Enhancing Inter-Process Communication in Java

Submitted in partial fulfilment

of the requirements for the degree of

Bachelor of Science Honours

of Rhodes University

Timothy Fischer

Grahamstown, South Africa

October 18, 2024

Declaration of Authorship

I, Timothy Ronald Francois Fischer, declare that Enhancing Inter-Process Communication in Java is my own work. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Rhodes University.
- This research report has not been submitted, in whole or in part, for any degree in any other university.
- Where I have consulted the published work of others, this is always clearly attributed.
- I have indicated by reference and acknowledgment the areas that are not my own work. Except for such quotations, this research report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the research report is based on work done by myself jointly with others, I have clarified what others did and what I have contributed to myself.

Abstract

The rise of computationally intensive applications increases the need for parallel programming techniques that effectively utilize multi-process architectures. However, effectively utilizing multiple processes in an application can be complex and requires careful consideration of factors such as dependencies, overhead, and synchronization. The Java programming language has very little support for accessing multiple processes in a single Java Application. Implementing more robust inter-process communication mechanisms can allow Java applications to effectively utilize multi-process architectures.

This study investigates how using more advanced input and output tools to improve upon existing inter-process communication techniques for the Java language. The java.nio library offers many performance-enhancing techniques for input and output operations compared to the standard java.io library. Four inter-process communication mechanisms were implemented: Named pipes, message queues, shared memory, and sockets.

The newly implemented input and output mechanisms showed significantly improved performance across all IPC mechanisms. The study's results showed the new implementation operating with latencies as little as 12.0% of the previous implementations recorded latency.

ACM Computing Classification System

Thesis classification under the ACM Computing Classification System[1] (2012 version valid through 2024):

- **Computing methodologies** ∼ **Concurrent computing methodologies** ∼ **Concurrent algorithms**

General-Terms: Inter-Process Communication, Message Passing, Java, Systems V IPC, JNI, java.io, java.nio

[1] https://www.acm.org/publications/class-2012

Acknowledgements

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA and Rhodes University. The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

I would also like to acknowledge all those who directly and indirectly supported me through my work. In particular, I would like to extend acknowledgments to my supervisor, Professor George Wells, who has guided me throughout my time conducting this study.

Contents

List of Figures

Listings

IPC	Inter-Process Communication
IO	Input Output
FIFO	First in First out
JNI	Java Native Interface
JVM	Java Virtual Machine
JDK	Java Development Kit
NIO	New Input Output

1

Introduction

1.1 Context of Research

The rise of computationally intensive applications, particularly in science and engineering fields, has pushed the boundaries of what traditional computers can handle (Ciccozzi *et al.*, 2022). Therefore, modern software systems rely heavily on parallel programming techniques for high performance and scalability. However, effectively implementing these techniques can be very complex and challenging(Belikov *et al.*, 2013). Implementing parallel applications requires careful consideration of factors like data dependencies, communication overhead, and synchronization. Therefore, inter-process communication (IPC) mechanisms are vital, enabling processes to exchange data, synchronize activities, and share resources effectively and efficiently. However, Java is very limited regarding IPC (Wells, 2009). This is because Java focuses more on multithreaded parallelism and distributed computing rather than multi-process parallelism. Java developers often have to resort to external libraries or platform-specific tools to implement IPC effectively for Java applications.

This research focuses on building on previous works by Wells (2009), who developed a Library called Linux IPC. This library provides a set of tools for IPC between multiple processes in a single machine. This research aims to improve this library by implementing more efficient input and output mechanisms, focusing on reducing latency. Wells used the standard Java input and output library (java.io), which provides a simple and lightweight mechanism for controlling input and output. However, the Java JDK offers other tools, such as New Input Output (NIO), which gives more fine-grained control over IO at the cost of added complexity. Utilizing java.nio could lead to more efficient IPC mechanisms that perform with reduced latency.

1.2 Motivation

Wells (2009) observed a gap in how inter-process communication was implemented in Java compared to other programming languages. Java has excellent support for multi-threaded parallelism. However, there was a lack of support for effectively utilizing multiple processes. The only way to send messages between processes was by using mechanisms developed for distributed systems. Passing messages between processes can be done using sockets in a "loopback" style network connection. This method requires messages to traverse the entire network protocol stack twice, adding much overhead.

Wells implemented three IPC mechanisms into a library called Linux IPC. The three mechanisms were pipes, messaging queues, and shared memory. Semaphores are added as a synchronization technique for the shared memory mechanism. The implementation utilized the messaging queues, shared memory, and semaphore mechanisms from System V IPC, a native library found in UNIX systems. On the Java side, the standard java.io library was used for IO to the underlying IPC mechanisms. Some results from testing this implementation can be seen in Figure 1.1 below, which shows that sockets had the highest latency, performing worse than all other IPC mechanisms implemented by Wells.

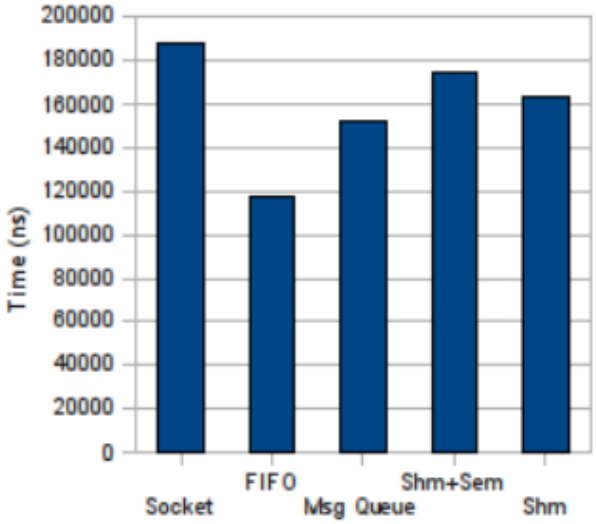

Figure 1.1: Simple Results, Wells (2009)

To further motivate this point, the results from a study conducted by Venkataraman and Jagadeesha (2015) can be seen below in Figure 1.2. This study compared the differences in latency and throughput for common IPC mechanisms. The tested mechanisms were sockets, pipes, and shared memory. Once again, sockets were the worst-performing mechanism of the three, showing higher latency and lower throughput for message passing.

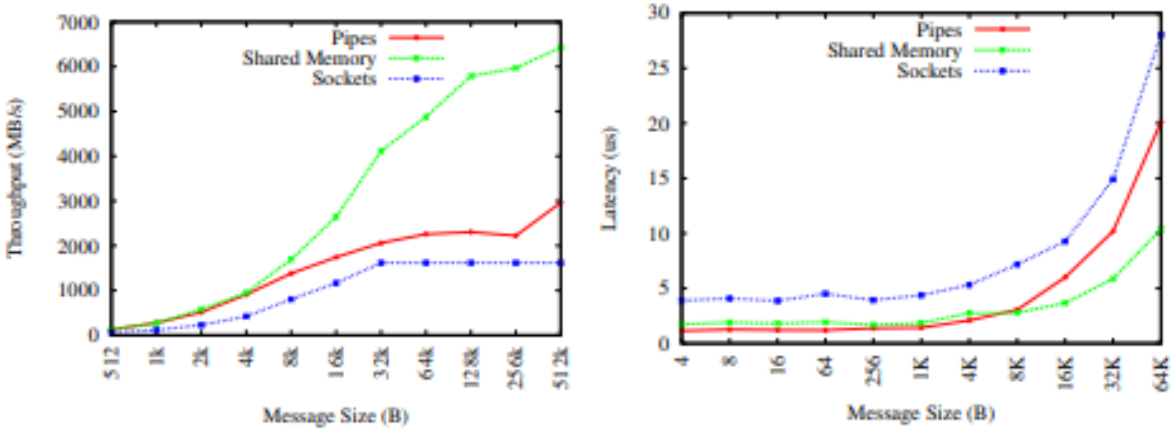

Figure 1.2: Results, Venkataraman and Jagadeesha (2015)

This further motivates that there is a need for robust and efficient IPC libraries like Linux IPC developed by WElls. However, some improvements can be made.

1.3 Research Statement

Linux IPC relies on the standard java.io library for input and output operations. While java.io provides a solid foundation for IO operations, java.nio (New Input Output) offers alternative IO mechanisms. This study explores the potential benefits of using java.nio to develop more efficient inter-process communication mechanisms for Java applications running in a single UNIX environment. The research will investigate the performance advantages of the tools provided by java.nio, such as buffers and channels. By evaluating these tools, this study aims to establish how java.nio can contribute to the development of more efficient inter-process communication mechanisms for Java applications.

1.4 Research Questions

- Is there a significant performance difference between java.io and java.nio for IO operations?
- Will utilizing tools provided by java.nio instead of java.io improve the performance of message-passing libraries such as Linux IPC?

1.5 Research Objectives

- Investigate the performance differences between java.nio and java.io.
- Reimplement Linux IPC using java.nio for the IO operations, focusing on reducing latency.

1.6 Limitations

Wells (2009) implemented Linux IPC using System V IPC mechanisms for inter-process communication. Since this research focuses on Java-based IO to access these underlying IPC mechanisms, The proposed improvements in this study must also utilize System V IPC, thereby also limiting this implementation to UNIX systems.

1.7 Approach

The first step of this study involves conducting a comprehensive literature review to gain an understanding of how message-passing libraries operate and how each component of Linux IPC functions so that improvements can be made. The differences between java.nio and java.io, the mechanisms they provide, and their performance characteristics will also be reviewed.

Next, the design and implementation of the proposed changes to Linux IPC will begin. The insights and observations gained during the literature review will be used to implement new IO operations to access underlying Unix IPC mechanisms.

Once the implementation is complete, Experiments will be conducted on the new and previous implementations to assess and compare their performance. These experiments will focus on computing round-trip times. This study will end with a discussion of the results and insights gained from the experiments.

1.8 Thesis Outline

The remainder of this report reads as follows:

Chapter 2: *Concepts and Literature Review:* This chapter explores the concepts relating to inter-process communication and tools used in the study by Wells (2009). Other studies are also analyzed and discussed to find insights related to these concepts.

Chapter 3: *Methodology:* This chapter defines the changes proposed to reimplement the IO operations of Linux IPC and the motivation for these changes. The requirements of the experiments needed to assess these changes are also defined in this chapter.

Chapter 4: *Implementation:* This chapter gives a detailed description of changes made to Linux IPC. These changes are compiled into a new library. A detailed comparison between the two implementations is discussed. A detailed implementation of the experiments used to assess this implementation is also discussed in this chapter.

Chapter 5: *Results:* This chapter reports on the results obtained from the experiments conducted in this study.

Chapter 6: *Conclusion and Future Work:* This chapter summarises the work done in this study and discusses if and how the research objectives were achieved. Any avenues for future work or improvement are also discussed.

2

Concepts and Literature Review

This chapter explores the concepts relating to inter-process communication and tools used in the study by Wells (2009). Other studies are also analyzed and discussed to find insights related to these concepts.

2.1 Inter-process Communication

Inter-process communication (IPC) refers to mechanisms and tools used to send data from one process to another (Venkataraman and Jagadeesha, 2015). IPC mechanisms can be categorized broadly based on four main criteria:

- Communication Isolation: If communication is restricted to related processes.
- Data Access Mode: If the process can read, write, or create the data.
- Number of processes: If the data is passed between more than two processes.
- Synchronisation: If the communication is blocking or non-blocking (synchronous or asynchronous)

2.2 IPC Mechanisms

This research focuses on implementing new IO operations for the IPC mechanisms implemented in Linux IPC. These IPC mechanisms are Named pipes, Message queues, and Shared memory.

A named pipe, also called a FIFO, provides a synchronous, unidirectional communication channel between two processes (Mutia, 2014). Communication using named pipes works on a First-In, First-Out basis, ensuring that data is read in the order it was written (Wells, 2009). When using named pipes, a file is created in the machine's filing system and is used for reading and writing, but no actual disk IO occurs. However, named pipes offer limited functionality compared to message queues or shared memory (Mutia, 2014). They are mostly suited for simple, direct communication where more complex control over message passing is not required.

A message queue functions similarly to a pipe but allows processes to send and receive messages asynchronously (Mutia, 2014). Messages are stored in the queue until the receiving process retrieves them. There are two components for each element of a message queue: the message and the message type. The message type allows processes to retrieve messages based on a particular message type.

Shared memory allows multiple processes to access the same memory segment (Mutia, 2014). A process can map the shared memory segment into its address space, giving direct access to this memory segment. This means there is no data copying between processes, as both processes will access the same shared memory segment. This is useful when larger messages need to be shared. Since multiple processes access the same memory segment, synchronization is needed (Wells, 2009). Synchronization is necessary to prevent race conditions that may cause data inconsistencies.

2.3 System V IPC

System V Inter-Process Communication (IPC) introduces a set of communication mechanisms introduced into UNIX systems to overcome the limitations of traditional pipes and signals (Fleisch, 1986). It provides three primary communication and synchronization mechanisms: message queues, shared memory, and semaphores (Wells, 2009).

2.4 JNI

To implement the underlying IPC tools using libraries such as System V IPC in a Java application, some interface is required (Wells, 2009). The Java Native Interface (JNI) allows Java methods to call or be called by native methods written in C/C++. This means that through some IO operations, Java applications can access underlying operating system functions and external libraries present in native programming languages (Grichi *et al.*, 2019). The JVM hands over control to the native code when a native method is called through JNI (Lee, 2019). This means that context switching is involved during the execution of different environments. Figure 2.1 below shows where JNI operates between Java and native applications.

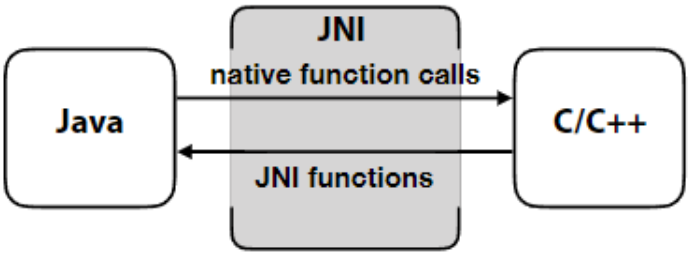

Figure 2.1: Overview of Java and C interoperation, Lee (2019)

Using JNI can be complex as it requires careful handling of data type conversion between Java and native code (Wells, 2009). JNI can convert primitive data types from native code to Java code directly (Grichi *et al.*, 2019). However, calling JNI methods to do this does introduce some overhead. Unfortunately, not all types can be converted between Java and native code, and extra overhead can be introduced when unknown object types need to be converted.

2.5 Input and Output in Java

The overall goal of this study is to implement IPC tools for Java applications. To do this, input and output operations are required for Java applications to pass messages into the underlying Linux IPC mechanisms. In programming, input/output refers to an interface between two endpoints (Travis, 2003).

The Java JDK offers two main libraries that handle input and output. They are the standard java.io library, which was introduced in JDK 1.0, and the java.nio library, which was initially introduced in JDK 1.4. The following sections will explore the workings java.io and java.nio.

2.5.1 java.io (Standard IO)

Standard java.io is split into two components: byte-oriented IO and text-oriented IO (Dickens and Thakur, 2000). Byte-oriented IO mainly focuses on using streams. The input stream and output stream classes provide the foundation for byte-oriented IO.

Streams represent an ordered sequence transferred from source to destination. Streams transfer data sequentially, processing one byte or character at a time. The Input Stream class is used to read data from some source. The output stream is used to write data to some destination. These classes proved useful functions for manipulating bytes of data for IO purposes. The input and output stream classes contain various methods for transferring bytes of data. The output stream class contains writing methods, while the input stream class contains reading methods. These methods can process a single byte of data or a specific portion of a byte array. These input and output stream classes are abstract and can be further extended to provide additional functionality. These abstract stream classes are never used directly.

There are a handful of subclasses available that extend these stream classes, such as file streams and buffered streams. These subclasses all provide specific functionalities

for different scenarios. File streams provide mechanisms useful for interacting with files. buffered streams provide an intermediary buffer that can store data during processing.

2.5.2 java.nio (New IO)

There are three core components offered by java.nio, namely buffers, channels, and selectors (Artho *et al.*, 2013).

The Buffer is the most important component in java.nio (Travis, 2003). Buffers are objects that act as containers and function similarly to an array. There are two methods to create a buffer. The first method is done by allocating a new empty buffer. The second method uses the wrap() function to create a buffer out of an already existing array.

Buffers have three state variables: position, limit, and capacity. These three attributes can be used to track the state of a buffer and the data within. The position attribute keeps track of how much data is in the buffer. In terms of an array, it keeps the index of the last byte in the buffer. The limit attribute keeps track of how much free space there is within the buffer. The capacity attribute states the maximum size of the buffer.

Buffers are accessed in almost every IO operation, usually by using channels. Data can be passed through channels to and from buffers. The "get()" and "put()" methods can also be used to access and store data in buffers directly. Both of these methods take in positional arguments and can access and store data at a specified position within a buffer.

The methods "flip()" and "clear()" are used to manipulate the position and limit attributes for effective buffer usage. The "clear()" method resets a buffer position and limits attributes to zero, effectively emptying the buffer contents. However, the "flip()" method sets the limit attribute to the value of the position attribute and then sets the position attribute to zero. This is useful when data has just been stored in the buffer and now needs to be sent out of the buffer.

Buffers can optionally be allocated as a direct buffer. Direct buffers are a performance optimization technique designed to enhance the efficiency of buffers (Pugh and Spacco,

2004). In contrast to the standard java.io approach, which relies on byte arrays stored in the JVM memory heap and managed by the Garbage Collector, direct buffers allocate memory directly in the system's native memory heap. This approach eliminates the need for the JVM to copy data between its memory heap and the native heap, thus reducing data copying when executing operations.

A channel is similar to a stream in terms of reading and writing data (Travis, 2003). However, channels are bi-directional, meaning they can be used to both read and write data. There are a few special channels such as file channels and socket channels for handling more specific operations.

Selectors allow for multiplexing when using channels for network-specific applications (Artho *et al.*, 2013). This is particularly useful when implementing systems with large amounts of concurrent IO operations. Selectors can query and keep track of multiple socket channels at the same time. Each socket channel has a unique identifying key that the selector uses to track and query for data.

2.6 Related Studies

A Study by Pugh and Spacco (2004) developed MPJava, a pure-Java message-passing framework. In the implementation of MPJava, they use java.nio for input and output for Java applications. They avoid using the JNI and Native code due to some downsides, such as performance penalties and limitations on JVM. However, other message-passing frameworks generally rely on accessing native code through JNI as a performance optimization technique. Therefore, the overall goal of this study is to implement message-passing mechanisms that can perform as well as these native code implementations but without the reliance on native code. Since MPJava focuses on implementing message passing for distributed systems, it must rely on networking tools such as socket channels and selectors. There are, however, some insights that are useful for non-distributed systems. MPJava uses direct byte buffers to allocate memory directly in the native memory heap rather than the JVM memory heap. This avoids the overhead of garbage collection and reduces

the need for data copying between Java and native memory. Direct buffers can be accessed directly in system-level input and output operations to improve performance. Using direct buffers also removes the need to copy data between the JVM memory heap and the native memory heap. This reduces overhead and improves performance by minimizing data copying. MPJava also pre-allocates the buffers to avoid the cost associated with frequent memory allocation and garbage collection. MPJava performed similarly to other systems that used native libraries. It particularly excelled at handling larger message sizes. The results highlight significant overhead in standard java.io implementations due to data conversions and memory overhead, whereas java.nio reduced this overhead.

A study by Baker *et al.* (2006b) aimed to implement a messaging-passing library similar to MPJava called MPJ Express. This implementation focuses on implementing MPI-oriented messages passing through a mix of native tools and java.nio. This is in contrast to the previous study, which was purely implemented in Java. There were two core components of this study: the input-output component, which was implemented using java.nio, and the native component, which utilize the Myrinet Express library, a native code interface geared towards high-performance networking. One of the key issues addressed in the paper is the overhead involved with utilizing JNI and native code, particularly the need to copy data between the JVM and the operating system. To reduce this, MPJ Express implemented direct byte buffers similar to the previous study. The combination of Myrinet Express, and JNI allowed networking tools to interact with direct buffers, Allowing for more efficient message passing. Data can be transferred directly between the network and the application memory without intermediate data copying.

Another study by Baker *et al.* (2006a) set out to discover a buffer management strategy that optimizes communication for Java messaging passing systems like their previous work, MPJ Express. A key part of the study was showing the benefits of using direct byte buffers in Java. The decision to use java.nio direct buffers were motivated by the need to avoid consistently allocating and deallocating buffers for every messaging passing operation. This constant creation and destruction of buffers introduces much unnecessary overhead, especially where many smaller messages are passed. A few methods for pooling buffers were highlighted in this study. The first approach uses a list of several pre-allocated

buffers of varying sizes, all within a single memory region. When a buffer is needed, the system finds a free buffer in the list that best fits the required size. When a buffer of the requested size is not available, the algorithm finds a larger block and splits it into two smaller blocks. These split buffers are tracked so that they can be joined again after each has become free. The second approach functions similarly to the first but creates multiple lists of buffers within different memory regions. While the study found that the first approach generally outperforms the second in terms of speed and memory footprint, the second approach offers a structure that could be beneficial in cases where the memory needs to be compartmentalized. Both strategies avoid frequent creation and destruction of buffers. By pooling buffers and reusing them, the system reduces the overhead associated with buffer allocation. This overhead would be worsened if the buffers were allocated within the JVM memory heap due to garbage collection.

A study by Aamir and Jawad (2009) aimed to implement an addition to the MPJ Express Java messaging system discussed previously. The main goal of this study was to optimize communication between processes on the same machine by utilizing shared memory. MPJ Express relies on network-based message passing for processes in a single node of the distributed system. Therefore to pass a message from one process to another on the same machine, messages need to traverse the entire network protocol stack twice. This adds unnecessary overhead. To implement the new shared memory addition, they utilized the shared memory mechanisms offered by System V IPC. It avoids network-based overheads and allows for direct communication through shared memory segments. This, unfortunately, introduces portability issues as System V IPC is a UNIX-specific library. There is also some additional overhead due to the need for JNI in order to call these native methods. This is due to context switching between Java and native code, which leads to performance penalties, especially for small messages. However, JNI is a necessary component for accessing low-level system libraries like System V IPC. They tested this implementation against mpiJava, a standard library used for parallel programming on distributed systems. It utilizes a shared memory device that is highly optimized for single-system communication, making mpiJava particularly efficient when communicating between processes on the same physical machine. The newly implemented shared mem-

ory mechanism for MPJ Express offered good performance improvements and performed better than mpiJava, where larger messages were concerned. However, it suffers from the overhead introduced by JNI. Therefore, for smaller messages, the mpiJava library performs better than this implementation due to this JNI overhead. One key observation is the need to minimize JNI calls to reduce the overhead associated with JNI and its impact on performance.

A study by Ramos *et al.* (2013) aimed to implement more scalable and efficient communication middleware for multi-core systems. More specifically, to implement more efficient IPC for each node within a distributed system that utilizes multiple processes. They implemented a custom shared memory queue mechanism. However, this was purely implemented in Java and runs completely within the JVM. Because of this, they used java.nio for efficient IO mechanisms to implement this shared memory queue. This queue uses a zero-copy mechanism for message passing. In previous studies, messages are stored in direct buffers before being passed to IPC mechanisms. In this study, a reference to the buffer is passed instead. This means that the message is not being passed through the IPC mechanism and greatly reduces the latency of passing data. When a receiving process wants to retrieve a message, the reference to the sending processes buffer is retrieved instead. The receiving process can then copy the message directly into its own buffer from the sending process buffer. This reduced the amount of copying required to pass messages. However, this does raise synchronization issues. Processes must coordinate to ensure that messages are not manipulated while copying data from another process's buffer. This zero-copy mechanism is particularly effective for larger message sizes and significantly reduces the overhead associated with message passing.

A study conducted by Grichi *et al.* (2019) aimed to establish and promote effective practices for using JNI by identifying and analyzing real-world examples from 100 systems. The researchers aim to catalog practices that can help developers overcome the limitations of JNI, particularly in areas critical to the stability and security of JNI-based systems. The ultimate goal is to create a comprehensive set of guidelines that can serve as a reference for developers using JNI. One of the main points highlighted in this study is that calling Java code from native code causes significant overhead. This is due to the context

switching between Java and native environments. To mitigate this issue, the study recommends minimizing the number of JNI calls as much as possible. Another point made by the study is that additional overhead is introduced when dealing with complex data types such as custom Java objects. These objects cannot simply be converted to native code objects, and therefore some additional processing is required. The study recommends using primitive data types as much as possible since these can be directly mapped between Java and native languages. Another key point made by the study is that accessing Java arrays from native code introduces overhead. This increased overhead arises since the JVM may need to create copies of the array and perform type conversions so that they are compatible with the native code environment. This is similar to the point made previously. In order to avoid this, the study recommends using direct byte buffers from java.nio. Since the direct buffers are allocated in native memory, there is no need for type conversion or data copying. The native code and the Java application can both directly interact with the buffer allocated in native memory. This solves the issue of using Java arrays and significantly reduces overhead.

2.7 Discussion

Based on the review of the studies above, there are a few notable observations:

- **Observation One:**

  The performance benefits of direct buffers: The performance benefits of direct buffers are consistently highlighted across many studies above. This is because direct buffers allocate memory directly to the native memory heap and outside of the JVM memory heap. Usually, if some operation requires data from memory allocated in the JVM, this data must first be copied to native memory. With direct buffers, this is no longer required and reduces the amount of data copying required to perform operations. Directly allocated buffers allow native code to interact with and manipulate these buffers. Therefore, there is less data copying during IO operations.

- **Observation Two:**

  Use buffers efficiently: Buffer pooling strategies used by Baker *et al.* (2006a) help to optimize memory management and reduce overhead from frequent allocation and deallocation of buffers. This is particularly useful when processes pass many smaller messages, as this would introduce a lot of overhead. Systems like MPJava by Pugh and Spacco (2004) also pre-allocate buffers so that they are reusable, minimizing the need to create and destroy buffers. Therefore, pre-allocating and reusing buffers is important for optimizing performance.

- **Observation Three:**

  Minimise data copies: Messages must be copied multiple times when passed from one process to another. Tools like direct buffers and careful buffer management can reduce the amount of data copies needed. Studies by Ramos *et al.* (2013) both implement a zero-copy mechanism where a reference to the buffer with some message is stored in the IPC mechanism rather than copying the message. This means messages do not need to be directly copied into the IPC mechanism. Strategies such as this greatly reduce overhead when passing messages by reducing the number of data copies.

- **Observation Four:**

  Minimise JNI calls: Studies by Pugh and Spacco (2004) and Ramos *et al.* (2013) avoid using native libraries due to the overhead associated with calling native code through JNI. This is due to the points made by Grichi *et al.* (2019) and Aamir and Jawad (2009), where context switching by JNI calls and complex data type conversion all add significant overhead when calling native methods. This overhead is, unfortunately, unavoidable. The only way to mitigate this is by minimizing the amount of native function calls from Java applications. This can be achieved by performing multiple operations within a single native method so that fewer native methods need to be called, leading to fewer JNI calls.

2.8 Summary

This chapter explored some of the approaches taken in implementing message-passing systems and related concepts. A few key observations were made by analyzing these studies. One of the main observations highlighted by the related studies was the performance benefits of direct buffers. Using a direct buffer inherently reduces the amount of data copies required to pass messages in any scenario. Therefore, implementing direct buffers will be the focus for improving the Linux IPC library.

3

Methodology

This chapter defines the changes proposed to reimplement the IO operations of Linux IPC and the motivation for these changes. The requirements of the experiments needed to assess these changes are also defined in this chapter.

3.1 High level Methodology

This study aims to develop improvements to the Linux IPC library implemented by Wells (2009). Linux IPC is broken down into two main components. The first component contains the IO operations allowing Java applications access to the underlying IPC mechanisms. The second component contains native code methods. These methods relay messages from the first component to the underlying IPC mechanisms. Together, these two components give Java applications access to effectively utilize the underlying IPC mechanisms available in UNIX systems.

This study focuses on modifying the first component of Linux IPC by replacing the IO operations. In Linux IPC, these IO operations are implemented using java.io streams. This new implementation will focus on implementing direct buffers using java.nio in this component. This design choice is based on Observation One in Chapter Two since direct buffers offer many performance benefits and do not rely on the JVM memory heap for storing data. However, significant modifications must be made to the second component to effectively interact with these direct buffers. This new implementation will be compiled into a new library called Linux NIPC.

The interaction between the components of Linux NIPC can be seen in Figure 3.1 below.

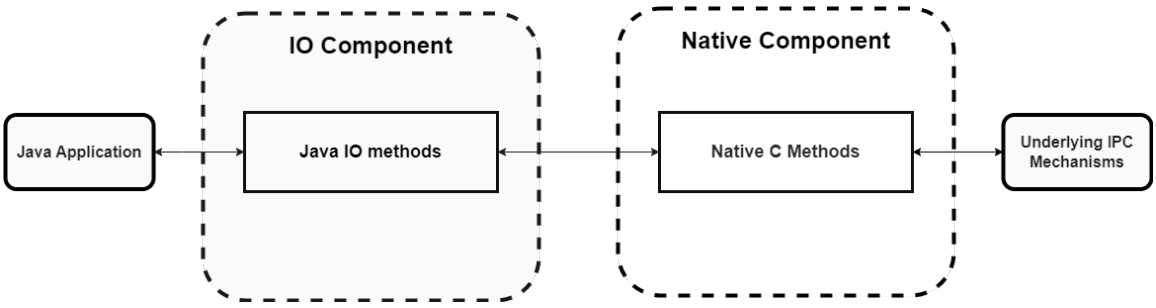

Figure 3.1: Interaction of Components

3.2 Linux NIPC design

The following sections highlight the design of each component of Linux NIPC, namely, the IO component and the native component.

3.2.1 IO Component

The IO component allows Java applications access to the underlying IPC mechanisms. The IO operations for each of the four IPC mechanisms highlighted in Chapter Two (named pipes, message queues, shared memory, and sockets) will be implemented in a separate Java class using the word "channel" as a naming convention. Each channel class will have the same general structure. The design of Linux NIPC will closely follow the implementation of Linux IPC. The core component of each channel class will be a buffer instead of a byte array. This buffer will be pre-allocated when the constructor for a particular channel class is called. This design choice is based on Observation Two in Chapter Two, where reusing the same buffer for IO operations is recommended. Each "channel" class will have a constructor, a reading, and a writing method. The constructor will initialize the underlying IPC mechanism and allocate the buffer. The reading and writing methods will be called the relevant native methods or channels to pass messages.

Implementing IO operations for named pipes is straightforward. Once a native method initializes the pipe, file channels can be used to read and write messages from the buffers to the pipe's file directory.

Implementing IO operations for messaging queues required more native code interaction. Passing messages through the message queue requires specific native methods to be called. These native methods will be called from the reading and writing methods in the IO component and will pass a reference to the direct buffers rather than the message.

Implementing IO operations for shared memory is more complex than the previous two IPC mechanisms. Similar to a message queue, messages must be passed by calling native methods. A reference to the direct buffer is passed to the native methods rather than the message. The pipes and message queues implemented in this study have inherent synchronization. However, shared memory does not. Therefore, the synchronization must be implemented explicitly. There are two implementations for the shared memory mechanism in Linux IPC. Both are implemented in a similar manner. However, there are slight differences in how the synchronization is implemented. One uses a separate native method for each semaphore operation. The other packs these semaphore operations together into fewer native methods. Therefore, the second implementation has significantly fewer native method calls to utilize the shared memory mechanism. Based on the results shown in Figure 1.1 in Chapter One, the implementation with fewer native method calls performed better than the other, having a lower latency. Based on Observation Four in Chapter Two, this makes sense, where increased native method calls mean increased JNI calls, which leads to increased overhead. Therefore, it makes sense to focus on improving the shared memory implementation with fewer native method calls.

Each "channel" class will require a method to deallocate and close the underlying IPC mechanism.

3.2.2 Native Component

The native component gives the IO component access to the underlying IPC mechanisms. Although the native component is not this study's main area of interest, some modifications are still required. The Linux IPC implementation by Wells (2009) passes the message from the IO component to the native component. This means the message needs

to be copied from one array in the Java code to another in the native code. The message must also be copied between the JVM memory heap and the native memory heap, adding an additional data copy. Therefore, two data copies are involved when calling native methods.

However, as discussed in Observations One and Three in Chapter Two, the number of copies can be reduced. By implementing direct buffers, which allow Java applications to allocate memory in the native memory heap, messages do not need to be copied between the two memory spaces, reducing the amount of data copies. Instead, a reference to the direct buffer can be passed from the IO component to the native component. The native methods can directly access and copy messages from the buffer into the underlying IPC mechanisms.

3.3 Experimental Design

The following sections highlight how the performance of Linux NIPC will be tested.

3.3.1 Preliminary Experiment

Since this study focuses on each IPC mechanism's IO component, performing a preliminary experiment on only this component makes sense. This preliminary experiment involves conducting a performance comparison between java.io streams and java.nio channels and buffers. This comparison aims to evaluate the inherent performance differences of each library in isolation, without the added overhead associated with JNI and native code.

3.3.2 Round-Trip Experiment

An experiment will be implemented to test the performance of Linux NIPC. This experiment is based on the same experiment used by Wells (2009) to test the performance of

Linux IPC and will follow the same process as implemented by Wells. The experiment will measure the round-trip time of message passing between two processes.

To conduct this experiment for Linux NIPC, the following criteria must be met:

- The experiment must be executed in two separate terminals so that two separate processes are used.
- The processes must first be synchronized before any timings are computed.
- A message must be passed from one process to another and back to the first. This is considered one round trip.
- These round trips must be timed over several rounds and an average of the total time taken over these rounds.
- The timings for the first round must be excluded due to increased latency from the setup process.
- The message passed between processes must use separate instances of the IPC mechanism being tested. For example, one pipe is used to pass a message from process one to process two. A different pipe is used to pass the message from process two back to process one.
- The round trip timings must be computed using varying message sizes ranging from small to large message sizes.

Using this experiment, the performance of the Linux NIPC implementation in this study can be compared to the Linux IPC implementation by Wells.

3.4 Summary

This chapter gave a high-level overview of the proposed improvements to the Linux IPC library and how each component of the library functions. These improvements will be

compiled into a new library called Linux NIPC. The IO component, which holds the Java IO operations, will be implemented using tools from java.nio. The focus will be on implementing a direct buffer for processes to pass messages through to the underlying IPC mechanisms. This chapter also highlights the requirements for conducting experiments to test this new implementation. The experiments must be conducted similarly to the previous implementation by Wells so that a fair comparison between the two implementations can be made.

4

Implementation

This chapter gives a detailed description of changes made to Linux IPC. These changes are compiled into a new library. A detailed comparison between the two implementations is discussed. A detailed implementation of the experiments used to assess this implementation is also discussed in this chapter.

4.1 Linux NIPC general Implementation

The following sections give a detailed description of the implementation of each component of Linux NIPC:

4.1.1 IO Component

This component allows client programs to interact with the underlying IPC mechanisms. This component has four "channel" classes, where each class holds the IO operations for a particular IPC mechanism. Each channel class comprises a constructor, a public write method, a public read method, and a public close method. Each class contains other private methods depending on the functionality of the particular IPC mechanism. Java applications can call these public methods in order to interact with the underlying IPC mechanisms.

The constructor must first be called in order to use a particular IPC mechanism. The generalized code for the constructors can be seen in Listing 4.1 below. This constructor will call any relevant native methods for initializing the underlying IPC mechanism or

mapping to an existing one. The constructor will also pre-allocate a direct buffer at a fixed size. The buffer can only be accessed by the process that created it and is used when passing messages into the underlying IPC mechanisms.

```
1    public GeneralConstructor() throws IOException {
2        //==================================
3        // NATIVE INITIALIZATION METHOD CALL
4        //==================================
5        buffer = ByteBuffer.allocateDirect(BUFFER_SIZE);
6    }
7
```

Listing 4.1: General Constructor

The "write()" method is used to write messages to a particular IPC mechanism. The generalized code for the write method can be seen in Listing 4.2 below. This method takes in some byte array representing the message and calls the relevant native write method to pass the message to the underlying IPC mechanism. Since the buffer capacity is fixed, the message might be too large for the buffer. In this case, the message is broken down into chunks, each of which is passed separately through the buffer and into the underlying IPC mechanism. A while loop and some control variables are used to process the message in chunks. The "totalBytesWritten" variable tracks how much of the message has been written. The 'bytesToWrite' variable represents the size of a single chunk. Each chunk is written to the direct buffer before the native write method is called to copy that chunk from the buffer into the underlying IPC mechanism.

```
1    public void write(byte[] data) throws IOException {
2        int totalBytesWritten = 0;
3        int totalBytes = data.length;
4        while (totalBytesWritten < totalBytes) {
5            buffer.clear();
6            int bytesToWrite = Math.min(MAX_BUF_SIZE, totalBytes - totalBytesWritten);
7            buffer.put(data, totalBytesWritten, bytesToWrite);
8            //=========================
9            // NATIVE WRITE METHOD CALL
10           //=========================
```

```
            totalBytesWritten += bytesToWrite;
        }
    }

```

Listing 4.2: General Write Method

The "read()" method is used to retrieve messages from a particular IPC mechanism. The generalized code for the read method can be seen in Listing 4.3 below. This method takes in the length of the message that needs to be retrieved and calls the relevant native read method. The native read method will caopy the message from the underlying IPC mechanism into the buffer. The message can then be retrieved from the buffer and returned to the client program. The message might be too large for the buffer and can be broken down into chunks. A while loop and control variables are used to retrieve the message one chunk at a time. The "totalBytesRead" variable tracks how much of the message has been retrieved. The "bytesToRead" variable represents the size of a single chunk. These chunks are then collected and concatenated into a byte array representing the complete retrieved message, which is returned to the client program.

```
    public byte[] read(int totalBytes) throws IOException {
        int totalBytesRead = 0;
        byte[] totalMessage = new byte[totalBytes];
        while (totalBytesRead < totalBytes) {
            buffer.clear();
            int bytesRead = readChannel.read(buffer);
            if (bytesRead == -1) { break; }
            buffer.flip();
            int bytesToRead = Math.min(bytesRead, totalBytes - totalBytesRead);
            //========================
            // NATIVE READ METHOD CALL
            //========================
            totalBytesRead += bytesRead;
        }
        return totalMessage;
    }

```

Listing 4.3: General Read Method

Each channel class also has a "close()" method for closing the underlying IPC mechanism and any channels used during message passing.

4.1.2 Native component

The native component of Linux NIPC is responsible for copying messages from the direct buffers into the underlying IPC mechanisms. This component comprises a Java class and a native C class. The Java class is an interface allowing the IO component to interact with and call the native methods in the native component. The native C class holds a collection of methods, each representing a specific System V IPC operation. These operations can initialize the underlying IPC mechanisms, pass messages to and from the direct buffers into the underlying IPC mechanisms, and perform reverse JNI calls to call Java methods.

Since the IO component will allocate memory in the native memory heap using direct buffers, the native component must be able to access these buffers. A reference to the direct buffer must be passed when calling native methods from the IO component. This allows the native methods to map the memory segment used by the direct buffer using the "GetDirectBufferAddress()" method. This buffer can then be directly accessed by native code. From there, a simple memory copy is done to copy messages from the direct buffer to the underlying IPC mechanisms. The generalized code for accessing direct buffers in native code is shown in Listing 4.4 below:

```
1  JNIEXPORT jint JNICALL NativeMethod (JNIEnv * env, jobject obj jobject buffer) {
2      void *bufferAddress = (*env)->GetDirectBufferAddress(env, buffer);
3      msgbuf *message = (msgbuf *)malloc(sizeof(msgbuf) + size);
4      memcpy(&(message->msg), bufferAddress, sz);
5  }
6
```

Listing 4.4: Direct Buffer Access

4.2 Linux NIPC Detailed Implementation

The following section gives more detail on the implementation of the IO component of each individual IPC mechanism.

4.2.1 Named Pipe Channel

When initializing a named pipe, the named pipe channel constructor calls the "mkfifo()" native method. This creates a temporary writeable file. This file can be opened for reading or writing, and it exists temporarily as long as the pipe is in use.

This implementation uses file channels from java.nio. The file channels are used to open the file and pass messages to the pipe. The "setReadChannel()" and "setWriteChannel()" methods must be called to set the file's directory. The pipe can be opened for either reading or writing. The general reading and writing methods will then use these channels to pass messages through the pipe.

4.2.2 Message Queue Channel

When initializing a message queue, the message queue channel constructor calls the msgget() native method. This method creates the message queue and assigns it a unique key, which is used to identify the queue. The key is essential as it ensures that the processes are accessing the correct message queue.

The "msgrcv()" and "msgsnd()" native methods are used by the general read and write methods to retrieve and send messages to the messaging queue. The key and a message type are passed as parameters to these methods. The message type helps categorize messages, allowing processes to prioritize or filter messages based on their type. This allows selective retrieval of messages, allowing clients to retrieve only the messages they are interested in.

4.2.3 Shared Memory Channel

When initializing a shared memory segment, the shared memory channel constructor calls the "shmget()" native method. When this method is called, a reverse JNI call is used. From the native code, a Java method will be invoked to set the correct segment key value, segment address value, and semaphore key value. These values must be passed to reading and writing methods when interacting with the shared memory segment. This ensures that the processes are accessing the correct shared memory segment and the semaphores can track which process has exclusive access.

When calling the general read and write methods, the native code will use standard memory copy operations to pass messages to the shared memory segment. This means that race conditions will incur if multiple processes try to access this shared memory segment simultaneously. Therefore, semaphores are used to manage the synchronization of this shared memory space. When the "shmget()" native method is called, each process is given a semaphore key. Before reading or writing to a shared memory segment, a process must perform a "wait" operation on the semaphore to ensure that they have exclusive access. Once access is granted, the process can pass messages to the segment. After a process has completed a read or write operation, the process performs a "signal" operation to release the semaphore, allowing other processes to access the shared memory segment. Using semaphores in this way ensures that only one process modifies the shared memory segment at a time.

4.2.4 Socket Channel

When using sockets to communicate between processes, a "feedback loop" is used. This means that messages are passed through a port within the same host. One process must act as the server, and another must act as the client. When the constructor of the socket channel class is called, there is an option to set the process to with a server or client. The server process will listen for incoming connections. The client process will then request to connect to the server process. Messages can then be passed through this connection.

This implementation uses socket channels from java.nio. A socket and server socket channel connect to the same port to pass messages between the server process and the client process.

4.3 Comparison of Linux IPC and Linux NIPC

the IO component of each implementation uses different IO libraries. Linux IPC utilizes java.io with its stream-oriented IO. Linux NIPC utilizes java.nio with its buffers and channels.

Linux IPC extends the primitive stream classes from java.nio. Linux NIPC does not extend any primitive classes. Linux IPC implements a global byte array used as an intermediary buffer. These arrays reside within the JVM memory heap and thus are subject to the garbage collector. Linux NIPC uses the special byte buffers from java.nio instead of byte arrays. The buffers are allocated as direct buffers. Therefore, the buffers reside within the native memory heap and are more efficient.

Since Linux IPC implements classes that extend the primitive stream classes, each IPC mechanism must be broken down into separate classes for input and output. The reading and writing methods implemented in these custom stream classes overload the reading and writing methods from the primitive input and output stream classes. These custom reading and writing methods invoke native methods to pass messages into the underlying IPC mechanisms. Linux NIPC does not extend any primitive classes. The reading and writing methods in Linux NIPC are implemented similarly to the Linux IPC reading and writing methods. Both implementations take in some byte array representing the message and allow for messages to be broken down into chunks and passed into the underlying IPC mechanisms one chunk at a time.

4.4 Experimental Setup

The following sections detail how the experiments were conducted in this study.

4.4.1 Preliminary Experiment Design

The preliminary experiment tests both java.io and java.nio in isolation. This was implemented by performing file IO. File IO was chosen since it is similar to operating on named pipes, a simple IPC mechanism. The time taken to write a message to a file and then read the same message from the file is computed. Reading and writing operations will be timed separately. This is to test how the different operations compare across both libraries.

4.4.2 Round-Trip Experiment Design

The round-trip experiment tests the implemented IPC mechanisms and involves computing a round-trip time. The experiment was conducted as follows:

- **Step One:** Call the constructor for the IO component of the IPC mechanism. This will initialize the underlying IPC mechanism and preallocate the buffer. Additional setup methods must also be called, such as creating channels.
- **Step Two:** Synchronize the processes. Since this simulation is executed by two processes, the processes must be synchronized before any timings can be taken. This ensures an accurate round-trip time is computed.
- **Step Three:** Perform the round-trip timing. The time taken for a message to be passed from one process to another and back to the first is measured. This step is then repeated several times for a given number of rounds, and an average is then taken over the number of rounds to get a more accurate timing.

- **Step Four:** Step three is then repeated for messages of varying sizes ranging from 1024 bytes to 40 960 bytes, increasing by 1024 bytes in each round. This shows how well the Linux NIPC performs when passing smaller and larger message sizes.
- **Step Five:** Close the underlying IPC mechanisms and deallocate the buffers.

4.4.3 Performance Measure

This experiment will measure the round-trip time for passing a message. This directly measures the latency of passing a message between processes. The primary goal of this study is to reduce this latency when utilizing the implemented IPC mechanisms.

4.4.4 Hardware Specifications

All experiments were run on Ubuntu version 22.04. The hardware specifications are listed below:

- Processor: 12th Gen intel© Core™ i5-12400 x 6
- RAM Capacity: 16 gb

4.5 Summary

This chapter gives a detailed description of implementing the proposed Linux NIPC library. The IO component of each IPC mechanism has a similar structure with the smaller differences highlighted. This library focuses on effectively utilizing direct buffers for passing messages. Additionally, since direct buffers are used, the native component must access these direct buffers instead of passing the message between the different memory heaps. This contrasts with the Linux IPC implementation, which extends the primitive steam classes. The following steps are taken to conduct the experiment used to test these

implementations: Initialise the underlying IPC mechanism, Synchronize the processes, perform round-trip timings, and close the underlying IPC mechanism.

5

Results

This chapter reports on the results obtained from the experiments conducted in this study.

5.1 Preliminary Experiment Results

This experiment aimed to evaluate the differences between java.io and java.nio in isolation, without the added overhead of JNI calls and native code. The results recorded for this experiment are shown in figure 5.1 below:

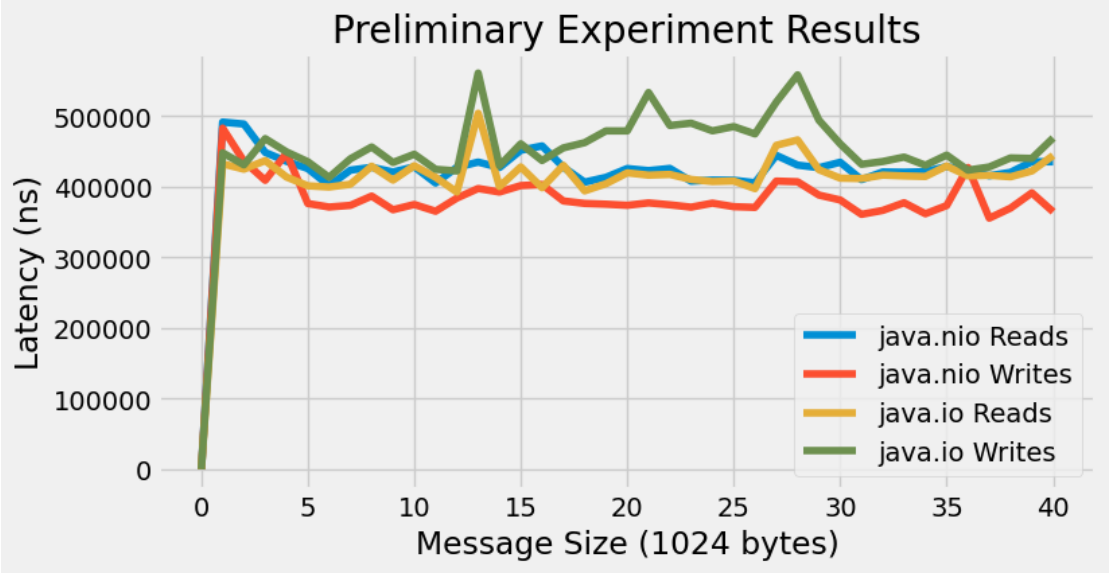

Figure 5.1: Preliminary Experiment Results

These results show that java.nio performs the same or better than java.io. The timings for reading were relatively the same for both java.io and java.nio. The average latency across

all timings for reading operations for java.nio was 98.1% of the java.io latency. However, there is a larger difference in writing operations. The average latency across all timings for writing operations for java.nio was 84.3% of the java.io latency.

5.2 Round-Trip Experiment Results

This experiment aimed to evaluate the difference in performance between the implementation from this study and the implementation by Wells (2009). The timings recorded are measured in nanoseconds. The results of the timings reported for each implementation can be seen below, followed by a comparison between the two implementations.

5.2.1 Linux NIPC Results

The results recorded for the Linux NIPC implementation from this study are shown in figure 5.2 below.

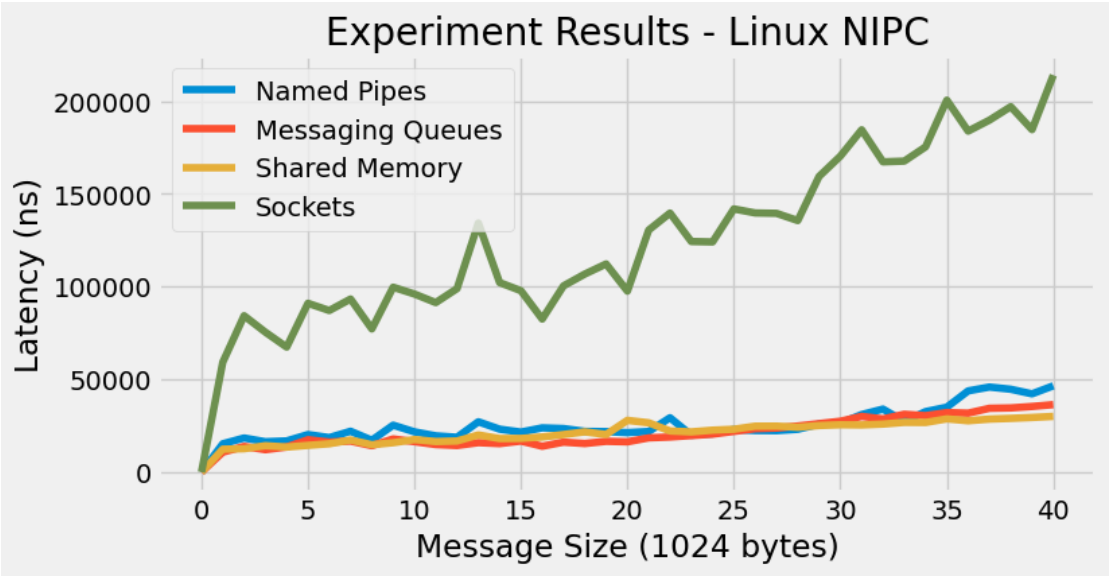

Figure 5.2: Linux NIPC results obtained in this study

As expected, the implemented IPC mechanisms, namely named pipes, message queues, and shared memory, performed significantly better than sockets.

The IPC mechanism that performed best against sockets was the message queues. Overall, the average latency across all the timings taken was 16.6% of the socket's latency. At the smallest message size, 18.0%, and at the largest message size, 16.9%. Shared memory performed slightly worse, the average latency across all the timings taken was 17.4% of the socket's latency. At the smallest message size, the latency was 20.7% of the socket's latency. At the largest message size, the latency was 14.0% of the socket's latency. Named pipes performed the worst, where the average latency across all the timings taken was 20.5% of the socket's latency. At the smallest message size, 25.7%, and at the largest message size, 21.7%.

5.2.2 Linux IPC Results

The results recorded for the Linux IPC implementation by Wells (2009) are shown in Figure 5.3 below:

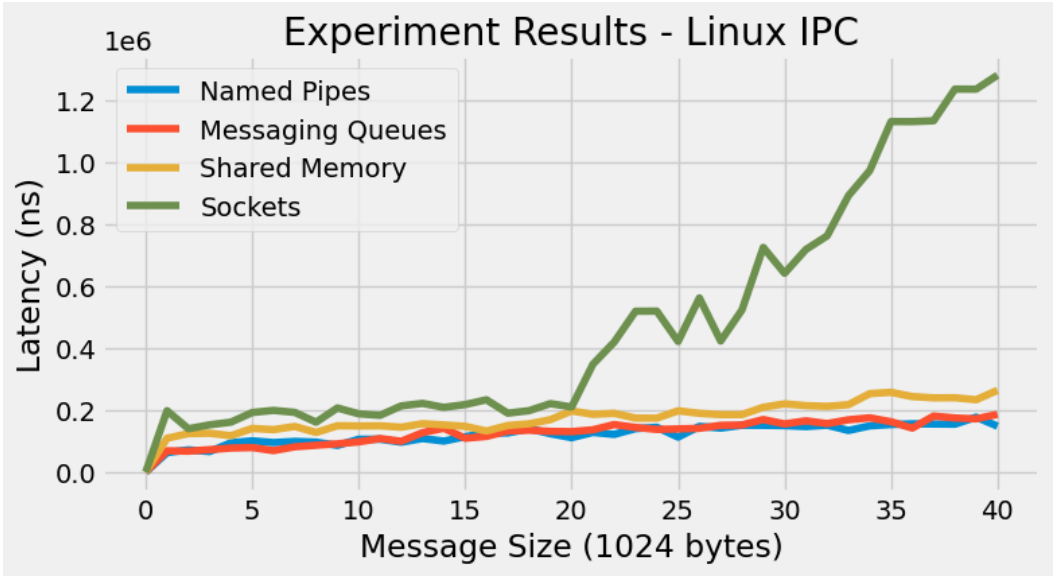

Figure 5.3: Linux IPC results obtained in this study

The IPC mechanism that performed best against sockets was the named pipes. Overall, the average latency across all timings taken was 36.9% of the socket's latency. At the smallest message size, 31.6%, and at the largest message size, 11.5%. Message queues performed slightly worse, the average latency across all the timings taken was 37.5% of the socket's latency. At the smallest message size, 34.6%, and at the largest message size, 14.4%. Shared memory performed the worst, where the average latency across all the timings taken was 52.8% of the socket's latency. At the smallest message size, the latency was 55.1% of the socket's latency. At the largest message size, the latency was 20.5% of the socket's latency.

these results show similar patterns when compared to the results recorded by Wells, shown in Figure 5.4, when running the same experiment on different hardware. Both show that the implemented IPC mechanisms perform similarly to sockets up to a point. Thereafter, the latency of sockets increases significantly.

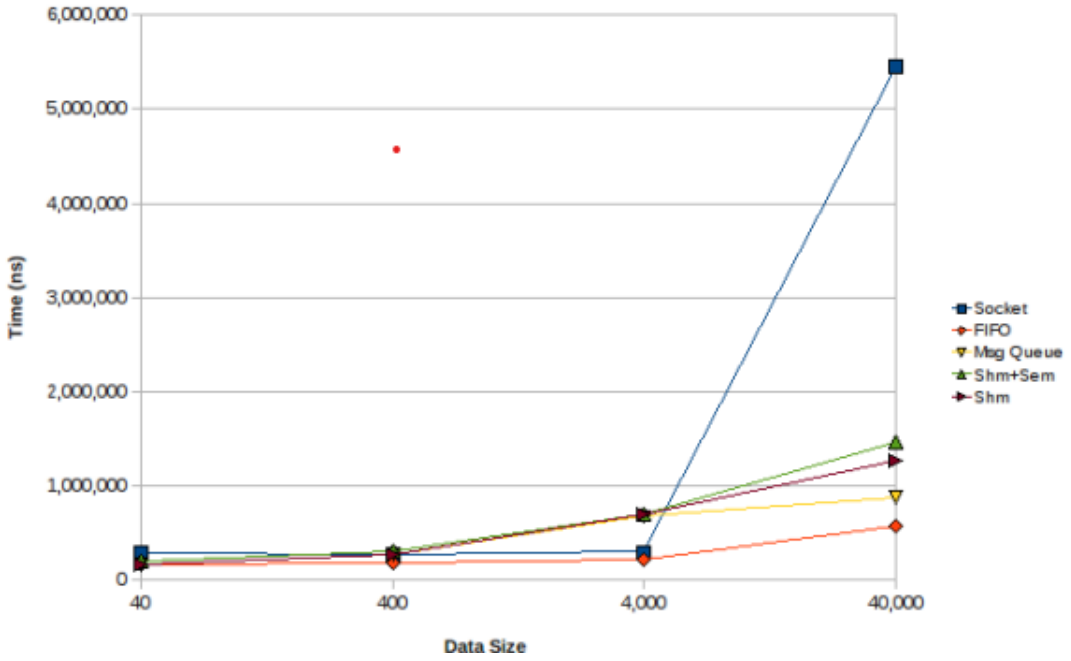

Figure 5.4: Linux IPC results obtained by Wells (2009)

5.3 Comparison

The results obtained by Wells in Figure 1.1 and the results shown in Figure 5.3 suggest that named pipes should perform better than message queues and shared memory. However, based on the results shown in Figure 5.2, this study's implementation of named pipes performed slightly worse than message queues and shared memory. This could be because the named pipes rely on the IO component to handle data copying. In contrast, the message queues and shared memory rely on the native component to handle data copying.

The following sections compare the results of each IPC mechanism from each implementation individually.

5.3.1 Named Pipes

A comparison of the results obtained for named pipes between the two implementations is shown in Figure 5.5 below:

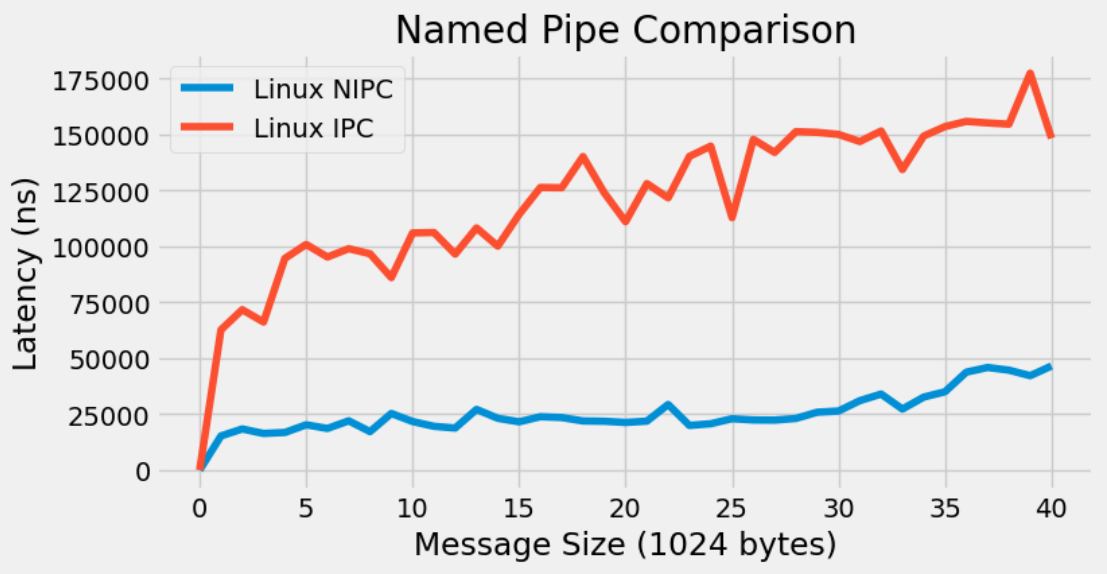

Figure 5.5: Named Pipe Results Comparison

The Linux NIPC implementation for named pipes performed significantly better. The average latency across all timings taken was 20.9% of the Linux IPC implementation. At the smallest message size, 25.6%, and at the largest message size, 31.3%.

5.3.2 Message Queues

A comparison of the results obtained for message queues between the two implementations is shown in Figure 5.6 below:

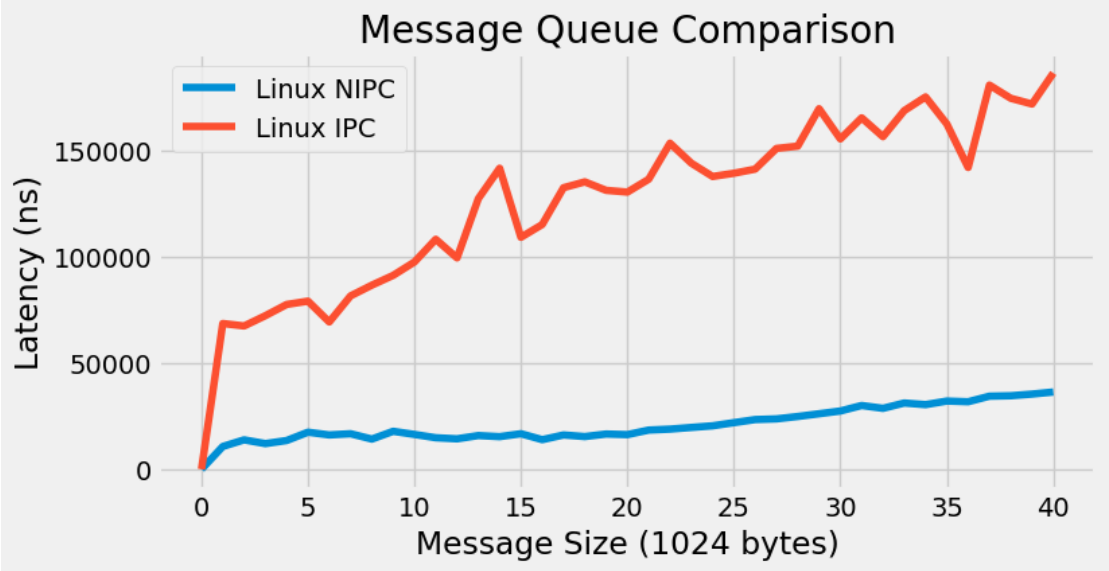

Figure 5.6: Message Queue Results Comparison

The Linux NIPC implementation for message queues performed significantly better. The average latency for this implementation across all the timings taken was 16.5% of the Linux IPC implementation. At the smallest message size, 15.6%, and at the largest message size, 19.4%.

5.3.3 Shared Memory

A comparison of the results obtained for shared memory between the two implementations is shown in Figure 5.7 below:

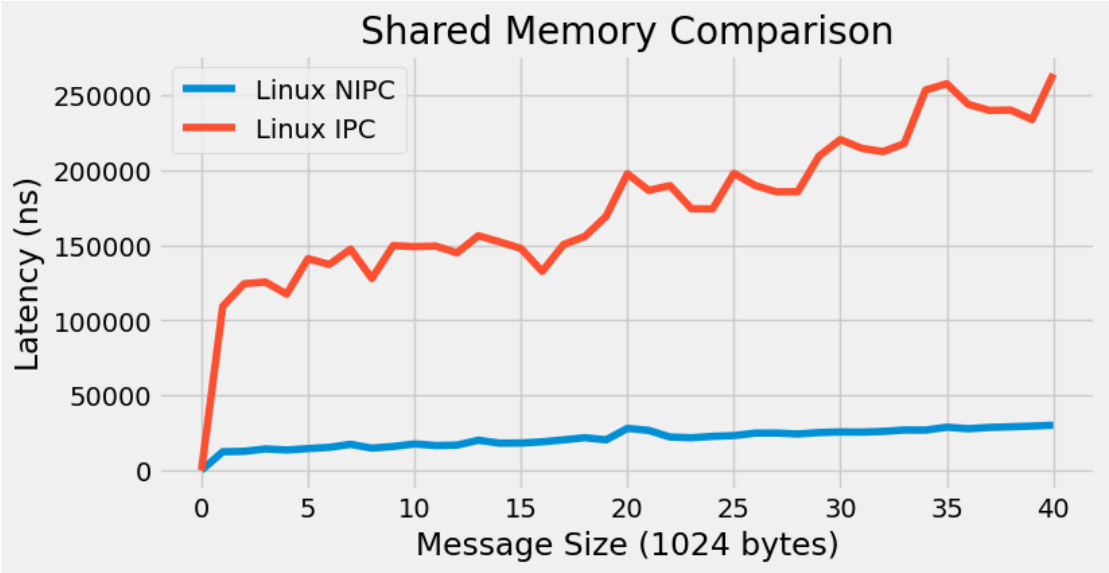

Figure 5.7: Shared Memory Results Comparison

The Linux NIPC implementation for shared memory performed significantly better, with the average latency across all the timings taken was 12.0% of the Linux IPC implementation. At the smallest message size, 11.2%, and at the largest message size, 11.3%.

5.4 Summary

This chapter reported on the results obtained from the preliminary and round-trip experiments. The preliminary experiment results showed very little improvement in performance when utilizing java.nio compared to java.io. The round-trip experiment results were obtained for the Linux IPC and NIPC implementations. A comparison between these implementations was conducted. The results showed a significant reduction in the

latency of passing messages for Linux NIPC compared to Linux IPC. It was also noted that some IPC mechanisms performed differently depending on the implementation. Named pipes performed the best for the Linux IPC implementation but not for the Linux NIPC implementation.

6

Conclusion and Future Work

This chapter summarises the work done in this study and discusses if and how the research objectives were achieved. Avenues for future work and improvement are also discussed.

6.1 Concluding Remarks

This study aimed to investigate the performance differences between java.io and java.nio. This study also aimed to improve the Linux IPC library implemented by Wells (2009). The main objective was to reimplement the IO operations that allow Java applications access to the underlying IPC mechanisms available in Unix systems. The proposed improvements involved implementing IO tools offered by the java.nio library, such as direct buffers and file channels.

The results obtained from the preliminary experiment showed that the performance of java.nio is the same or better than java.io, depending on the IO operation performed. This experiment was conducted using only file IO to pass messages. This experiment could be taken further, and each mechanism provided by the two IO libraries is compared to get a more comprehensive comparison detailing the performance difference between them.

From the results obtained from the round-trip experiment, the implementation developed in this study performed significantly better than the previous implementation by Wells. These results showed a significant difference in the latency of the two implementations. therefore, utilizing java.nio led to a significant improvement in the Linux IPC library. However, some trends among the results were not the same between the two implementations. Named pipes performs best in the Linux IPC implementation. But message queues

performs best in the Linux NIPC implementation. This could be due to the context of the implementation of the IO component for each IPC mechanism. Named pipes relied on file channels implemented in the IO component for copying messages. Shared memory and message queues relied on memory copy operations in the native component for copying messages. These are two different environments and can, therefore, lead to very different results when performing message passing. Further investigation is required in order to assess which environment is better for performing data copying.

Overall, these results give conclusive evidence that implementing java.nio under certain contexts can lead to significantly improved performance. The main goal of this study was to improve the efficiency of message-passing libraries. Direct buffers offered by java.nio allow Java applications to allocate memory directly to the native memory heap. The underlying IPC mechanisms can then access these buffers directly. This reduces the number of data copies required to pass messages and, in turn, improves the efficiency of passing messages. therefore, in this context, java.nio is very effective regarding performance gain. However, based on the preliminary experiment results, java.nio does not always perform better than java.io. these results showed that java.nio performs the same or slightly better with basic IO operations without context. Therefore, if direct buffers were not needed or made no difference under a certain context, there are no benefits to utilizing java.nio. Further experimentation is required to verify this claim.

6.2 Future Work

There are a few avenues for future work concerning the library developed in this study.

The main limitation of the Linux NIPC library developed in this study is that it depends on naive libraries and underlying IPC mechanisms unique to UNIX systems. Fortunately, some work porting this type of library to other operating systems, such as Windows or Mac OS. In the study by Smith and Wells (2017), they developed a similar library using IPC mechanisms native to the Windows operating system. This implementation again

utilizes JNI and libraries to access underlying IPC mechanisms available in Windows environments.

Another approach would be to develop a pure Java implementation of some IPC tools. Systems like MPJava by Pugh and Spacco (2004) and the study by Ramos *et al.* (2013) implement message-passing systems that do not rely on any underlying IPC mechanisms or native libraries. Since direct buffers allocate memory in the naive memory heap, multiple processes are able to access the same direct buffers. This can lead to the development of custom IPC mechanisms using direct buffers. Such a library would be portable and able to run on any operating system that supports Java.

References

Aamir, S. and Jawad, M. Towards efficient shared memory communications in MPJ Express. In *Java workshop at the 23rd IEEE International parallel and distributed processing symposium*. 2009.

Artho, C., Hagiya, M., Potter, R., Tanabe, Y., Weitl, F., and Yamamoto, M. Software model checking for distributed systems with selector-based, non-blocking communication. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 169–179. IEEE, 2013.

Baker, M., Carpenter, B., and Shafi, A. An approach to buffer management in Java HPC messaging. In *International Conference on Computational Science*, pages 953–960. Springer, 2006a.

Baker, M., Carpenter, B., and Shafi, A. MPJ Express: towards thread safe Java HPC. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2006b.

Belikov, E., Deligiannis, P., Totoo, P., Aljabri, M., and Loidl, H.-W. A survey of high-level parallel programming models. *Heriot-Watt University, Edinburgh, UK*, 1:2–2, 2013.

Ciccozzi, F., Addazi, L., Asadollah, S. A., Lisper, B., Masud, A. N., and Mubeen, S. A comprehensive exploration of languages for parallel computing. *ACM Computing Surveys (CSUR)*, 55(2):1–39, 2022.

Dickens, P. M. and Thakur, R. An evaluation of Java's IO capabilities for high-performance computing. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 26–35. 2000.

Fleisch, B. D. Distributed System V IPC in LOCUS: a design and implementation retrospective. *ACM SIGCOMM Computer Communication Review*, 16(3):386–396, 1986.

Grichi, M., Abidi, M., Guéhéneuc, Y.-G., and Khomh, F. State of practices of Java native interface. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, pages 274—-283. IBM Corp., USA, 2019.

Lee, S. JNI program analysis with automatically extracted C semantic summary. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 448—-451. 2019.

Mutia, R. I. Inter-process communication mechanism in monolithic kernel and microkernel. *Department of Electrical and Information Technology Lund University, Sweden*, (2014), 2014.

Pugh, W. and Spacco, J. Mpjava: High-performance message passing in Java using java.nio. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16*, pages 323–339. Springer, 2004.

Ramos, S., Taboada, G. L., Expósito, R. R., Tourino, J., and Doallo, R. Design of scalable java communication middleware for multi-core systems. *The Computer Journal*, 56(2):214–228, 2013.

Smith, D. and Wells, G. Interprocess communication with Java in a Microsoft Windows environment. *South African Computer Journal*, 29(3):198–214, 2017.

Travis, G. Getting started with New IO (NIO). *IBM Corporation*, 9, 2003.

Venkataraman, A. and Jagadeesha, K. K. Evaluation of inter-process communication mechanisms. *Architecture*, 86:64, 2015.

Wells, G. Inter-process communication in Java. In *PDPTA*, pages 407–413. 2009.