
  type='b',
  xaxt="n",
  ylim=c(6000, 6400)
)
axis(side=1, at=1:4, lab=c("40","400","4 000","40 000"))
```

```
dev.off()
```

The graphs generated aided in the discussion chapter of this text due to the visual illustration of the results. It showed the excellent speed up that can be achieved by making use of this library as opposed to the use of a socket mechanism. The results obtained are discussed in 5

Chapter 4

Implementation of Windows' IPC Mechanisms

4.1 Introduction

This chapter presents an in-depth discussion of the implementation of WindowsIPC. It includes a thorough discussion of the design methodology, problems that were encountered as well as the relative success or lack thereof of each IPC mechanism. In addition, discussion centers around the various WINAPI functions used and why this was the case. Section 4.2 discusses system compilation and file generation. It includes a discussion of the build script that was developed as well as the files generated by the system.

4.2 Files, File Generation and System Compilation

4.2.1 System Files

- WindowsIPC.java
- WindowsIPC.C
- make.bat
- clean.bat

- run.bat
- WindowsIPC.dll (System generated)
- WindowsIPC.h (System generated)
- WindowsIPC.obj (System generated)
- WindowsIPC.lib (System generated)
- WindowsIPC.exp (System generated)

The primary Java file for this system is called `WindowsIPC.java`. This class contains all the native methods and support methods that are needed for the implementation of the system. The system is designed in such a way that users can declare an object of the class WindowsIPC and use it to invoke the relevant IPC functions depending on the needs of the application.

The C file that contains the implementation of the JNI functions is of the same name: WindowsIPC.c. WindowsIPC.h is system generated and contains the function prototypes which are implemented in WindowsIPC.c. The majority of implementation was done in WindowsIPC.c, with tests conducted in small Java programs.

A number of files are system generated and hence not implemented directly. These are namely the files required to access Windows' APIs.

4.2.2 File Generation and System Compilation

As mentioned in Chapter 3, I did not make use of an IDE throughout the implementation phase of this project, but instead used Windows Batch Scripts to build the system and generate the required files.

The primary batch script that I developed is called make.bat. This script performs the initial file generation and Java compilation. It generates the native library called WindowsIPC.dll that the JVM loads to make use of the JNI functions defined in WindowsIPC.h and implemented in WindowsIPC.c. It is important to note that since this script defines directory locations that are specific to the two machines that were used for development, it will not work on any other additional machines without significant modification. The script initially changes the working directory to the local GitHub repository

directory that contains the files of the system. This is performed using a simple `cd` command. It then calls the vsvarsall.bat batch script that sets up the environment variables of that particular terminal session. This script configures the command line for 32 bit or 64 bit compilation and comes standard with a Microsoft Visual Studio installation. This is required to make use of cl.exe (the Visual C++) compiler. When calling vcvarsall.bat, "x86" is passed in as an argument to force the system to be used within a 32 bit context. This prevents any 32 bit versus 64 bit conflicts (as discussed in Section 3.2) when used in conjunction with a 32 bit JVM.(MSDN, 2016b). The script then compiles WindowsIPC.java using the Java compiler (`javac`) followed by generating WindowsIPC.h. This is done by executing `javah` and specifying the classpath of WindowsIPC.java (i.e. the location of the .class file). Following the generation of WindowsIPC.h, `cl` is executed. To run this command and to integrate it successfully with Java's JNI, the correct include directories that contain the required header files needed for the generation of the native library must be specified. This is achieved by using the `-I` flag and then specifying the location of the win32 header files as well as the header files needed by the JNI. The `-LD` flag is used to specify WindowIPC.c and `-FE` specifies the name of the native library (i.e. the name of the .dll file generated by `cl`). In addition, this command generates .lib, .exp and .obj files of the same name. These files are required to access the Windows API. WindowsIPC.java is then executed to ensure that it was correctly compiled and that all the required files were generated successfully.

An additional script called clean.bat simply deletes all generated files. It uses a simple `del` command to delete the files with a specific extention. For example, `del *.class` deletes all generated class files and `del WindowsIPC.dll` deletes the generated native library.

The script run.bat simply executes WindowsIPC.java by calling the Java interpreter `java`. This was written to perform a quick execution of WindowsIPC and hence prevent the need to rebuild the entire system.

WindowsIPC.java loads the native library (WindowsIPC.dll) generated by `cl` by using the following Java syntax:

Listing 4.1: Loading The Native Library

```
static {
  System.loadLibrary("WindowsIPC");
}
```

"WindowsIPC" is the name of the .dll file that contains the required native functions.

This code is specified outside of any method within the Java file, hence it is defined as a static code block. This is performed below the declaration of the native functions.

4.3 Java Sockets

The implementation of Java Sockets does not contain any use of the JNI and simply is implemented as standard Java methods in WindowsIPC.java. Since this implementation is making use of Java's networking capabilities, the `java.net` package needed to be imported. This is necessary to create objects such as `ServerSocket` and `Socket` object and other networking-related objects (Oracle, 2016). There are two method implementations, one represents the socket server and another that represents the socket client. This aims to conform to Windows' client-server model of IPC implementation.

The server method is designed in such a way that it allows a client to connect to it, then pass data to it as it likes. The method then simply returns the client message as a byte array. The method is called `createJavaSocketServer` that takes a port number as an argument. Some error checking is in place that ensures the port number is valid and unprivileged. If an invalid port number is passed in, an error is raised. Otherwise, a `SocketServer` object is created that passes the port number into its constructor. Then a `Socket` object is created and initilised by calling `accept` on the original `ServerSocket` object, as depicted in Listing 4.2. The `accept` method waits for a client connection.

Listing 4.2: Java Sockets: `createJavaSocketServer`

```
public byte[] createJavaSocketServer(int port) {
  byte[] messageRcv = null;
  ServerSocket serverSocket = new ServerSocket(port);
  Socket server = serverSocket.accept();
  DataInputStream in = new DataInputStream(server.getInputStream());
  int length = in.readInt();
  messageRcv = new byte[length];
  if (length > 0)
    in.readFully(messageRcv, 0, messageRcv.length);
  server.close();
  // some excption handling...
  return messageRcv; // return the message sent from the client
}
```

A `DataInputStream` object, called `in` is created. This object is used to extract the message sent by the client from the socket connection. It does this by invoking `getInputStream` using the `Socket` object. A byte array called `messageRcv` is declared which is used to store the message received from a client connection. The size of the array is specified by determining the length of the message received using `in` and calling `readInt` on it. If there is a message, then `readFully` gets the message and dumps it into the the byte array. The server is then closed and the byte array message is returned. All of this code is wrapped in a try-catch block. It catches any `IOExceptions` that may be raised. It has been omitted from Listing 4.2.

The Java Socket client was implemented in Java method called `createJavaSocketClient`. This method returns an integer value that represents whether the method executed as expected or not. A returned value of zero indicates that the method executed as expected. -1 indicates an error occured. The method takes the host name, port and message as arguments. The host name, in this case, should always be localhost (127.0.0.1). The port number should be above 1024 to ensure privileged port numbers are not used. The programmer who makes use of this method should specify 127.0.0.1 and the port number that was specified when the server was created. The message parameter is a simple byte array. To create the client, an object of `Socket` called `client` is created. This object is instantiated by passing loopback and the port number to the `Socket` constructor. An `OutputStream` object is instantiated by invoking `getOutputStream` on `client`. This is the primary output stream that is used to send the byte array message to the server. This is achieved by passing this object to a new `DataOutputStream` object's constructor. This object is called `out`. The actual message is sent by using `write` which called on `out` and takes the message as an argument. The client is then closed. If any error is occured (such as an `IOException`), then -1 is returned, otherwise zero is returned which indicates the method successfully sent the message to the existing server application. This method also includes some exception handling code, which is required when using `Socket` objects. This process is illustrated in Listing 4.3 below. Note: It also omits the exception handling code.

Listing 4.3: Java Sockets: `createJavaSocketClient`

```
public int createJavaSocketClient(String host, int port,
                                   byte[] message)
{
    Socket client = new Socket(host, port);
    OutputStream outToServer = client.getOutputStream();
```

```
        DataOutputStream out = new DataOutputStream(outToServer);
        out.writeInt(message.length);
        out.write(message);
        client.close();
        return 0; // success
    }
```

Overall, the implementation of Java Sockets using `Socket` objects was relatively straightforward, with little difficulty.

4.4 Named Pipes

4.5 Anonymous Pipes

4.6 Mailslots

4.7 Windows Sockets

4.8 File Mapping

4.9 Data Copy

4.10 Dynamic Data Exchange

4.11 Clipboard

4.12 Component Object Model

Chapter 5

Results

Chapter 6

Discussion

Chapter 7

Conclusion

References

AndroidDevelopers. 2016. *JNI Tips*. `https://developer.android.com/training/articles/perf-jni.html`. [Date Accessed: 18 August 2016].

Beveridge, J, & Wiener, R. 1997. *Multithreading Applications in Win32*. Reading, Massachussets: Addison-Wesley.

Bovet, D, & Cesati, M. 2003. *Understanding the Linux Kernel*. 2nd edn. Sabastopol, CA, USA: O'Reilly.

Dawson, M., Johnson, G., & Low, A. 2009. *Best practices for using the Java Native Interface*. http://www.ibm.com/developerworks/library/j-jni/. [Date Accessed: 18 March 2016].

Dijkstra, Edsger W. 2002. The Origin of Concurrent Programming. New York, NY, USA: Springer-Verlag New York, Inc.

Garg, Vijay K. 2005. *Concurrent and Distributed Computing in Java*. Hoboken, New Jersey: John Wiley & Sons.

Goetz, B, Peierls, T, Bloch, J, Bowbeer, J, Holmes, D, & Lea, D. 2006. *Java Concurrency in Practice*. Addison-Wesley.

Hayes, Brian. 2007. Computing Science: Computing in a Parallel Universe. *American Scientist*, **95**(6), 476–480.

Hyde, P. 1999. *Java Thread Programming*. Indianapolis, USA: Sams Publishing.

Lea, D. 1999. *Concurrent Programming in Java: Design Principles and Patterns*. 2nd edn. Canada: Addison-Wesley.

Lea, D. 2005. The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, **58**(3), 293 – 309. Special Issue on Concurrency and Synchonization in Java Programs.

Lee, T. 2016. *Interprocess Communications*. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx`. [Date Accessed: 22 April 2016].

Lewandowski, Scott M. 1997. Interprocess Communication in UNIX and Windows NT. *Brown University*.

Liang, S. 1999. *The Java Native Interface: Programmer's Guide and Specification*. Palo Alto: Addison-Wesley Professional.

Mac Developer Library. 2016. *Validating Input and Interprocess Communication*. `https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/ValidatingInput.html#//apple_ref/doc/uid/TP40007246-SW3`. [Date Accessed: 29 April 2016].

Magee, J, & Kramer, J. 2006. *Concurrency: State Models & Java Programming*. Chichester, England: John Wiley & Sons, Ltd.

MSDN. 2016a. *Microsoft API and Reference Catalog*. `https://msdn.microsoft.com/en-us/library/`. [Date Accessed: 22 April 2016].

MSDN. 2016b. *Setting the Path and Environment Variables for Command-Line Builds*. `https://msdn.microsoft.com/en-us/library/f2ccy3wt.aspx`. [Date Accessed: 16 August 2016].

Oracle. 2010. *Introduction to the Java Programming Environment*. https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqb/index.html. [Date Accessed: 23 February].

Oracle. 2016. *Java Platform Standard Edition 8 API Specification*. `http://docs.oracle.com/javase/8/docs/api/`. [Date Accessed: 6 March 2016].

Russinovich, M, Solomon, D. A., & Ionescu, A. 2012. *Windows Internals Part 1*. 6th edn. Microsoft Press.

Russinovich, M. E., & Solomon, D. A. 2009. *Windows Internals Covering Windows Server 2008 and Windows Vista*. 5th edn. Redmond, Washington: Microsoft Press.

Scanlon, J. 2015. *How to Run Old Programs on Windows 10*. `http://www.techradar.com/news/software/operating-systems/how-to-run-old-programs-on-windows-10-1300470`. [Date Accessed: 11 August 2016].

Schauble, C, J C. 2003. *Distributed Operating Systems: Disadvantages of Semaphores.* `http://www.cs.colostate.edu/~cs551/CourseNotes/ConcurrentConstructs/DisAdvSems.html`. [Date Accessed: 22 March 2016].

Slinn, M. 2012. *Benchmarking JVM Concurrency Options for Java, Scala and Akka.* `http://www.infoq.com/articles/benchmarking-jvm`. [Data Accessed: 29 April 2016].

Stallings, W. 2009. *Operating Systems: Internals and Design Principles.* 6th edn. Upper Saddle River, NJ: Pearson Education International.

Taboada, Guillermo L, Ramos, Sabela, Expósito, Roberto R, Touriño, Juan, & Doallo, Ramón. 2013. Java in the High Performance Computing arena: Research, Practice and Experience. *Science of Computer Programming*, **78**(5), 425–444.

Tanenbaum, A. S., & Bos, H. 2015. *Modern Operating Systems.* 4th edn. Edinburgh: Pearson.

Vinoth, R. 2014. *IPC Mechanisms in Windows.* `http://www.slideshare.net/mfsi_vinothr/ipc-mechanisms-in-windows`. [Date Accessed: 22 April 2016].

Wells, G. C., & Anderson, M. 2013. Efficient Interprocess Communication in Java. *Department of Computer Science, Rhodes University*, 1–7.

Wells, G.C. 2009. Interprocess Communication in Java. *Pages 407–413 of:* Arabnia, H.R. (ed), *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09).* Las Vegas: CSREA Press.

Wells, G.C. 2010 (Sept.). Extending Java's Communication Mechanisms for Multicore Processors. *In: 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010).* (Poster).

Wisniewski, Robert W, Kontothanassis, Leonidas, & Scott, Michael L. 1994. Scalable spin locks for multiprogrammed systems. *Pages 583–589 of: Parallel Processing Symposium, 1994. Proceedings., Eighth International.* IEEE.