SACJ 29(3) December 2017
Research Article

Interprocess communication with Java in a Microsoft Windows Environment

Dylan Smith, George Wells

Department of Computer Science, Rhodes University, Grahamstown, 6140, South Africa

ABSTRACT
The Java programming language provides a comprehensive set of multithreading programming techniques but currently lacks interprocess communication (IPC) facilities, other than slow socket-based communication mechanisms (which are intended primarily for distributed systems, not interprocess communication on a multicore or multiprocessor system). This is problematic due to the ubiquity of modern multicore processors, and the widespread use of Java as a programming language throughout the software development industry. This work aimed to address this problem by utilising Microsoft Windows' native IPC mechanisms through a framework known as the Java Native Interface. This enabled the use of native C code that invoked the IPC mechanisms provided by Windows, which allowed successful synchronous communication between separate Java processes. The results obtained illustrate the performance dichotomy between socket-based communication and native IPC facilities, with Windows' facilities providing significantly faster communication. Ultimately, these results show that there are far more effective communication structures available. In addition, this work presents generic considerations that may aid in the eventual design of a generic, platform-independent IPC system for the Java programming language. The fundamental considerations include shared memory with semaphore synchronisation, named pipes and a socket communication model.

Keywords: Java, Windows, interprocess communication

Categories: • **Computing methodologies ∼ Concurrent computing methodologies** • *Computing methodologies ∼ Parallel computing methodologies*

Email:
Dylan Smith dylan.smith888@gmail.com,
George Wells G.Wells@ru.ac.za (CORRESPONDING)

Article history:
Received: 25 Apr 2017
Accepted: 17 Oct 2017
Available online: 8 Dec 2017

1 INTRODUCTION

Java is a widely used general-purpose programming language that supports an object-oriented programming model with portability and simplicity (Oracle, 2010). The language provides comprehensive multithreading capabilities, such as thread creation, communication and synchronisation. However, the language currently lacks support for interprocess communication (IPC) but instead relies on distributed network programming mechanisms (Wells, 2010).

This means that Java programmers are forced to use socket communication for communication between Java processes (i.e. Java programs executing in separate Java Virtual Machines with their own distinct address space) executing on the same machine. This approach is inefficient, as it forces communication to traverse the layers of the network 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. In particular, the use of multiprocessing, rather than multithreading, allows for a robust separation of concerns, and improved fault tolerance. Our investigation of IPC in Java was prompted by considering the provision of a Linda system for multicore systems in Java (Gelernter, 1985). In this context, it is desirable, from a system-design point of view, to separate the Linda tuple space server component from the rest of the application, rather than have it running as a thread within the client application. Fault tolerance is also improved, as the tuple space server will not be halted if the client application encounters a run-time error (or *vice versa*).

Java, however, does provide a mechanism by which it can access native code, known as the Java Native Interface (JNI). The JNI 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 native code written in C or C++ (Liang, 1999), providing access to the low-level operating system features (memory, I/O, IPC mechanisms, and so forth) (Dawson, Johnson, & Low, 2009). As such, the JNI framework can be used to develop an alternative to Java socket-based IPC, allowing programmers to access low-level features provided by the operating system.

The Linux and Oracle 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 on Microsoft Windows or any other operating system. The Linux and Solaris implementations showed promising results, and an attempt to replicate these performance benefits on a Microsoft Windows platform served as the primary motivation for this research. This required significant research into the internal Windows IPC mechanisms and structures, including communication and synchronisation mechanisms, in addition to various implementation methodologies. **WindowsIPC** is the name given to the Java library designed in this research project.

Ultimately, a useful goal would be to describe potential generic IPC features that could serve as an implementation guide to a design that is operating system independent. This would take the form of a Java IPC library that could be used on Microsoft-based operating systems, various Unix flavours, and Apple's Mac OS X, thus covering the great majority of systems in use today.

2 CONCURRENCY IN JAVA AND MICROSOFT WINDOWS

The Java programming language provides comprehensive support for multithreaded programming by means of the `Thread` class and the `java.util.concurrent` package. From its inception, the language provided multithreading programming facilities through means of the `java.lang.Thread` class, and methods declared as `synchronized`, which utilise monitor concepts such as `wait`, `notify` and `signal`. Version 5.0 of the Java platform introduced high-level APIs for concurrency in the language in the form of the `java.util.concurrent` package (Oracle, 2016) (a brief overview of threads and the concurrency library are provided in Sections 2.1 and 2.2 below).

In terms of IPC, Java only supports socket communication, using the `localhost` "loopback" network (Wells, 2009). This means that network-based communication mechanisms such as Remote Method Invocation (RMI), which is intended primarily as a distributed computing mechanism, can also be used for IPC.

2.1 java.lang.Thread

Java provides the `java.lang.Thread` class that allows the explicit creation and control of thread objects (Garg, 2005). Java threads have their local variables organised as a stack, and can access shared instance variables. Threads can be viewed as lightweight processes running within the context of a single Java Virtual Machine (JVM, the run-time byte code interpreter used for executing Java programs) (Magee & Kramer, 2006). Java threads support priorities and preemption, with minimum and maximum priority values, and methods to query and modify priorities. This can heuristically affect operating system schedulers (Lea, 1999). According to Hyde (1999), making use of Java threads can have some benefits as well as drawbacks. Benefits 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 I/O operations. Drawbacks exist, such as the need for the instantiation of `Thread` objects, which introduces overhead and use of memory resources. They also require processor resources for the 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 running in separate JVMs. If a thread calls a method declared as `synchronized`, its execution will be blocked unless the object in question is unlocked, or the calling thread already holds the synchronisation lock for that object (Brosgol, 1998).

2.2 java.util.concurrent

The package `java.util.concurrent` (which is based on the standard `Thread` mechanisms) provides concurrency classes that include synchronisation, concurrent data structures and thread life-cycle management (Lea, 2005). These facilities are sufficient for developing good multithreaded applications, and aim to eliminate common problems that can arise in multithreaded programming, such as deadlock, race conditions and unintentional thread interactions (Wells & Anderson, 2013).

In terms of concurrent data structures, the `java.util.concurrent` package provides classes such as `ConcurrentLinkedQueue`, which contains methods such as `peek`, `poll` and so forth which allow threads to access concurrent data safely. The `Executor` framework provides for the creation and life-cycle management of threads in various sophisticated and efficient ways. The `java.util.concurrent` package offers synchronisation classes that provide counting semaphores, barriers, latches, and synchronised data exchange facilities.

The mechanisms provided by `java.util.concurrent` ultimately require a shared address space for JVM threads to synchronise, and have access to shared objects (Wells, 2009). Since Java uses the classical monitor concept (Hoare, 1974), threads can be susceptible to deadlock, therefore great care is required when performing synchronisation.

2.3 Interprocess Communication

Current Java IPC mechanisms rely on network programming with the “loopback" network. For higher level mechanisms, the distributed programming features based on network protocols can be used, such as Remote Method Invocation (RMI) and the Java Message Service (JMS).

According to Wells (2009), Java provides a solution for interprocess communication by using distributed computing with the Internet Protocol’s (IP) loopback network (i.e. using the 127.0.0.0/8 network address family). This means that communication can take place between multiple processes on a single machine. This takes the form of socket programming using network protocols (such as TCP and UDP), through the use of Java’s standard socket API (Taboada, Ramos, Expósito, Touriño, & Doallo, 2013). Messages can be encapsulated into network packets and passed through the IP stack as necessary. Messages are passed to programs running in separate JVMs using the loopback network, thereby providing a form of IPC. Taboada et al. (2013) state that Java’s network support over TCP/UDP is an inefficient communication mechanism. In addition, research conducted by Wells (2010) also found it to be inefficient. These findings are confirmed by our results (see Section 4).

Remote Method Invocation (RMI) allows methods to be called in an object that are running within the context of another JVM. RMI classes are effectively a homogeneous set of communicating JVMs (Waldo, 1998). Arguments of the called method are packetised and sent over a network to another JVM, where they are 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 network 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, with multi-language support (Wells, 2010). Yet another form of distributed communication is the Java Message Service (JMS) which provides communication between applications (Oracle, 2016). These high-level distributed mechanisms provide a solution to IPC but, according to Taboada et al. (2013), at the cost of poor performance.

It should be emphasised that the standard facilities for concurrent programming in Java are inherently thread-based as opposed to process-based. The current network communication mechanism through the IP stack has significant overhead when used for processes on a single computer system.

2.4 Java IPC in Linux and Solaris

Linux- and Solaris-based libraries providing IPC in Java were 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 pipes. The implementation aimed to make the native Java methods resemble the system calls as closely as possible. This includes the method names and parameter lists. Problems were encountered with the differences in data representation between Java and C. Issues such as differences in error handling (i.e. exceptions vs. returning −1 as an error flag), control system calls, semaphore operations and shared memory were also found.

Simple communication benchmarking was performed and compared against the network loopback connection. It was found that all the Unix System V IPC mechanisms were significantly faster than the loopback method. Named Pipes were found to be the fastest, partly due to the fact that they made minimal use of JNI calls. This was the case for tests running on Ubuntu 8.05.1 and Solaris.

The initial implementation contained some portability issues since explicit calls to native code were being made. A Shared Memory Object Framework (SMOF) was developed by Wells and Anderson (2013) that aimed to create a more abstract solution to Java's IPC problems, thereby simplifying its porting to an operating system such as Windows. The SMOF provides the programmer with classes that are based on 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 a shared object by using a *key*, which is a file name that acts a point of reference. Testing involved assessing the latency of sending simple messages between processes. The loopback approach and Named Pipes were used as benchmarks. SMOF was found to be 80% faster than Named Pipes and 87% faster than the loopback mechanism (Wells & Anderson, 2013).

2.5 Interprocess Communication in Windows

Since the initial implementations that had been developed on Linux and Solaris had shown good performance benefits (Wells, 2009), the next logical step was to develop a Windows version. As such, it requires an investigation into the native IPC mechanisms offered by the operating System. Windows offers 11 mechanisms, with **Named Pipes**, **Anonymous Pipes**, **Mailslots**, **Windows Sockets**, **File Mapping**, **Semaphores** and **Data Copy** implemented in this project. **Clipboard**, **Component Object Model (COM)**, **Dynamic Data Exchange (DDE)** and **Remote Procedure Calls (RPC)** were discarded for a number of reasons. The Clipboard is primarily event-based, with a user "cutting and pasting" data between applications. COM and DDE are older technologies and are extremely complex to integrate into native C code via the JNI. RPC is a distributed computing mechanism, with RMI providing a much cleaner solution in Java. The remainder of this section discusses the mechanisms used in the design and implementation of WindowsIPC.

Windows provides two types of pipe communication mechanisms: 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'. Asynchronous communication is not supported by Anonymous Pipes (Lewandowski, 1997). According to the Windows API, 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 pipes. `CreateNamedPipe` and `ConnectNamedPipe` are used by the server and client respectively. Standard `ReadFile` and `WriteFile` functions read and write from/to the pipe as necessary.

Windows Mailslots follow the client-server model with one-way interprocess 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 within a single system, or across a network. Message size is only limited by what the server specifies when the slot is created.

Winsock creates a channel between communicating processes and is protocol independent with asynchronous communication capabilities. Data communicated by processes can be stream-based in the form of bytes or packetised in the form of datagrams, but no quality of service is guaranteed (Vinoth, 2014). In terms of local IPC, the `localhost` network can be used to send and receive data, as long as the correct IP addresses are specified.

File Mapping allows a process to see the contents of a file as a block of memory within its own address space, and can be used by multiple processes as 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 Mapping's integrity. This IPC technique can only work on a single machine, and the file cannot exist on a remote computer.

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

The Windows API provides functions that allow processes to synchronise as necessary. This is particularly important with regards to shared memory, and when there is competition for resources. Functions such `CreateEvent`, `CreateSemaphore` and `CreateMutex`, amongst others, return handles that processes can use for synchronising their activities.

2.6 Summary

The current trend of multicore computing emphasises the need for parallelism, multithreading and communication in general (Taboada et al., 2013). It is important to utilise these concepts to make software scalable (Slinn, 2012), and to make full use of multicore processors. Investigation of Java's concurrency mechanisms found that the language is heavily thread oriented, and that it provides good support for multithreaded programming. In contrast to this is the language's lack of provision for communication between separate JVM processes. The existing Linux and Solaris implementations that used JNI to provide Java with IPC were successful and proved that a more efficient method of IPC could be developed without using the loopback network and inefficient socket programming.

Previous work has shown that JNI introduces performance overheads when calling native code (Wells, 2009), 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, the support for common IPC features among different operating systems should be considered when considering a generic version of Java IPC.

Windows provides a comprehensive set of IPC mechanisms that can be integrated with Java processes, albeit with considerable complexity when compared to Unix IPC system calls. Mac OS X is based on Unix, and provides mechanisms that are common to Windows and Linux, namely shared memory and named pipes. POSIX IPC and System V IPC, which are provided by Mac OS X, are only common to Unix-based operating systems, and cannot be used in a Windows implementation.

3 DESIGN AND IMPLEMENTATION

The first phase of this project involved becoming familiar with the JNI framework, and required an understanding 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. Once sufficient understanding of JNI had been gained, work began on designing the native C library. Eventually the following Windows IPC mechanisms were adopted: *Named Pipes*, *Anonymous Pipes*, *Mailslots*, *Windows Sockets*, *File Mapping*, *Windows Semaphores* and *Data Copy*. Each mechanism follows a client-server model of implementation. Java Sockets were also used, and served as a simple baseline performance benchmark to determine whether the native communication facilities could provide better performance. The IPC mechanisms are provided by a Java class called `WindowsIPC`, with public native methods that can be invoked by instantiating a new object of this class. Smith (2016) gives full implementation details.

3.1 Named Pipes

The implementation of Named Pipes provides synchronous communication of bytes between server and client Java programs. A server program initially creates a Named Pipe (which is simply a file path), which a client Java program can connect to. The client sends its byte array message through the pipe and into the server process. The server code simply returns the byte array received through the JNI back to the Java program. The server and client functions terminate once this communication has taken place. The server process makes a number of WINAPI calls, with `CreateNamedPipe` creating the pipe and `ReadFile` reading the data written to it. The client process returns an integer value specifying its success. Since Named Pipes appear as a file path, utilising Java's standard I/O mechanisms to read and write data from/to the pipe provided an additional performance boost due to avoiding an additional JNI call (see Section 4.2).

3.1.1 Implementation Details

To provide a deeper insight into the use of JNI to provide access to the underlying Windows IPC mechanisms, more detail is provided here for the case of Named Pipes. The other IPC mechanisms required similar approaches and are not described in as much detail (as already noted, full details for all the IPC mechanisms implemented can be found in Smith (2016)).

The native method for creating a Named Pipe is defined in the `WindowsIPC` class as follows:

```
public native byte[] createNamedPipeServer(String pipeName);
```

This is subsequently implemented as the following C function in the native code DLL:

```
JNIEXPORT jbyteArray JNICALL Java_WindowsIPC_createNamedPipeServer
    (JNIEnv * env, jobject obj, jstring pipeName) {
    ... // function code
    } // Java_WindowsIPC_createNamedPipeServer
```

The parameter `jstring pipeName` refers to the pipe name passed into the JNI code from the Java code (and is given the same name here to make this clear).

The Windows function `CreateNamedPipe` creates the actual pipe, and returns a handle to the newly-created pipe object. The Windows function call is made as follows, with suitable default values provided for the parameters (`nameOfPipe` is the name specified by the parameter, `pipeName` above, after conversion from Java's string format to that used by C):

```
pipeHandle = CreateNamedPipe (
    nameOfPipe,
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_MESSAGE |
    PIPE_READMODE_MESSAGE |
    PIPE_WAIT,
    PIPE_UNLIMITED_INSTANCES,
    BUFFER_SIZE,
    BUFFER_SIZE,
    NMPWAIT_USE_DEFAULT_WAIT,
    NULL
    );
```

This example also highlights the complexity common to almost all the Windows system calls required to make use of its IPC facilities. `PIPE_ACCESS_DUPLEX` forces the pipe to be full-duplex, allowing reading and writing to/from either end of the pipe (i.e. the pipe is bi-directional). `PIPE_TYPE_MESSAGE` specifies that the messages sent across the pipe are in the form of a message stream. `PIPE_READMODE_MESSAGE` similarly specifies that messages read from the pipe are in the form of a message stream. `PIPE_WAIT` indicates that the process must block until another process has completed reading the data from the read end. `PIPE_UNLIMITED_INSTANCES` specifies the number of instances created for the pipe. In this case, the constant specifies that 255 may be created. `BUFFER_SIZE` specifies the size of the input and output buffers of the pipe. This is a constant defined using the standard

C `#define` directive with a value of 50 000. `NMPWAIT_USE_DEFAULT_WAIT` specifies the default timeout value. `NULL` is used to specify the pipe's security attributes (in this case, simply specifying that the pipe cannot be inherited by any child processes created by this process). For a full explanation of the parameters and all the options available see MSDN (2016).

3.2 Anonymous Pipes

As mentioned previously, Anonymous Pipes are intended to provide communication between parent and child processes through handle inheritance or sharing handles via another IPC mechanism such as shared memory. Bearing this in mind, within the context of this project, Anonymous Pipes only allow for communication between Java threads. In this case, a read handle to the newly created pipe is passed to a Java thread that reads the data. Two native methods within `WindowsIPC` create the pipe and read from the pipe respectively, with the read simply returning the bytes received. Ultimately, Anonymous Pipes serve as a proof of concept and their use is rather limiting since another IPC structure would be needed to distribute the pipe's read and write handles to other unrelated processes.

3.3 Mailslots

The implementation of Mailslots follows the client-server model of communication and provides synchronous, one-way communication between Java processes. It closely resembles Named Pipes in that the name of a Mailslot is specified as a file path name. A server process or thread can create the Mailslot, followed by a client process connecting and depositing a message into the newly created Mailslot. As such, the server blocks until a message has been deposited. It subsequently returns the message and simply prints it onto the terminal. The server process uses the WINAPI function `CreateMailslot` to create the slot and, in the same way as Named Pipes, uses `ReadFile` to read the message deposited. The client uses `WriteFile` to deposit data in a Mailslot. As in the case of Named Pipes, Mailslots' use of file path names can also provide an additional performance boost through avoiding an additional JNI call for I/O (see Section 4.2).

3.4 Windows Sockets

The Winsock implementation of network sockets is exceptionally complicated and presented an interesting challenge, particularly its integration with the JNI framework. It follows the same theme of client-server communication, with two native functions representing the client and server processes. To make use of Winsock, explicit code that initialises the use of `WS2_32.dll` must be used as well as code that terminates its use upon function return. It also requires various header files to utilise the Winsock APIs. Another challenge Winsock presented was its lack of use of handle values, common to most Windows system calls. It rather uses various data structures and constants that specify the required values needed to create and connect to sockets. In general, socket information is contained in a structure called `WSADATA` including Winsock version, description and the maximum number of sockets that may be opened. `WSAStartup` is a WINAPI function that is used to initialise the use of the `WS2_32.dll` file and should be called prior to initialising the socket. This needs to be done for both the server and the client process. In addition, an `addrinfo` data structure contains host address data detailing the "address family" (IPv4 or IPv6) being used as well as the socket type pertaining to the connection (see the Windows API for detailed descriptions of the various flags and parameters (MSDN, 2016)). TCP was used as the transport protocol for our purposes.

3.5 File Mapping

Windows File Mapping is used here to provide shared memory communication between two Java processes with semaphore synchronisation. A process initially writes data to a shared region by calling `createFileMapping`. This function utilises various WINAPI functions (`CreateFileMapping` and `MapViewOfFile`) to memory map a buffer containing the shared bytes. In addition, it initialises a Windows semaphore structure as a binary semaphore, thereby providing mutual exclusion functionality (see Section 3.6). The secondary Java process attempts to access the shared buffer by waiting for the semaphore to reach a *signalled* state. If signalled, the bytes are simply read (using `OpenFileMapping` and `MapViewOfFile`) and returned. If the semaphore is *non-signalled*, the process blocks until access is granted. Windows File Mapping is an easy method with which to implement IPC, albeit with the complex synchronisation APIs that Microsoft provides. It is relatively flexible in terms of the type of data that can be mapped into shared memory. The data can be in the form of plain bytes, or an actual file in the file system.

3.6 Semaphores

The semaphore mechanism is designed in such a way that it allows programmers to initially create a semaphore object, specifying a name and initial and maximum counts. Another process can retrieve a handle to the newly created semaphore object and test whether the object is in a signalled or non-signalled state. The `WindowsIPC` class contains four native method declarations that are used to implement semaphores. These methods are used for semaphore creation, retrieval, waiting and signalling. These are `createSemaphore`, `openSemaphore`, `waitForSingleObject` and `releaseSemaphore` respectively. Each method is modelled on the corresponding WINAPI call of the same name and as such takes the same arguments. The `createSemaphore` method specifies the initial and maximum count and name of the semaphore, whilst `openSemaphore` attempts to open the semaphore specified by the name passed into the method. This returns a handle to the semaphore structure. The method `waitForSingleObject` takes the opened semaphore's handle as an argument and waits for it to be in a signalled state. `WAIT_RESULT0` (which resolves to the value zero) is returned if signalled, otherwise `-1` is returned. The final method, `releaseSemaphore`, increments the semaphore's value by a specified amount.

3.7 Data Copy

Data Copy sends messages to other applications using Windows' GUI APIs. It allows an application to identify a window running in the background and dispatch a message to it using the flag

`WM_COPYDATA`. Since this project provides communication between Java programs that are not necessarily GUI-based, a *Message-Only Window* is created, which is invisible to the end-user and simply serves as a message receiver. There are two native methods that are implemented that utilise this communication technique, namely `createDataCopyWindow` and `sendDataCopyMessage`. The first method creates the message-only window that a message can be sent to and returns the message received to the Java program. The second method uses the WINAPI function `SendMessage` to send a message to an already existing message-only window. There is also a callback function (defined in `WindowsIPC.c`) called `WindowsProcedure` that handles the receipt of the message. This IPC mechanism is limited in that it only sends simple string values as messages. This is due to the "clunky" GUI-based method by which it implements communication.

4 TESTING AND RESULTS

The computer used for testing was a standard desktop machine with an Intel Core i5-6400 Quad-Core CPU running at 2.70GHz with 8GB of DDR3 RAM, running Windows 10 Enterprise 64-bit Insider Preview, build 14393. Java version 1.8.0_91 was used.

4.1 Performance Testing

To test each IPC mechanism, two separate Java programs are created that send messages to each other using an instance of the `WindowsIPC` class. The messages sent between each Java program are simple byte arrays. The array size is varied between 40, 400, 4 000 and 40 000 bytes for testing. For each IPC mechanism and message size, the sending of the message is executed ten times and the time for each run is recorded, using the standard Java `System.nanoTime()` method. The results given below are the average time for the ten runs in each case. The receiving program simply prints the message received.

Java socket communication was assessed first, and this result was used as a performance baseline. The remaining features were subsequently tested, and their performance compared to that of socket-based communication.

4.2 Results

As Table 1 and Figure 1 show clearly, using Windows' native IPC structures provides excellent performance benefits, with significantly faster communication than Java Sockets. These results clearly demonstrate the performance improvements, and are consistent with the results obtained by Wells (2009) for Unix-based systems.

For better clarity, the results are also shown as a line graph in Figure 2. This graph excludes the two sets of socket-based results, allowing for a better comparison of the performance of the native IPC facilities.

Named Pipes and Mailslots provide the best performance results and, additionally, illustrate that an additional speed-up can be obtained by using standard Java stream I/O to avoid a JNI call to

IPC Mechanism	**40 Bytes**	**400 Bytes**	**4 000 Bytes**	**40 000 Bytes**	**Mean of All Msg Sizes**	**% of Sockets**
File Mapping	172365.8	199438.3	197965.8	227152.9	199230.7	3.6
Mailslots †	44441.3	35115.1	38286.6	67247.2	46272.6	0.8
Mailslots *	18274.9	19407.8	19294.4	42893.2	24967.6	0.5
Named Pipes †	49010.1	43686.1	59091.6	98246.4	62508.6	1.1
Named Pipes *	16613.4	28922.6	50482.5	94130.8	47537.3	0.9
Anonymous Pipes	33264.8	33869.0	35530.6	70078.7	43185.8	0.8
Winsock	1459463.8	1510701.8	1563827.5	1581158.1	1528787.8	27.8
Data Copy	440602.6	466449.6	595170.2	652940.5	538790.7	9.8
Java Sockets	5162099.4	5301728.7	5658769.6	5863569.4	5496541.8	100.0

Key: † using JNI calls for I/O; * using standard Java stream I/O.

Table 1: Windows IPC Performance (all times in ns)

write the data. To expand on the latter point, for Named Pipes, the average performance across all message sizes is improved by 24% when standard Java I/O is used for communication, rather than JNI. Furthermore, the improvement for 40-byte messages is 66%, but for 40 000-byte messages is only 4%. This clearly indicates that the cost of using JNI for I/O decreases as the message size increases, but is still significant.

Making use of standard Java stream I/O to write data is even more beneficial when used in conjunction with Mailslots. It is 46% faster to use this mechanism than to use a client that connects to the Mailslot via a call to native code. On average, Mailslots provide a faster solution than both piping mechanisms that Windows offers, with the added benefit of a significantly easier implementation.

The performance of Winsock is particularly surprising for a network socket-based communication mechanism, with an average message-sending time of 1 528 787.8ns (1 528.8μs) across message sizes, which is approximately five times faster than that of Java Sockets despite the JNI call. This is presumably due to optimisations made by Microsoft in their APIs. Despite the fact that Winsock comprehensively outperforms Java Sockets, it is still relatively slow in comparison to the other IPC features tested.

Interestingly, File Mapping (i.e. shared memory) does not offer the best performance in comparison to the other Windows IPC mechanisms, which is surprising for a shared memory communication mechanism. This could be due to additional overheads induced by the semaphore set-up when initialising the shared data as well as opening the semaphore when a process attempts to read the data. The JNI calls also contribute to the overheads created by these operations. Ultimately, it is a good mechanism that allows synchronised access to shared data, but with additional overheads in comparison to the piping mechanisms and Mailslots.

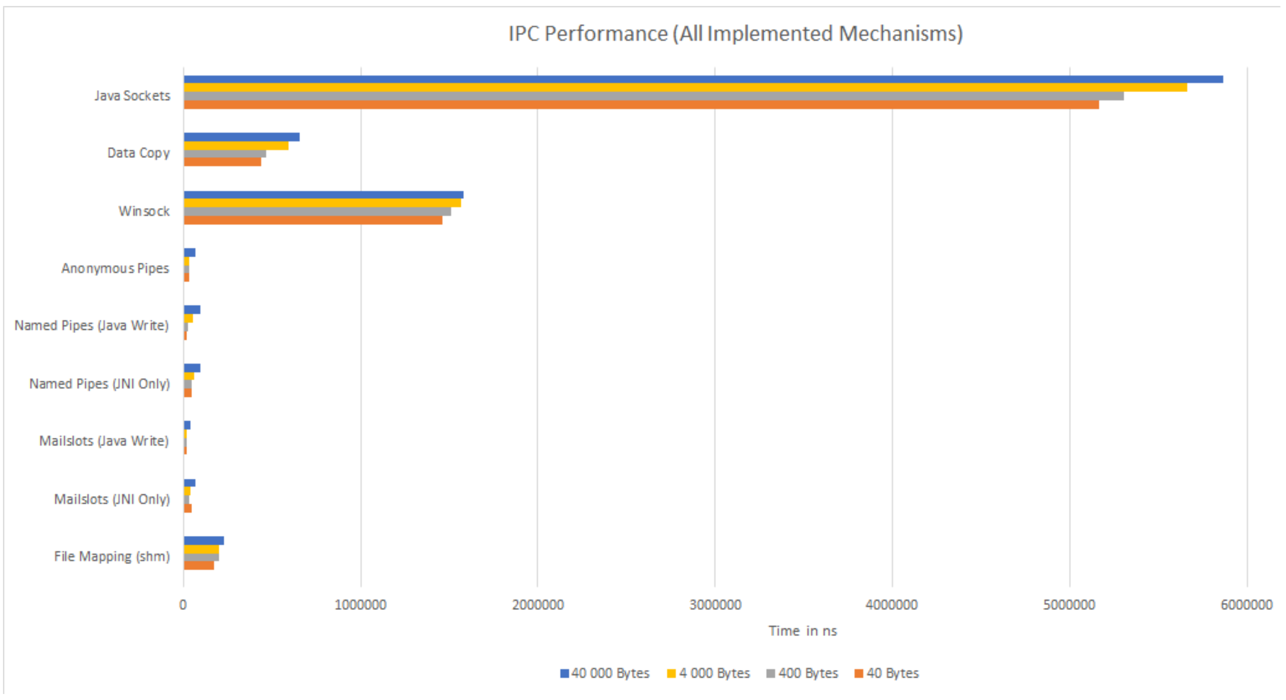

Figure 1: Performance of All IPC Mechanisms Implemented

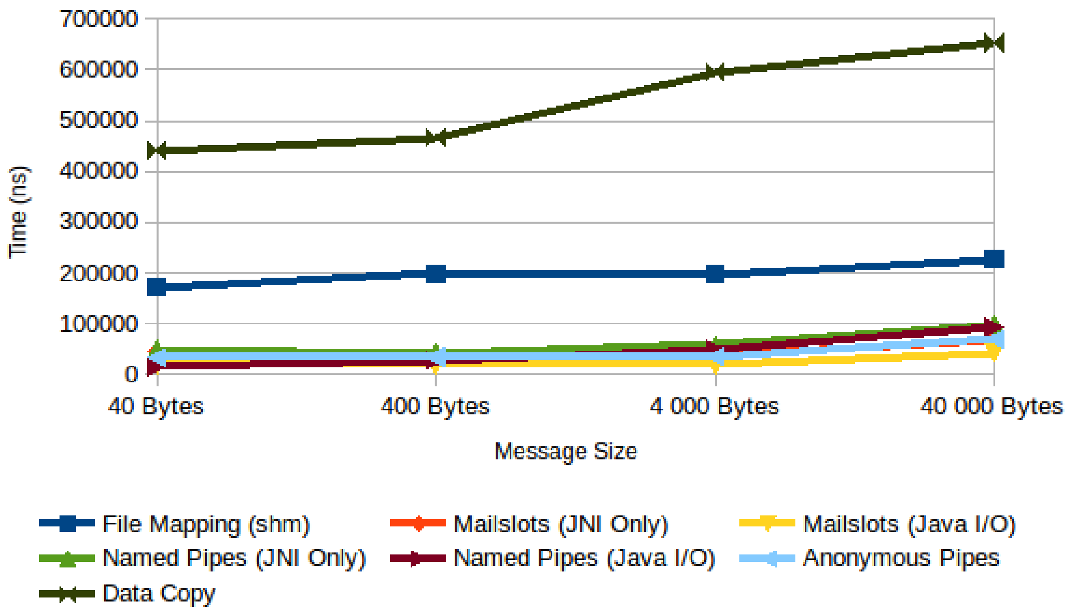

Figure 2: Performance of IPC Mechanisms for Differing Message Sizes

Data Copy provides relatively poor performance in comparison to the other non-socket communication methods presented in this chapter, with an average message sending time of 538 790.7ns (or 538.8μs). Furthermore, when considering the complex implementation experienced and the awkward window-based message sending mechanism, it is not a practical communication mechanism. It is also an IPC structure that is explicitly Windows-based and as such does not contribute to a generic IPC design.

5 DISCUSSION

The results obtained confirm the poor performance of socket-based interprocess communication for the Java programming language. Significant overhead is introduced due to messages traversing the protocol stack. The average message-send time yielded by Java Sockets (5 496 541.8ns) proves these statements, particularly when compared to the other mechanisms implemented in the `WindowsIPC` class. As a result, making use of native IPC structures provides considerably better performance; even utilising Windows' socket communication mechanism (Winsock) proved more efficient, with its message-sending time approximately 27% of the total Java Sockets message-sending time (on average for all byte sizes). Though Winsock provides performance gains, other native communication mechanisms resulted in excellent communication performance, which agrees with the results obtained in previous studies (Wells, 2009; Wells & Anderson, 2013). Mailslots and Named Pipes proved to be the most interesting IPC mechanisms with impressive performance results, both being considerably faster than Java Sockets and Winsock. These mechanisms are unique in that their performance is consistently impressive even with the cost of calling JNI code. The added benefit of writing data using Java I/O streams contributes to their usability and provides even better performance due to avoiding an additional JNI call. These two native Windows IPC mechanisms appear to be the most viable communication mechanisms for the Java programming language if performance is prioritised, and are recommended. Their implementation also proved to be relatively simple in comparison to the other IPC structures.

In general, native Windows code offers much better communication performance, but its drawback is that the Win32 APIs are extremely complex. This is starkly obvious when comparing the equivalent system calls for the Unix System V IPC mechanisms, which are generally much simpler.

5.1 Generic Considerations

With prior work that has demonstrated the performance benefits of providing access to system-level IPC mechanisms in Linux and Solaris (Wells, 2009), and having confirmed similar benefits in Windows in this study, it is useful to consider how a generic solution might be provided that works across multiple operating system families. The first step would be to eliminate mechanisms not common to Windows, Mac OS X, and Unix derivatives. From the review conducted, shared memory is common to all of these widely-used operating systems, and provides a reasonably efficient method with which to implement IPC (as illustrated by SMOF mentioned in Section 2, LinuxIPC and WindowsIPC). A generic version could be developed that utilises these concepts and suitably abstracts away operating system dependencies.

Named Pipes is an additional option, as Mac OS X supports the `mkfifo` system call which creates a Named Pipe (or FIFO file). This is identical to the Linux `mkfifo` system call. Since Windows' Named Pipes also create a named entity in the file system, a suitable abstraction could be developed that integrates these calls. The challenge would be to determine a method with which to simplify the Windows' `CreateNamedPipe` system call and its numerous parameters. In addition, Unix and Windows pipe naming conventions differ, with Linux/Mac allowing a pipe name anywhere in the conventional file system. Windows requires the keyword "Pipe" to prefix the pipe name (e.g. `\\.\Pipe\MyPipeName`). File pathname separators (slashes or backslashes) and escaping also differ, which would require careful consideration.

In terms of synchronisation, a counting semaphore would be the best solution since all three widely-used operating systems support this feature. Again, a suitable method would need to be found to abstract the complexities present in Windows' semaphore structure. If a synchronisation mechanism requires ownership of the protected resource, then a mutex (supported by Windows, Linux and Mac) can be developed. Ultimately a JNI wrapper design could be implemented that attempts to combine these mechanisms into a single call. Since native code would be used, a recompile would be necessary on the target machine.

Linux and Mac OS X are very similar in terms of their internal structures. This is due to their common Unix foundation and background. The challenge that arises, is the significantly different mechanisms that are employed by Windows. This is shown in the complexity of the APIs utilised for this project's implementation, which are more complicated than that of Linux and Mac and are substantially more difficult to learn. Ultimately the biggest challenge is to determine a method with which to abstract away Windows' complexity into a mechanism that is similar to that of the Unix world. Therefore it is important that a technology such as the Window Subsystem for Linux is investigated as this could prove to be an appropriate starting point for a generic IPC design (Hammons, 2016). Together WindowsIPC and LinuxIPC provide a good starting place from which to embark on the design and implementation of a generic, portable IPC library for Java.

6 CONCLUSION AND FUTURE WORK

This research has demonstrated the inefficiencies of socket communication for interprocess communication in Java, and has provided an alternative that utilises the native IPC facilities of the Microsoft Windows operating system. It has presented substantial performance gains, with communication times considerably faster than sockets. In addition, it has confirmed the use of shared memory and pipe communication as the IPC methods of choice, particularly when considering a generic IPC design.

This project has also opened further research avenues, specifically with regard to the design of a generic IPC library, and an investigation into the capabilities and limitations of the Windows Subsystem for Linux (Hammons, 2016), and it can be said that a valuable contribution has been made to the field of Java concurrency.

However, limitations do exist with the current library. WindowsIPC communication is all synchronous in nature. This means that communication is limited to a blocking server program that awaits a client connection that sends a message. This is not ideal, and should be extended if a widely-applicable library is to be provided. In addition, not all Windows interprocess communication mechanisms have been implemented (e.g. synchronisation techniques such as mutexes), although justifications are provided where omissions exist. This means that their potential (however unlikely) performance benefits are yet to be determined. This project has also omitted the use of other technologies and libraries such as the Windows Subsystem for Linux, and Boost.Interprocess (Gaztanaga, 2012). Hence future work could be done on designing and implementing an extended WindowsIPC library with asynchronous communication and additional IPC mechanisms.

The ultimate goal of our research is to provide a generic and efficient IPC library that is operating system independent, and supports Java's property of executable program portability.

References

Brosgol, B. (1998). A comparison of the concurrency features of Ada 95 and Java. In *Proceedings of the 1998 Annual ACM SIGAda International Conference on Ada* (pp. 175–192). SIGAda '98. Washington, D.C., USA: ACM. https://doi.org/10.1145/289524.289572

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].

Garg, V. (2005). *Concurrent and distributed computing in Java*. Hoboken, New Jersey: John Wiley & Sons.

Gaztanaga, I. (2012). Boost.Interprocess. http://www.boost.org/doc/libs/1_53_0/doc/html/interprocess.html. [Date Accessed: 20 September 2016].

Gelernter, D. (1985, January). Generative communication in Linda. *ACM Trans. Programming Languages and Systems*, *7*(1), 80–112. https://doi.org/10.1145/2363.2433

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). *Java concurrency in practice*. Massachusetts: Addison-Wesley.

Hammons, J. (2016). Windows Subsystem for Linux Overview. https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/. [Date Accessed: 19 September 2016].

Hayes, B. (2007). Computing science: Computing in a parallel universe. *American Scientist*, *95*(6), 476–480.

Hoare, C. (1974, October). Monitors: An operating system structuring concept. *Commun. ACM*, *17*(10), 549–557. https://doi.org/10.1145/355620.361161

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

Lea, D. (1999). *Concurrent programming in Java: Design principles and patterns* (2nd). Reading, Massachusetts: 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. https://doi.org/10.1016/j.scico.2005.03.007

Lewandowski, S. (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.

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

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

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

Smith, D. (2016). *Interprocess communication with Java in a Microsoft Windows environment — heading towards a generic IPC design*. Rhodes University. Dept. of Computer Science.

Taboada, G., Ramos, S., Expósito, R., Touriño, J., & Doallo, R. (2013, May). Java in the high performance computing arena: Research, practice and experience. *Sci. Comput. Program. 78*(5), 425–444. https://doi.org/10.1016/j.scico.2011.06.002

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

Waldo, J. (1998, July). Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, *6*(3), 5–7. https://doi.org/10.1109/4434.708248

Wells, G. (2009, July). Interprocess communication in Java. In H. Arabnia (Ed.), *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09)* (pp. 407–413). Las Vegas: CSREA Press.

Wells, G. (2010). Extending Java's communication mechanisms for multicore processors. In *8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*. (Poster). Vienna, Austria.

Wells, G. & Anderson, M. (2013). *Efficient interprocess communication in Java*. Dept. of Computer Science, Rhodes University.