Syracuse University

Electrical Engineering & Computer Science Department

CSE-690 - Independent Study

"Inter-process Communication System using Shared Memory"

Design Document

Instructor: Professor Jim Fawcett

Prepared By: Ammar Salman - SUID 670086269

December, 17th 2017

Abstract

In this project, we have designed, implemented, and tested an “Inter-process Communication System using Shared Memory”. This design document provides some details about the main principled idea and technique used for the built structure, and a working description of the implementation. The appendix provides the untouched results of our experiments.

Our concept was based on utilizing shared memory as a common working area for messages exchange and inter-communication. Our testing has shown a considerable success in performance and reliability, for this early development. Fine-tuning and improvements could yield better performance for medium and long messages.

In addition, we have made a benchmark comparison between our shared-memory based systems, with two well established professional communication systems, namely the Windows WCF with Basic HTTP Binding, and WCF with Net Named Pipe Binding.

The comparison with WCF showed our system outperformed the basic and the pipe binding systems for the short and large number of messages exchange, while it slowed down for larger ones. Our initial explanation for this anomaly is rooted in the design of putting a cap on the waiting time of the sending thread for the receiver thread.

The other explanation could be found in the WCF Basic HTTP Binding system utilization of the full functionality of direct access socketing, where the receiver can process the message while receiving it. This advantage was not available to our system and could be the case for the slower pace of WCF Net Named Binding.

Contents

I. Introduction

Inter-process communication systems have been around ever since operating systems and the concept of ***virtual address spaces*** were introduced. It is a system that manages sending and receiving data from one process to another.

All operating systems use the concept of virtual address space to manage memory and allocate spaces to processes. Each process is allocated a number of memory blocks and handles them internally the way it needs. Generally, processes do not attempt to use memory outside their virtual boundaries, and when they need to, they call the operating system for help.

Operating systems provide their own Application-Programming-Interfaces (APIs) to allow guest programs to use the host OS services. OS services mainly run in two domains: kernel and user domains.

Kernel domain is owned and managed by the operating system and no program has any direct access or control over any kernel object. Calls to kernel objects are made through the provided API and are executed by the operating system. On the other hand, user domain objects are owned and managed by the guest programs.

There are different execution levels on any machine. The operating system acts as the lord of the hardware. It has absolute control over the machine resources. Supported by hardware, operating systems provide different privilege levels for guest programs. For instance, in Windows or Linux, there is the unprivileged mode and the administrator (or root) mode.

There are many misconceptions regarding the administrator mode that suggest the running program has absolute control. That is simply not true. Administrator programs are not allowed to execute in kernel mode nor they can enter the kernel domain. All calls made by such programs are translated into operating system calls executed by the operating system itself. The operating system can terminate these programs whenever it sees suitable.

The reason access-control is important in this area is its direct relation to the concept of shared-memory. As already explained, there is a virtual address space for each program, which is not allowed to interfere with the addresses spaces of other programs. However, if there is a need for two processes to use the same memory area, the so-called t ***shared-memory*** is used.

Shared-memory is a set of memory blocks that do not belong to any process. They are shared by processes that want to use it. The owner is still the operating system, which brings us back to the access-control. Only the operating system can allocate and deallocate shared-memory address spaces.

When processes need to use shared memory, they have to ask the operating system to allocate the area and give them what are called '***handles***' to that area. Handles is a term widely used when it comes to Windows. It is defined as a pointer to an object that can be a process, thread, memory area descriptor, window… etc.

When multiple processes want to access shared-memory area, how can they get handles to the same area? This is where the ***named objects*** concept kicks in.

Named objects are created and owned by the operating system given certain names. The name provides an easy way to manage things for users. Operating systems, however, might have some rules for naming objects. For example, Windows uses ***namespaces*** to hold named objects, and the namespace identifier must be provided along with a request to create a named object. Most commonly used namespace on Windows is the 'Global' namespace, which allows ***private namespaces*** for security.

In this document, we describe the design of an ***inter-process communication*** system that uses ***named objects*** and ***shared memory*** to achieve inter-process ***fast communication***.

We present the system design in Section II and describe the main components of the system and how they are interconnected. In Section III we describe the implementation and technical details. In Section IV, we discuss related systems and focus mainly on Windows Communication Foundation (WCF), which we used for comparison. In Section V, we compare our system performance with WCF. Finally, the appendix shows the raw output of our experiments.

II. Design

II.A Overview

The main goal is to achieve fast self-hosted message passing communication system. Shared-memory is very fast as all operations are done on the computer's main memory. The idea in this inter-process communication system is to use shared-memory as the medium for sending and receiving messages. Each process is assigned shared-memory area to receive messages in, and it is responsible for setting the size of that shared memory area. Each shared-memory area acts as a one-item queue. In this system, it is essentially a message made of a sequence of ANSII characters.

As shown in Figure II.1, each process has a shared-memory area associated with it. The corresponding process is the only one that can read from that area while other processes can write to it. Mutual-exclusion is ensured so that no two processes can access the same area at the same moment.

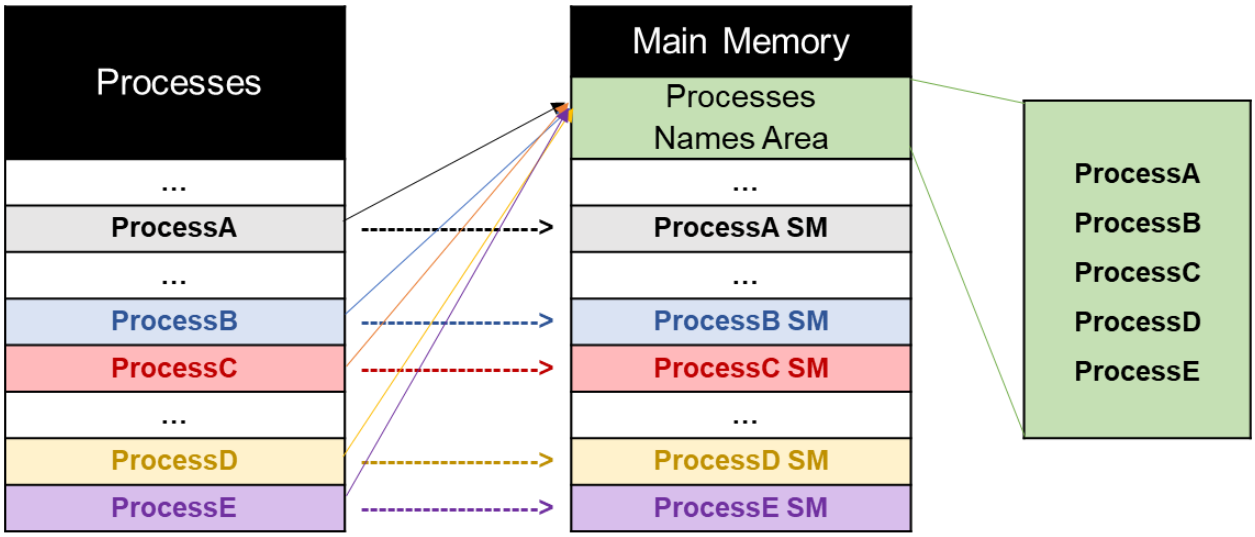

Figure II.1: Processes and main memory layout. On the left there is a list of processes. In the center there is the main memory, which includes an area described as 'Processes Names Area'. This area is responsible for holding the names of all processes in the system as shown on the right, which is the content of the 'Processes Names Area'.

This is a scenario of a buffer with a single-consumer and multiple-writers. The consumer is the process that the area is associated with, and the multiple-writers are all other processes in the system.

When a process fills in the area of another process, it notifies the target process and all writers must wait until the target process consumes and processes the message.

The shared area contains all processes names and that enables each process to know about other processes in the communication system. This allows for better management and connectivity. The first process to initialize communication will create that area and the rest will get reference to it.

II.B Class Relationships

There are four main classes used in this system: Inter-process communication, inter-process locking, blocking queue and message classes. Figure II.2 describes the main classes and their relationships. It is worth to note that different C++ Standard Template Library containers were used, but were omitted in the diagram to simplify pointing out the important roles of our classes.

Inter-process communication: it the main class in the system. It composes two blocking queues, one to store received messages and the other is used to hold outgoing messages. In addition, It composes multiple 'inter-process locking' instances, where the number of those instances depends on the number of processes in the communication

system. The process uses ‘message’ class to reconstruct a received message while pushing a message in the outgoing queue.

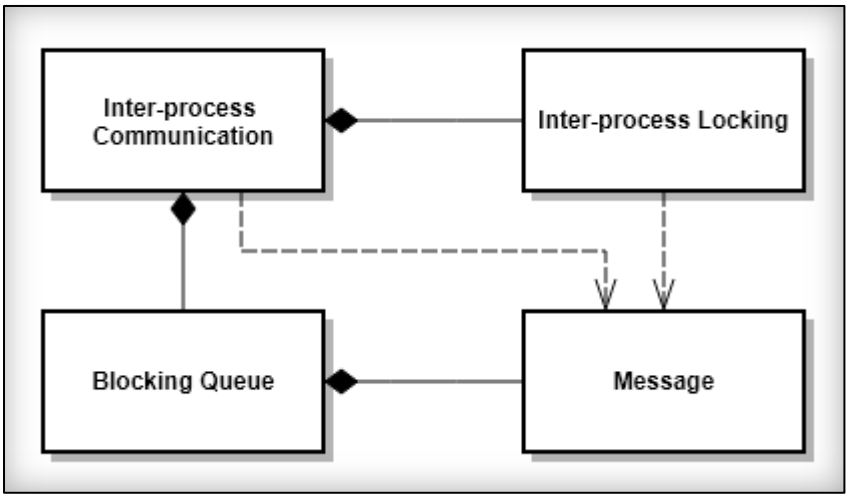

Figure II.2: Main classes and their relationships.

Inter-process locking: acts as a proxy or a channel between processes. This allows processes to synchronize access to the shared-memory area. ‘Message’ is used in this class when a process is putting a message inside another process’ area.

Blocking queue: a generic container used to hold outgoing and incoming messages. It composes ‘message’ as it contains multiple instances of that type.

Message: a simple structure with basic information. It holds information about source, destination, command and body. It must provide standard serialization/de-serialization mechanism in order for processes to understand how to deal with it.

II.C System Operations

The main operations can be summarized into the following:

1. **Initialize communication**: accepts name and attempts to start a communication object using that name. There are different options to use when attempting to initialize communication. This operation returns result value indicating whether or not the initialization worked.
2. **Close communication**: attempts to close communication.
3. **Open channel:** opens a channel to another process to send messages.
4. **Close channel**: closes a channel previously opened to another process.
5. **Post message**: put a message in the outgoing messages queue.
6. **Get message**: return a message from the received messages queue.

III. Implementation

The system was mainly developed as C++ library while making calls to Windows API C functions. The code was wrapped with external C functions to allow other programming languages to import and use the library. The library was also imported into C# code where all the testing took place.

III.A: Shared-Memory

To create shared memory, we had to refer to Windows API functions. The Windows API provides functions to allocate memory blocks from the main memory. There are two kernel objects involved in this procedure:

- **File mapping**: the raw allocated memory blocks from the main memory.
- **Map view**: maps the file mapping into virtual address space to allow processes to use the file mapping.

Creating and using file mapping and map view objects is done through Windows and the code is executed in the Windows kernel via translation calls which are also handled by Windows. Those details are important because they show the need of running our library in administrative mode in order for the system to function.

The Windows API allows users to ask for the allocation of variable memory size. Windows does not actually allocate the exact size the user wishes because it allocates in 4KB ***pages***. For example, asking for 1000 bytes for file mapping, ends up having a 4KB file mapping but accessible to only 1000 bytes. The rest is essentially a waste.

The same also applies to the map view object. It can map up to the maximum of the accessible size of the file mapping object.

Moreover, when a process creates and maps a file mapping object, and another process connects to that object, the other process does not know the size of that area. Windows API has a function called ‘Virtual Query’ that returns the size mapped by a map view object, but the reported size comes in multiples of 4K. For example, 4K+1 rounds-up to 8K. Hence it is not possible for the guest processes to know exactly the accessible size in a file mapping object.

With that in mind, we have designed our system to only accept per-process shared-memory areas that are actual multiples of 4KB in size. This way, we always ensure full utilization of the area and enable other processes to know the exact size of the file-mapping object with no speculation and that helps maintain sending reliability as discussed in III.D.

Anyway, all file mapping objects must be named for our system to work. We followed the naming conventions allowed by Windows API. Windows has namespaces and all any named object has to reside in a namespace. There are three main namespaces:

- Global namespace: anything can access it.
- Local namespace: access is restricted to the creator's session.
- Session namespace: reserved for Windows. Windows creates a Session namespace for each process and when that process creates an object using the Local namespace, that object is placed in the creator process Session namespace. Windows does not allow users to directly use the Session namespace.

Users are not restricted only to the Global and Local namespaces. While those namespaces are the only one available by default, users can create their own private namespaces. Private namespaces can also be shared by multiple processes. The main advantage is the security they provide.

In this implementation, we *used the Global namespace* with private namespaces in mind for future development to add security measures.

III.B: Management

Global Management

To let all processes be completely aware of each other, we must have a globally shared memory area where all those processes put and read each other's names from.

The shared-memory area has a name hardcoded in each communication object. The first communication object initialization creates that shared-memory area, puts its name there and maintains the handle to the area. When another object initializes, it obtains a handle to that shared area and put its name there. This way, that area will always have all the names of the active communication objects.

Anyway, to ensure **data integrity** in the shared-memory area, only one process can access it at any moment of time. To ensure **mutual exclusion**, we used a named ***mutex***, which also has a name hardcoded in each communication object. When an object wants to update the shared area, it must acquire the mutex first. At this stage, other objects will be blocked until the mutex is released. When the active object is done it releases the mutex for other objects to acquire it.

Per Object Management

The main operation is exchange messages t(send and receive) with other communication objects. Our model treats sending and receiving as two distinct operations that should not interfere with each other within the same process. Therefore, we added two different blocking queues in each object. One contains all messages meant for sending, and the other contains all received messages.

Furthermore, to completely separate the two operations within the same process, we initialized two threads per communication object. The sender thread takes messages from the outgoing messages queue and attempts to send them. The receiver thread

waits for messages from other communication objects and once signaled, puts the received message in the received queue. Section III.C provides more details.

III.C: Initialization & Closing

Initializing the communication object

The initialization procedure is meant to start the communication object with respect to other existing objects. When initialization begins, the object connects to the shared area, which has the names of all other communication objects. If the given name already exists in the area, the object will fail to initialize as another object already has the same name. Once the object is initialized, it will sends connection messages to all other objects notifying them of its initialization. This allows other objects to create connections to the newly initialized object. Moreover, it allows all objects to always have the latest names list available.

Closing the communication object

Closing is similar to initialization in the way it sends connection messages to all objects notifying them of its closing state. This allows other objects to close their connections to the closing object. This step is crucial because of the way kernel objects work.

When a kernel object is created, it is assigned a reference counter managed by the operating system. When someone connects to the object, the reference counter is incremented. When someone closes its connection to the object, the reference counter is decremented. Once the reference counter is zero, the operating system destroys it.

This explains the crucial need of notifying other communication objects of your closing. To allow them to lose their references to your kernel objects. If they did not lose their references, the closing communication object would not be able to re-initialize itself on the same name.

Anyway, upon closing, the two worker threads (sender and receiver) must be allowed to exit normally. That is why they are both notified of the closing and waited upon to finish their execution before the closing functionality completes.

III.D: Sending & Receiving

Communication here relies on the model of a single-consumer multiple-writer buffer. To achieve such model, the receiver object and the sender object (not sender and receiver thread within the same communication object) must handshake on each sent message.

Therefore, ***two semaphores*** are needed to associate with each communication object. One semaphore represents the ***consumer*** while the other represents ***all writers***.

The receiver thread of the communication object will wait on the consumer semaphore. At this stage, the writers semaphore is free for anyone. When another communication object wants to send a message, its sender thread will attempt to acquire the writer

semaphore of the target communication object. Once the semaphore is acquired; the message will be copied to the shared-memory area of the target object. The sender thread then signals the consumer semaphore of the target process notifying it that there is a message in your area.

Since it is a single-consumer buffer, the sender should not signal the writers semaphore since a message is already there and no one should overwrite it. Therefore, the sender just continues its loop. At this point, the receiver thread in the receiving object is notified, it reads the message, clears the memory area and finally signals the writers semaphore notifying any waiting writer that the its free to receive a new message.

Now, assume the receiver is getting a very large message and it needs time to process it. During that time, if some sender wants to send a new message, it may wait for a long before the receiver signals it to put the new message. Therefore, we allowed the system to set a time limit the sender should wait for the receiver to signal it. If the wait is timed-out, the sender pushes the message at the end of its outgoing messages queue, and attempts to send another message from the queue. This is to make sure the sender does not waste a lot of time trying to send one message.

III.E: Message Transfer

As described in the design section earlier. A message is essentially a structure with four main attributes; source, destination, command and body.

The main question is how do we ensure correct transfer the message from one process to another with an exact same content on the other end?

The answer is mainly in the serialization and de-serialization mechanism. The serialization procedure converts a structure into a series of bytes with known length. Then, the message can be sent over the medium. The reverse procedure of a serialized structure is called de-serialization, or reconstruction.

Initially, we used XML for serialization and de-serialization. Even though XML is flexible and provides a wide range of features, it turned out to be extremely slow to reconstruct. We used to convert the message into an XML string, send the string, and rebuild the message from an XML string on the receiving end.

For an average sized message, XML reconstruction took about 1000 times more than the transfer time, making the communication extremely slow. Therefore, we replaced XML with a basic and fast serialization/de-serialization mechanism.

Instead of converting the message into an XML string, we convert it to a string with markers. A marker is a fixed size hexadecimal number that tells how many bytes to read for the current attribute. For example:

0008processA0008processB0007command0004body

The first 4 characters are hex representation for 8, hence 8 characters are read after that number and put as the source. The next 4 characters are also hex representation for 8, so the following 8 characters are put as destination. And so on. With this mechanism, reconstruction time is extremely fast and does not degrade performance of the communication.

IV. Related Systems

IV.A: Sockets

Sockets are the most commonly used inter-process communication mechanism. They rely mainly on the TCP/IP model to transfer data from one place to another. However, sockets do not necessarily have to use TCP/IP. On Windows, sockets are smart enough to know when the communication is initiated by two processes on the same machine. In that case, the processes buffers are immediately connected to transfer data. This helps reduce significantly the overhead added by the communication.

IV.B: Named Pipes

Named pipes are widely used for inter-process communication on the same machine. While they provide the option to transfer data across networks, they are not a recommended option as the overhead for such systems is huge compared to sockets.

IV.C: Windows Communication Foundation (WCF)

WCF is a .NET library that was built on top of Windows Component-Object-Model (COM) and WinAPI sockets. It provides the ability to use both sockets and named pipes. It was optimized to use SOAP protocol and to run on Windows environment. Mono project managed to make WCF work on Linux and Mac systems.

WCF provides two main options for communication:

1. **Server-client model**: heavily relies on the operating system to provide the server service. On Windows, WCF uses IIS server, on Linux it uses Apache.
2. **Self-hosted model**: each communication system is completely self-hosted and does not rely on OS servers to do the job for it. This is useful to distribute working load.

WCF also uses different binding protocols that bind service hosts and their proxies. We are mostly interested in Basic HTTP Binding and Named Pipe Binding. In the next section, we will run comparisons.

V. Results & Analysis

V.A: Tests Description

We have compared our system to WCF to see performance differences. The tests were performed on three systems:

- Our shared-memory based system.
- WCF with Basic HTTP Binding.
- WCF with Net Named Pipe Binding.

All tests were performed on the same machine:

- Model: Acer Aspire R15 571TG.
- Processor: Intel Core i7 7500U. 4 cores – 4 threads. Clock rate: ~2.7GHz
- RAM: 12GB DDR4 @ 2133 MHz
- Storage: 1TB @5400RPM.
- Dual Display: Intel HD Graphics 620 and GeForce 940MX.

Message structure:

- Source: string representing source of the message.
- Destination: string representing destination of the message.
- Command: string-representing command of the message.
- Body: string representing the body of the message.

The tests involved different message sizes and different overall sizes:

Test no	No. of messages	Body size in characters
1	100,000	100
2	100,000	520
3	10,000	5,030
4	5,000	51,100
5	50	5,242,795
6	25	52,428,700
7	5	104,857,500
8	1	1,073,741,740

Table V.1 8 tests with variable size and number of messages

The last test (Test 8) where the message is nearly 1GB, was not performed on WCF systems as they do not allow such sizes. It was performed on our system. We mainly measured the following:

- Total time to send all messages.
- Average time per message.

- Average time per 1KB transfer.

V.B: Systems Configuration

Shared Memory System

In shared-memory system, there was only one process to make the tests easier, in that process there were two communication objects:

- Communication A: two threads, one for sending and one for receiving.
- Communication B: two threads, one for sending and one for receiving.

In addition to the threads described above, there is also the main thread, which issues the send requests, and keeps spinning until all messages are sent.

Total: 5 threads in 1 process.

The system was also configured to wait 5ms before the sender attempts to send another message and push the current one at the end of the outgoing messages queue.

Windows-Communication-Foundation Systems

In the WCF systems, there were two dedicated processes:

- Receiver process: one thread to receive messages and clear them to make space.
- Sender process: one thread to send messages to the receiver process.

Total: 2 threads in two processes.

It is worth to note that neither of the systems used in the testing is suitable for cross-machine communication. Although Basic HTTP Binding would work for cross-machine communication, it should have additional protocols to support those operations. Such additions would add overhead and since all of our tests are meant to be done on one machine, we excluded those protocols.

V.C: Results

The results are summarized in the following table:

No	Size	Shared-Memory IPC			WCF – Basic HTTP			WCF – Named Pipes		
		Total(s)	Avg	1KB(us)	Total(s)	Avg	1KB(us)	Total(s)	Avg	1KB(us)
100K	180B	4.8	48us	273	16.59	165us	943	4.8	48us	274
100K	600B	5.2	52us	88	16.47	164us	281	6.14	61us	104
10K	5KB	0.59	59us	11	1.83	182us	36	0.81	81us	16
5K	50KB	0.79	159us	3	1.85	371us	7	2.55	509us	10
50	5MB	2.25	45ms	8	1.65	33ms	6	2.77	55ms	10
25	50MB	12.85	514ms	10	11.4	456ms	8	14.13	565ms	11
5	100MB	5.78	1.16s	11	3.9	781ms	7	4.78	956ms	9
1	1GB	14.14	14.14s	13	-	-	-	-	-	-

Table V.2 performance comparisons between our system and WCF systems

V.D Analysis and comments

Figure V.1 shows the average message transfer time versus the message size. As shown in the figure, our system started significantly faster the WCF Basic, and about the same speed as WCF Pipes. When we started sending messages with size 5MB and larger, the increase of the message transfer time was larger than the increase in WCF Basic. Surprisingly, WCF-Piped message transfer time increase was also larger than that of WCF Basic.

To explain the behavior of our system, we looked at the fifth test (5MB messages) and noticed the average transfer time to be around 45ms. Our system, by default, sets the sender's timeout period to 5ms. That means many of the outgoing messages were being pushed back at the end of the queue (See III.D) which created an overhead explaining why the increase is larger than that of WCF-Basic.

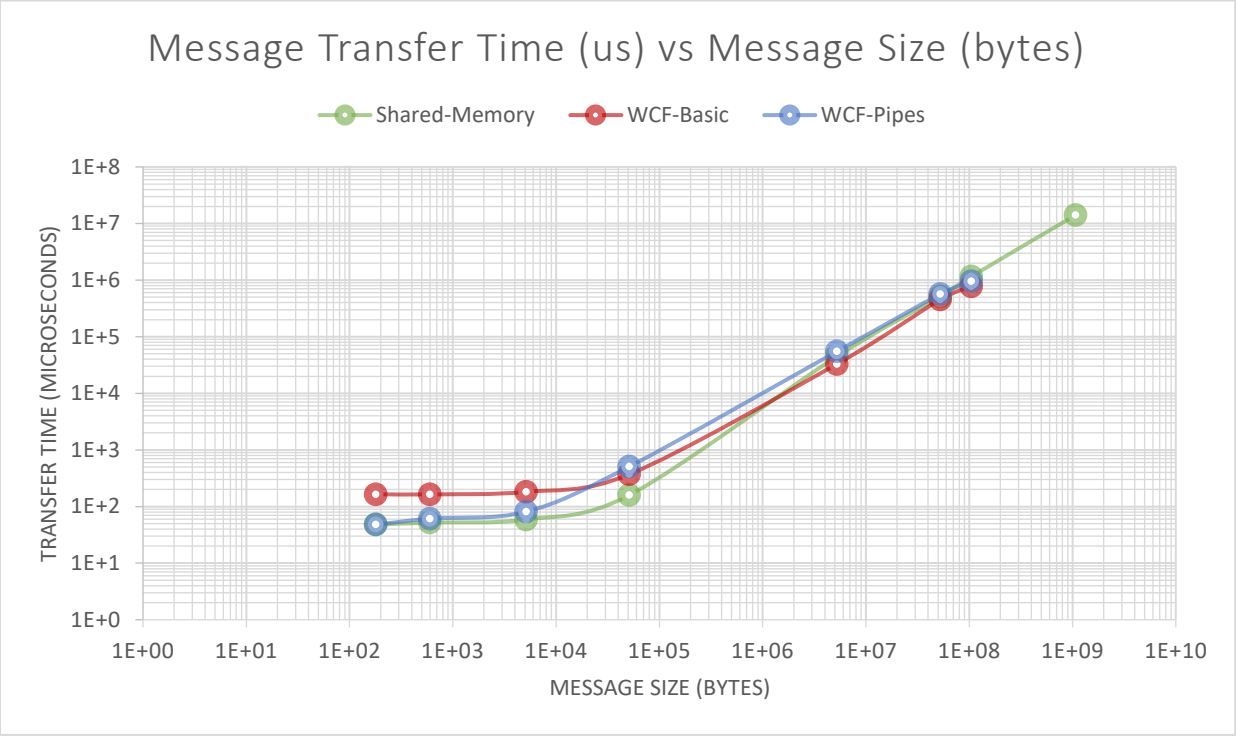

Figure V.1: Message Transfer Time versus Message Size

However, that still does not explain why WCF-Basic is still the fastest. In Section IV we talked about Windows Sockets and the way, they perform, and that could explain the fast performance of WCF-Basic. If WCF is using Windows sockets, the sockets would connect the host and the channel buffers directly and start the data transfer. This means that the host is consuming the message ***while*** the channel is sending it. This might explain the situation considering how our system (and possibly WCF-Pipes) was slower. In our system, the message is completely copied to the receiver's buffer ***before*** the

receiver is notified. In addition, the receiver completely copies the message from the buffer before it notifies any of the waiting writers.

Anyway, it is worth mentioning that when we tested our system to work between two ***different*** processes, we were able to get a transfer time of 36us per message which is about 1.25 times faster than what we showed in the results above. However, because of sometime shortage we will stick with the current results until we have time to perform further tests.

Appendix

Appendix A: Shared Memory System – Raw Test Results

```
================================================================================
----------------------< Demo: Sending 10 Normal Messages >----------------------
================================================================================

 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 Sending 5 messages from 'CommunicationA' to 'CommunicationB' and vice versa

      Source : CommunicationB
 Destination : CommunicationA
     Command : Command
        Body : Message #1

      Source : CommunicationA
 Destination : CommunicationB
     Command : Command
        Body : Message #1

      Source : CommunicationB
 Destination : CommunicationA
     Command : Command
        Body : Message #2

      Source : CommunicationA
 Destination : CommunicationB
     Command : Command
        Body : Message #2

      Source : CommunicationB
 Destination : CommunicationA
     Command : Command
        Body : Message #3

      Source : CommunicationA
 Destination : CommunicationB
     Command : Command
        Body : Message #3
```

```
      Source : CommunicationB
 Destination : CommunicationA
     Command : Command
        Body : Message #4

      Source : CommunicationA
 Destination : CommunicationB
     Command : Command
        Body : Message #4


      Source : CommunicationB
 Destination : CommunicationA
     Command : Command
        Body : Message #5

      Source : CommunicationA
 Destination : CommunicationB
     Command : Command
        Body : Message #5

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Sending 10 normal messages demonstration successfully finished.

================================================================================
================================================================================


================================================================================
---------< Sending 100000 Messages. Size: ~180Byte each, ~17MB overall >--------
================================================================================

 IPC on 'CommunicationB' will be receiving 100000 messages with bodies of size: 100
 Setting the message queue size in 'CommunicationB' to 4096 (equivalent to 1 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:27.990] - Sending 100000 msgs. Body size: '100'. From 'A' to 'B'
 [12/16/2017 02:15:32.803] - Sent 100000 msgs. Body size: '100'. From 'A' to 'B'

 Total time to send 100000 messages: 4.8130831 seconds
 Average time per message: 48 microseconds
 Average time per 1KB: 273 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
---------< Sending 100000 Messages. Size: ~600Byte each, ~57MB overall >--------
================================================================================
```

```
 IPC on 'CommunicationB' will be receiving 100000 messages with bodies of size: 520
 Setting the message queue size in 'CommunicationB' to 4096 (equivalent to 1 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:33.944] - Sending 100000 msgs. Body size: '520'. From 'A' to 'B'
 [12/16/2017 02:15:39.147] - Sent 100000 msgs. Body size: '520'. From 'A' to 'B'

 Total time to send 100000 messages: 5.2035378 seconds
 Average time per message: 52 microseconds
 Average time per 1KB: 88 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 10000 Messages. Size: ~5KB each, ~49MB overall >-----------
================================================================================

 IPC on 'CommunicationB' will be receiving 10000 messages with bodies of size: 5030
 Setting the message queue size in 'CommunicationB' to 8192 (equivalent to 2 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:40.304] - Sending 10000 msgs. Body size: '5030'. From 'A' to 'B'
 [12/16/2017 02:15:40.897] - Sent 10000 msgs. Body size: '5030'. From 'A' to 'B'

 Total time to send 10000 messages: 0.5937338 seconds
 Average time per message: 59 microseconds
 Average time per 1KB: 11 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 5000 Messages. Size: ~50KB each, ~244MB overall >----------
================================================================================

 IPC on 'CommunicationB' will be receiving 5000 messages with bodies of size: 51100
 Setting the message queue size in 'CommunicationB' to 53248 (equivalent to 13 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
```

```
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:41.929] - Sending 5000 msgs. Body size: '51100'. From 'A' to 'B'
 [12/16/2017 02:15:42.726] - Sent 5000 msgs. Body size: '51100'. From 'A' to 'B'

 Total time to send 5000 messages: 0.7968331 seconds
 Average time per message: 159 microseconds
 Average time per 1KB: 3 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
------------< Sending 50 Messages. Size: ~5MB each, ~250MB overall >------------
================================================================================

 IPC on 'CommunicationB' will be receiving 50 messages with bodies of size: 5242795
 Setting the message queue size in 'CommunicationB' to 5242880 (equivalent to 1280 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:43.820] - Sending 50 msgs. Body size: '5242795'. From 'A' to 'B'
 [12/16/2017 02:15:46.070] - Sent 50 msgs. Body size: '5242795'. From 'A' to 'B'

 Total time to send 50 messages: 2.2502363 seconds
 Average time per message: 45004 microseconds
 Average time per 1KB: 8 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 25 Messages. Size: ~50MB each, ~1250MB overall >-----------
================================================================================

 IPC on 'CommunicationB' will be receiving 25 messages with bodies of size: 52428700
 Setting the message queue size in 'CommunicationB' to 52428800 (equivalent to 12800 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:15:47.398] - Sending 25 msgs. Body size: '52428700'. From 'A' to 'B'
 [12/16/2017 02:16:00.243] - Sent 25 msgs. Body size: '52428700'. From 'A' to 'B'

 Total time to send 25 messages: 12.845093 seconds
```

```
 Average time per message: 513803 microseconds
 Average time per 1KB: 10 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
------------< Sending 5 Messages. Size: ~100MB each, ~500MB overall >-----------
================================================================================

 IPC on 'CommunicationB' will be receiving 5 messages with bodies of size: 104857500
 Setting the message queue size in 'CommunicationB' to 104857600 (equivalent to 25600 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:16:02.212] - Sending 5 msgs. Body size: '104857500'. From 'A' to 'B'
 [12/16/2017 02:16:07.994] - Sent 5 msgs. Body size: '104857500'. From 'A' to 'B'

 Total time to send 5 messages: 5.7818614 seconds
 Average time per message: 1156372 microseconds
 Average time per 1KB: 11 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
--------------< Sending 1 Messages. Size: ~1GB each, ~1GB overall >-------------
================================================================================

 IPC on 'CommunicationB' will be receiving 1 messages with bodies of size: 1073741740
 Setting the message queue size in 'CommunicationB' to 1073741824 (equivalent to 262144 Windows Pages)
 Initializing IPC on 'CommunicationA'
 Successfully initialized IPC on 'CommunicationA'
 Initializing IPC on 'CommunicationB'
 Successfully initialized IPC on 'CommunicationB'

 [12/16/2017 02:16:15.104] - Sending 1 msgs. Body size: '1073741740'. From 'A' to 'B'
 [12/16/2017 02:16:29.246] - Sent 1 msgs. Body size: '1073741740'. From 'A' to 'B'

 Total time to send 1 messages: 14.1421589 seconds
 Average time per message: 14142158 microseconds
 Average time per 1KB: 13 microseconds

 Closing 'CommunicationA' ... Closed
 Closing 'CommunicationB' ... Closed
```

```
 Demonstration successfully finished.

================================================================================
================================================================================
```

Appendix B: WCF Basic HTTP Binding – Raw Test Results

```
 Initializing Sender to use BasicHTTPBinding protocol...
 Sender initialized.

================================================================================
---------< Sending 100000 Messages. Size: ~180Byte each, ~17MB overall >--------
================================================================================


 [12/16/2017 02:04:25.210] - Sending 100000 msgs. Body size: '100'.
 [12/16/2017 02:04:41.795] - Sent 100000 msgs. Body size: '100'. From 'A' to 'B'

 Total time to send 100000 messages: 16.5854682 seconds
 Average time per message: 165 microseconds
 Average time per 1KB: 943 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
---------< Sending 100000 Messages. Size: ~600Byte each, ~57MB overall >--------
================================================================================


 [12/16/2017 02:04:41.795] - Sending 100000 msgs. Body size: '520'.
 [12/16/2017 02:04:58.266] - Sent 100000 msgs. Body size: '520'. From 'A' to 'B'

 Total time to send 100000 messages: 16.4704988 seconds
 Average time per message: 164 microseconds
 Average time per 1KB: 281 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 10000 Messages. Size: ~5KB each, ~49MB overall >-----------
================================================================================


 [12/16/2017 02:04:58.266] - Sending 10000 msgs. Body size: '5030'.
 [12/16/2017 02:05:00.094] - Sent 10000 msgs. Body size: '5030'. From 'A' to 'B'

 Total time to send 10000 messages: 1.8283179 seconds
 Average time per message: 182 microseconds
 Average time per 1KB: 36 microseconds
```

```
 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 5000 Messages. Size: ~50KB each, ~244MB overall >----------
================================================================================


 [12/16/2017 02:05:00.094] - Sending 5000 msgs. Body size: '51100'.
 [12/16/2017 02:05:01.954] - Sent 5000 msgs. Body size: '51100'. From 'A' to 'B'

 Total time to send 5000 messages: 1.8595725 seconds
 Average time per message: 371 microseconds
 Average time per 1KB: 7 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
------------< Sending 50 Messages. Size: ~5MB each, ~250MB overall >------------
================================================================================


 [12/16/2017 02:05:01.985] - Sending 50 msgs. Body size: '5242795'.
 [12/16/2017 02:05:03.641] - Sent 50 msgs. Body size: '5242795'. From 'A' to 'B'

 Total time to send 50 messages: 1.6564238 seconds
 Average time per message: 33128 microseconds
 Average time per 1KB: 6 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 25 Messages. Size: ~50MB each, ~1250MB overall >-----------
================================================================================


 [12/16/2017 02:05:04.001] - Sending 25 msgs. Body size: '52428700'.
 [12/16/2017 02:05:15.408] - Sent 25 msgs. Body size: '52428700'. From 'A' to 'B'

 Total time to send 25 messages: 11.4074298 seconds
 Average time per message: 456297 microseconds
 Average time per 1KB: 8 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================
```

```
================================================================================
------------< Sending 5 Messages. Size: ~100MB each, ~500MB overall >-----------
================================================================================


 [12/16/2017 02:05:16.221] - Sending 5 msgs. Body size: '104857500'.
 [12/16/2017 02:05:20.128] - Sent 5 msgs. Body size: '104857500'. From 'A' to 'B'

 Total time to send 5 messages: 3.9066359 seconds
 Average time per message: 781327 microseconds
 Average time per 1KB: 7 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================
```

Appendix C: WCF Net Named Pipe Binding – Raw Test Results

```
 Initializing Sender to use NetNamedPipeBinding protocol...
 Sender initialized.

================================================================================
---------< Sending 100000 Messages. Size: ~180Byte each, ~17MB overall >--------
================================================================================


 [12/16/2017 02:07:10.012] - Sending 100000 msgs. Body size: '100'.
 [12/16/2017 02:07:14.837] - Sent 100000 msgs. Body size: '100'.

 Total time to send 100000 messages: 4.8244818 seconds
 Average time per message: 48 microseconds
 Average time per 1KB: 274 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
---------< Sending 100000 Messages. Size: ~600Byte each, ~57MB overall >--------
================================================================================


 [12/16/2017 02:07:14.837] - Sending 100000 msgs. Body size: '520'.
 [12/16/2017 02:07:20.978] - Sent 100000 msgs. Body size: '520'.

 Total time to send 100000 messages: 6.1412774 seconds
 Average time per message: 61 microseconds
 Average time per 1KB: 104 microseconds

 Demonstration successfully finished.

================================================================================
```

```
================================================================================


================================================================================
-----------< Sending 10000 Messages. Size: ~5KB each, ~49MB overall >-----------
================================================================================


 [12/16/2017 02:07:20.978] - Sending 10000 msgs. Body size: '5030'.
 [12/16/2017 02:07:21.791] - Sent 10000 msgs. Body size: '5030'.

 Total time to send 10000 messages: 0.8125862 seconds
 Average time per message: 81 microseconds
 Average time per 1KB: 16 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 5000 Messages. Size: ~50KB each, ~244MB overall >----------
================================================================================


 [12/16/2017 02:07:21.791] - Sending 5000 msgs. Body size: '51100'.
 [12/16/2017 02:07:24.338] - Sent 5000 msgs. Body size: '51100'.

 Total time to send 5000 messages: 2.5471461 seconds
 Average time per message: 509 microseconds
 Average time per 1KB: 10 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
------------< Sending 50 Messages. Size: ~5MB each, ~250MB overall >------------
================================================================================


 [12/16/2017 02:07:24.369] - Sending 50 msgs. Body size: '5242795'.
 [12/16/2017 02:07:27.135] - Sent 50 msgs. Body size: '5242795'.

 Total time to send 50 messages: 2.7659181 seconds
 Average time per message: 55318 microseconds
 Average time per 1KB: 10 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
-----------< Sending 25 Messages. Size: ~50MB each, ~1250MB overall >-----------
================================================================================
```

```
 [12/16/2017 02:07:27.526] - Sending 25 msgs. Body size: '52428700'.
 [12/16/2017 02:07:41.656] - Sent 25 msgs. Body size: '52428700'.

 Total time to send 25 messages: 14.1306131 seconds
 Average time per message: 565224 microseconds
 Average time per 1KB: 11 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================


================================================================================
------------< Sending 5 Messages. Size: ~100MB each, ~500MB overall >-----------
================================================================================


 [12/16/2017 02:07:42.609] - Sending 5 msgs. Body size: '104857500'.
 [12/16/2017 02:07:47.391] - Sent 5 msgs. Body size: '104857500'.

 Total time to send 5 messages: 4.7817573 seconds
 Average time per message: 956351 microseconds
 Average time per 1KB: 9 microseconds

 Demonstration successfully finished.

================================================================================
================================================================================
```