Imperial College
London

SOFTWARE ENGINEERING GROUP PROJECT REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Low-Latency Inter-Process Communication for High Frequency Trading

Authors:
Ruo Chen,
Dongchi Li,
Wenzheng Shan,
Bruno Wu,
Zekun Yang,
Weixuan Zeng

Supervisor:
Paul Bilokon

January 10, 2022

Submitted in partial fulfillment of the requirements for the MSc Computing (Software Engineering) of Imperial College London

Executive Summary

In the rapidly evolving world of high-frequency trading (HFT) — algorithmic trading in which large volumes are bought and sold automatically at very high speeds — low latency is a matter of survival. In the words of Carl Cook, in HFT, a "microsecond is an eternity" [3]. Speed is paramount, so HFT firms tend to be highly specialised using sophisticated technology and highly competitive. Under the client's guidance, qSpark — an ultra low latency trading platform provider, a solution that achieves a latency of 11 nanoseconds per message was built. This is all the more critical today when HFT is responsible for about 50% of trading volume in U.S. equity markets [2].

High-frequency trading strategies are commonly built on distributed architectures with many independent components for high reliability. An effective way to improve the latency of these strategies is to shorten the communication overhead between components. Thus, prior to meeting qSpark, the supervisor suggested building a low-latency message bus for efficient communication between components through network sockets. However, after presenting the initial prototype to the client, midway the project, it was concluded that it was infeasible to compete with existing solutions, and there were other ways to develop a more useful solution.

As a result, the project had to pivot to low-latency inter-process communication (IPC), where there was a better opportunity to improve qSpark's HFT platforms. Trading platforms tend to have data pipelines with process nodes, and the communication overhead within the pipeline's critical path can be reduced. Thus, the project aims to provide a one-way communication medium between a single live, low latency, critical process and another secondary process — off the critical path — such as transaction loggers. Furthermore, the critical process (producer) overhead has to be minimised whilst the secondary process (consumer) latency is less critical. The producer should send small messages of 40 bytes under 100 nanoseconds of latency at a high throughput rate whilst maintaining high reliability. Finally, the latency variance should be minimised and large latency spikes mitigated, adversely impacting the trading strategy. A high-precision timer with a precision of tens of nanoseconds with minimal overhead was necessary to achieve all this.

The resulting solution was a ring buffer inspired by the LMAX Disruptor implemented using shared memory to send data between the producer and consumer with C++. This resulted in 66 nanoseconds of latency per message sent. The ring buffer also allowed for high throughput of messages whilst maintaining low latency and high reliability. Shared memory methods ensure that data is written and read from the ring buffer at ultra-low latency. Furthermore, isolating the processes to specified CPU cores helped reduce the latency and mitigate large latency spikes from cache misses. Finally, a batch writing feature was implemented to handle high throughput better, achieving 11 nanoseconds of latency per message on batch average.

Chapter 1 Introduction

High-frequency trading (HFT) is algorithmic trading in which large volumes are bought and sold automatically at very high speeds, usually built with high-performance computers and state-of-art network technology [2]. Currently, HFT stands for 50% of trading volume in U.S. equity markets and up to 43% in European equity markets [2]. These strategies rely on quickly analysing real-time market data and translating them into actions in the form of orders at large quantities (millions per day) and extremely high speeds, where even nanoseconds of latency matter. Financial market participants using such strategies capitalise on correcting infinitesimal market inefficiencies - the tiniest price discrepancies between buyers and sellers existing for fractions of a microsecond [11]. Furthermore, since the profit made is a minuscule portion of the orders, millions have to be performed daily to make a substantial return. Therefore low latency is paramount to the success of such strategies. However, with the increasing competition in the industry, participants have to focus on shaving off nanoseconds in their trading strategies to remain competitive.

The project outcome is a solution that can improve the latency of existing HTF strategies that companies use. The project was carried out in collaboration with qSpark, and their expertise was essential to shaping each iteration of the project.

At the start, prototypes of low-latency messaging buses were built. Their purpose was to ensure fast communication between different computers/servers, crucial for the functioning of HFT strategies. The initial prototypes were written in Rust and C++ with UDP for networking, then we compared with existing solutions such as RabbitMQ. However, it was concluded that socket communication was too slow for HFT — latency on the order of magnitude of microseconds was deemed uncompetitive.

Hence, midway through the project, there was a pivot to build a low-latency interprocess communication (IPC) mechanism — to transfer data between processes. According to the client, building an IPC mechanism was deemed to have more potential to be incorporated in HFT. However, it had to perform at sub-100 nanoseconds of latency and handle high throughput whilst maintaining correctness and reliability.

1.1 Objectives

Due to secrecy common in this finance branch, it was challenging to gather background literature on public information. Thus the client, qSpark, was the primary source of the objectives:

1. The system should allow for one way communication between a single producer to a single consumer without data loss.
2. The system should be able to horizontally scale to many instances of single producers and single consumers.

3. The producer should have a writing latency under 100 nanoseconds.
4. The producer should be able to send data packets of a fixed size of 40 bytes.
5. The consumer should be able to receive data packets of a fixed size of 40 bytes.
6. The writing latency's standard deviation should be under 15 nanoseconds.
7. The producer should be able to handle a high throughput of at least 10,000,000 messages per second at a consistent latency.
8. The occurrence and quantity of latency spikes on the system should be minimised.

1.2 Ethics

The ethics behind HFT are polarising, and the arguments are subject to much debate and research. Historically, HFT has been associated with significant controversy and ethical concerns. One of the significant opinions concerns its fairness [9]. Its dominance and power in the market have the potential for "market misuse and manipulation" — harming, particularly, small investors [9]. Market participants have also questioned its "lack of concern for the betterment of society, contributing little of value, and not creating value-added" [9]. Nevertheless, the increase in HFT practice has been correlated with reduced market abuse since it contributes to improving market liquidity and efficiency, reducing the costs for small investors to participate [9].

Low-latency programming has also been associated with Real-Time Machine Learning, one of the main areas of development of Defence Advanced Research Projects Agency (DARPA). Being part of the U.S. Department of Defence, such technology contributes to "next-generation defence systems, such as autonomous vehicles and arrays of sensors". Thus there is potential for military applications of our project, giving rise to further ethical concerns [8].

1.3 Acknowledgements

We want to acknowledge Nimrod Sapir, Nataly Rasovsky, Erez Shermer and Anna Arad from qSpark for help shaping the project's direction. The knowledge and expertise provided regarding HFT and low-latency programming were quintessential to developing the resulting solution. Moreover, the guidance given by the supervisor, Paul Bilokon, was invaluable to the formulation, planning and execution of the project.

Chapter 2 Design

2.1 Architecture

This part introduces the high-level architecture of the low-latency inter-process communication system, which contains a single producer, single consumer and shared memory. At the heart of the Disruptor [12] mechanism, a pre-allocated bounded ring buffer data structure and a header are instantiated. Each producer and consumer owns a unique disruptor instance as the interface to manipulate message data with shared memory. The architecture is shown in Figure 2.1.

2.1.1 Data Structure

Both header and ring buffer inside shared memory is pre-allocated by the producer. At the beginning of the shared memory address is the header structure, which includes the status of the ring buffer and the meta-data of producer and consumer.

Immediately after the header is the ring buffer [4], which contains data of a predefined size. The default ring buffer size is 1024, but it could be redefined by the user when initialising. Furthermore, each data follows the BufferData format of a 40-byte char array and an 8-byte integer as the payload. Instead of using a FIFO queue, a circular queue is used in our disruptor mechanism. Messages from producers are stored in the ring buffer, and two lock-free pointers, cursor and next are used for managing the writing position. When the written data exceeds the maximum length of the buffer, the new data is written from the beginning and overwrites the data used. However, we have safeguards to prevent overwriting, which we will present in Section 2.1.2.

2.1.2 Producer

The producer is responsible for initialising the shared memory and writing data into the ring buffer. The first step for a producer, when sending data, is to claim the position to write on. The purpose is to prevent producers from writing to the buffer without limit and overwriting data that consumers have not yet read. Then set data into the position calculated from the smallest position that the consumer reads and the next pointer. Finally, commit the transaction to update the pointers.

2.1.3 Consumer

The consumer is responsible for reading data from the ring buffer. When reading the data, the first step for consumers is to retrieve the reading index. Each time the consumer passes a reading index to the Disruptor. If the index exceeds the current cursor, the consumer will be blocked to wait for the producer to write on it. When new data is in the position, the busy waiting stops. However, busy waiting could harm CPU utilisation. Otherwise, the Disruptor return the current cursor position. In this way, consumers do not need to request the Disruptor multiple times to obtain all the current data. After retrieving the index from the first step, the consumer could directly read data through a specific API.

The final step is to commit the read transaction to update the consumer reading index array. Since each consumer only points to one component in the array, no concurrency mechanism is needed.

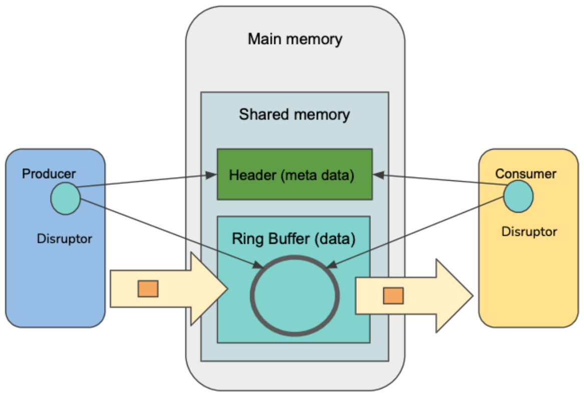


Figure 2.1: Disruptor Structure

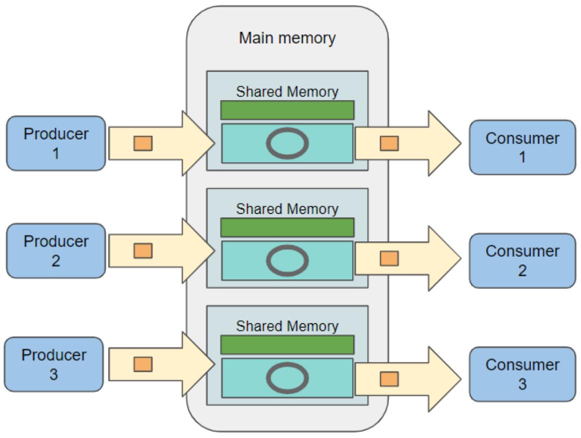


Figure 2.2: Multi-producer/Multi-consumer model

2.2 Disruptor design choices

The standard buffer, a first-in-first-out queue, is considered the initial data structure. The FIFO queue has many advantages; for example, it could be fast since no other calculations are needed to calculate the next written position. However, the maximum size of the queue when initialising the Disruptor cannot be determined. If the Disruptor dynamically allocates memory, the latency would be high, and there is a danger of exceeding the memory limit. Hence, the circular queue is used to address the above problem. Furthermore, if the cursor is greater than the size of the ring buffer, the index is kept growing, and the remainder of the cursor and size of the ring buffer is taken to determine the next written position. A trick to improve efficiency is mentioned in Section 3.2.

Additionally, synchronous queues, blocking queues where the producer sends a message and waits for the consumer to read, are also considered when designing the structure. Although it would be swift if the consumer and producer established a connection, the disadvantages are obvious. More specifically, the producer has to wait for the data to be read before doing anything else. Furthermore, some data might be lost or overwritten before consumption.

2.3 Inter-process communication

As mentioned in Section 2.1, the producer and the consumer are independent processes in the design. Thus, data generated by the producer should be transmitted to the consumer through inter-process communication (IPC). Choosing an appropriate inter-process communication model is critical to implementing a low-latency system. Common inter-process communication models [14] are described below:

2.3.1 Pipe

In Linux, there are two types of pipes: unnamed pipe and named pipe. Due to the anonymity of the pipe file descriptor, unnamed pipes can only be used between parent and child processes. To solve this problem, named pipes create a globally visible file so that any process can access it to use the pipe. However, both types of pipes offer one-way communication. One can only write into a pipe from the input end and read from the output end, which leads to low communication efficiency [6].

2.3.2 Message queue

A message queue is a linked list of messages stored in the kernel [7]. Messages from the message queue can be retrieved by referring to the message type rather than following a FIFO order. However, the message queue model requires four message copies to complete the data transmission:write to the message queue, copy messages from the message queue to the kernel, copy messages from the kernel to the read buffer, and read from the read buffer, which will increase the overhead of the communication and affect the performance of the system.

2.3.3 Shared memory

The shared memory model implements IPC by sharing a memory segment with multiple processes. Firstly, a process initialises the shared memory by requesting a memory segment from the kernel. The kernel will then return the physical address of the memory. Next, each process needs to map this physical address to its virtual address, attaching the process to the shared memory. In this way, each process can directly access the shared memory for reading and writing . Shared memory consists of only two message copies: writing to memory and reading messages. Thus, shared memory is the fastest method of inter-process communication.

After the comparison, shared memory was selected for the low latency system because of its high message transfer speed. Since all attached processes can access the shared memory at will, synchronisation becomes an inevitable problem. Choosing a low-overhead synchronisation approach should also be considered when designing the system.

2.4 Synchronization

As mentioned in Section 2.3.3, shared memory is used for inter-process communication in the design. However, shared memory itself does not provide any synchronisation mechanism, which means that any process attached to shared memory can access it at any time, leading to data inconsistency. Additionally, any synchronisation mechanism often adds extra overhead to the system. In order to solve this problem, the following synchronisation choices are made according to the features of the system:

2.4.1 Single-producer/Single-consumer

Start with the simplest case: single-producer/single-consumer. In this case, there are two synchronisation problems. One is to guarantee that the consumer always read the new data, and the other is what to do with the full ring buffer.

When to read

The first problem occurs when the reading is faster than the writing. In this case, the data that the consumer wants to read has not been updated by the producer. To solve this problem, a variable called *cursor* that points to the last index written by the producer is introduced. If the index that the consumer wants to read data from is smaller than the *cursor*, it means that all data prior to the index has been updated so that the consumer can directly read until the index of the *cursor*. Otherwise, the data the consumer wants to read has not been updated, and busy waiting is required until the producer updates this index.

Full buffer

Since the buffer is bounded, it will be filled up if the producer writes faster than the consumer reads. The producer needs to know whether the old data can be updated to continue writing. The last index accessed by the consumer will be saved to solve this problem. Before writing, the producer compares the claimed index to the last reading index. If the last reading index is greater than the claimed index, the data has been read and updated. Otherwise, the producer should wait until the consumer reads the data.

2.4.2 Multi-producer/Multi-consumer

In practice, the single-producer/single-consumer model is too ideal. The system needs to serve multiple producers and consumers simultaneously. As a result, the synchronisation problem faced by the system becomes more complicated. For example, when multiple producers are writing in parallel, it should be guaranteed that the same block of data is not accessed by more than one process. Typically, a mutual exclusion operation is used to ensure the integrity of the critical section. Nevertheless, it will result in extra overhead. Moreover, in the case of multiple consumers, a counter should be maintained for each data to keep track of whether all consumers read the data. However, updates to this counter can also lead to incorrect values due to synchronisation problems. Therefore, extra mechanisms should be considered to ensure the atomicity of updating operation, which also affects the system's performance. After investigating and consulting with the client, multiple single-producer/single-consumer models are adopted to solve the problem. As shown in Figure 2.2, multiple ring buffers are maintained simultaneously, with only one producer and one consumer for each ring buffer. As a result, the additional synchronisation overhead is reduced, but the system model is greatly simplified.

2.5 Batch processing

The system supports writing data in batches to reduce the latency by reducing the number of times copying data to the shared memory.

2.5.1 Usage

Since the producer alone cannot determine when the consumer will retrieve data, one way is to use an infinite loop and keep checking whether all data has been delivered. In the loop, the producer obtains the next available index and the max available index by calling ClaimIndex and get_max_write_index. Two variables, num_of_data_written and data_to_write are created to track how much data has been written to the shared memory and how much data can be written in the current iteration. Next, the producer calls the Set_data function to write data. The function takes three arguments: the next available index, the data's starting address, and the data's length. Finally, the producer commits.

2.5.2 Crux

The crux of implementing batch processing is that, although there is enough space in some scenarios, the data cannot be directly put into the ring buffer. Directly storing data of size three could result in a segmentation fault. This is illustrated in Figure 2.3, the following available index is two, and the largest index that could be written is 4. In order to correctly handle the situation, the data is divided into two parts, head and tail. The head part is stored from the next available index till the end of the array, and the tail part is stored from the beginning of the array, as illustrated in Figure 2.4. The implementation ensures that consumers can fetch the data one by one with the order maintained.

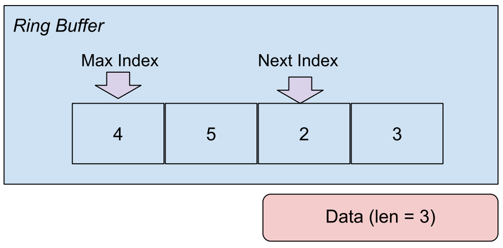


Figure 2.3: A scenario where data does not fit into the ring buffer

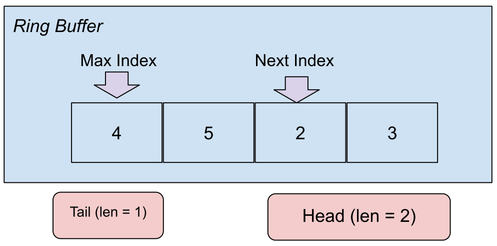


Figure 2.4: Dividing and saving the data

Chapter 3 Implementation

3.1 Ring Buffer

The core data structure of this project to deliver messages is a ring buffer. A ring buffer is a data structure connected end to end where an element is added to the beginning of the buffer when the cursor reaches the end of the buffer. This data structure has a first-in-first-out data characteristic [4]. This data structure is useful for transmitting data between asynchronous processes as data does not need to be reshuffled; instead, only the cursors are adjusted.

In the implementation, the underlying data structure used to store elements is `vector` from the standard library. There are two cursors in a ring buffer: the index of the last data that a consumer can read from and the index of the next write operation. It is required for developers to specify the capacity of the buffer. The capacity of a ring buffer is constrained to the power of 2. The reason is that the `&` operator can be utilised to calculate the index of an element when the size is the power of 2. There is a very high cost for the remainder calculation on the sequence number on most processors; our solution reduces this cost significantly. Also, the `[]` operator is overloaded: `ring_buffer[index] = buffer[index & (capacity - 1)]`. Through this, developers can get the data of index I without understanding the indexing of a circular buffer. The major difference between this implementation and the standard Disruptor lies in the shared memory. Consecutive memory addresses are allocated to ring buffer due to the inter-process nature. When a Disruptor is initialised, a ring buffer is created for it. There is also a shared memory manager responsible for managing shared memory. Thus, the ring buffer can be simply attached to a piece of shared memory using the shared memory manager. Pre-allocation for the memory for the ring buffer is done at initialisation [12].

3.2 Compiler Choices

It occurred to us that compiler choices may make a difference in the speed of a program, and it is important to test code performances using different C++ compilers before making decisions. The two most used compilers are introduced for testing: g++ and clang. However, it turned out that compilers do not affect code performance much. So we left CMake to make the decision. CMake is used to compile the source code where CMake automatically finds a suitable compiler on the user's computer.

Apart from compiler choices, optimisation levels in a compiler are vital to the performance of programs. These levels include `-O2`, `-O3` and `-Ofast`. During our research, it is found that `-O3` will not reduce the size of generated code and `-Ofast` completely breaks the floating-point math [1]. `-O2` is the ideal one as it usually reduces the size of the generated code, generates more warnings, and it has a better performance compared to other optimisation flags. Figure 3.1 shows the writing speed of `-O2` and `-Ofast`, it is easy to see that writing using `-O2` is faster than `-Ofast` and it has a smaller variance. In conclusion, `-O2` is used as the optimisation flag in this project.

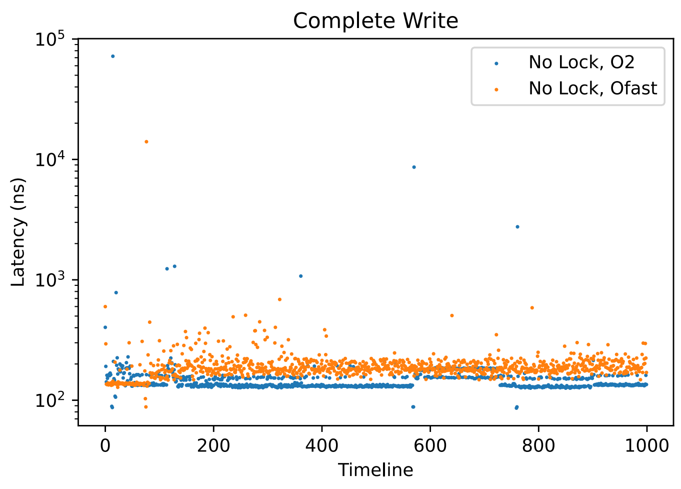


Figure 3.1: Comparison of -O2 and -Ofast

3.3 Benchmarking

As indicated earlier, achieving low latency is crucial for the messaging system. Accuracy and precision to a nanosecond level are hence compulsory to the measurements being made, which guides and evaluates optimisations applied iteratively. This subsection focuses on the specific techniques that comprise a reliable benchmarking tool.

3.3.1 Consideration of Existing Tools

Before constructing a custom benchmarking suite, several existing frameworks were investigated and attempted. However, they are either not sufficiently flexible in design. Leveraging them for the message bus requires considerable efforts or introduces excessive performance overhead, preventing us from measuring the system latency at a nanosecond level.

In particular, Benchmark by Google is a generic microbenchmarking framework that provides a clean interface defined by a C/C++ header file. Unfortunately, Benchmark gave varying latency results among measured samples, where the numerical gaps between them are even larger than the one to be measured. Nonetheless, some other benchmarking frameworks were explicitly designed for other applications. Leveraging those would bring unnecessary overheads due to additional software abstraction. Developing a lightweight benchmarking mechanism dedicated to the system would then be sensible.

3.3.2 High-resolution Timers

Timers that measure execution time with the granularity of nanoseconds and bring little latency overhead by invoked were sought. Some efforts were put into selecting the appropriate timers to use. Works [14] by Aditya et al. narrow the search space down into RDTSC and a Linux function `clock_gettime()`, which outperform others in terms of precision and variance.

RDTSC stands for "Read Timestamp Counter". There is a timestamp register in x86 architectures that hold the processor clock cycle count, whose value can be read by an x86-64 assembly instruction `rdtsc`. This approach should give sufficiently fined-grained results and introduce minimal latency expense. Despite that, conversion from processor clock cycles to time value in nanoseconds is required, which is hard for processor cores with dynamic clock rates. Besides, the values read from multiple cores may not be consistent and even relevant. Furthermore, RDTSC is only available on x86 machines.

`clock_gettime()` is another nanosecond-scale timer provided by Linux, which offers the elapsed time since the system boots. It offers multiple types of timer values. Using `CLOCK_MONOTONIC` guarantees the timer value to increase monotonically, regardless of process blocking or processor core switching. This approach works perfectly on Linux, which was chosen for the rigorous testing environment of benchmarking.

However, macOS has a different implementation of the `clock_gettime()`, which rounds the timer values to the nearest $1,000$ nanoseconds. Even worse, Windows does not offer this particular timing mechanism. For development, an external timing tool called `rchnanotimer` is hence adopted in combination with `clock_gettime`, depending on the type of OS executing. Although `rchnanotimer` has a slightly coarser resolution in comparison to the former, it is sufficient to support benchmarking during normal development and debugging.

3.3.3 Measuring Approaches

There are two sets of system latency baselines that were frequently referred to during development and optimisation – the read-and-write latencies of the `shm` primitive as the ideal values and the ones of the initial prototype system that was developed in early iterations. The aim was to manipulate the internals of the existing system to yield new benchmarking latencies smaller than the prototype measurements while as close to the ideal one as possible.

Since `shm` employs virtual memory mapping tricks to share memory data between processes, the data written by a producer process are immediately available to any consumer process. The software latency of physically writing in and reading out the data on the messaging data structure and process scheduling was thus measured. In order to study the subtleties of the messaging system during runtime, the latency samples were collected individually rather than in aggregation.

The above was done by inserting timer-calling functions surrounding the invocations of `Disruptor` that are required to perform reading or writing in both a benchmarking producer and consumer. Elapsed Time values are calculated and outputted to `stdout` at the ends of benchmarking iterations.

As the client suggested, in order to prove the robustness of the messaging bus in more real-world scenarios, burst writing and reading were involved in benchmarking. The producer process periodically sleeps for a specific period during runtime, while the corresponding consumer process remains unchanged.

Moreover, batch writing as an extended feature of the messaging system is benchmarked separately. Given that the producer process writes multiple data entries within one set of mandatory calls to the `Disruptor`, throughput (i.e. $\frac{\#Messages}{Unit\ time}$) was used to indicate the system performance in batch processing mode.

3.3.4 Isolated Environment

While benchmarking the messaging bus on a modern computer system, noise could affect the result's accuracy and precision. For example, background threads share the same processor core with the target thread, periodic or accidental context switches of processes, or runtime migration to another core of the target process. Therefore, the following operational techniques were applied in the rigorous benchmarking environment.

Processor Core Isolation

Whenever the target process (either a producer or a consumer) is being benchmarked on a physical core, it should be ensured that no other process is scheduled on the same core. Otherwise, the timer could accidentally consider the execution time of other processes. On Linux, adding `GRUB_CMDLINE_LINUX="isolcpus=N,M"` to the system file `/etc/default/grub` and rebooting the machine disable any process to be scheduled onto the specified CPUs `N, M`, unless being explicitly told so. It isolates the individual processor cores from the rest of the computer system during normal execution.

CPU Affinity

After the above configuration, the isolated cores can be assigned as dedicated to benchmarked processes. Linux offers the `taskset` utility to pin processes to specific CPU cores. Since no other processes can use the cores they execute on, issues like process context switches can be eliminated.

Hyper-threading Off

Merely to reassure that only one thread is active for each of the isolated cores, it is desirable to disable the hyper-threading feature of the processor cores explicitly. This can be done by writing `0` to the system file `/sys/devices/system/cpu/cpuX/online` for an exerted core `X`. However, the testing server physically does not support hyper-threading, so this configuration can be skipped.

3.4 Cache Optimisation

The cache prefetching technique [13] is applied on the consumer side. Cache prefetching refers to a speedup technique that fetches the most likely accessed instruction or data to the cache for future use. Since the Disruptor always returns the largest position that the consumer can read, the Disruptor prefetches the value from the index position that the consumer entered to the maximum value that can be read into the cache. This approach marginally increases the latency. One possible reason is that the efficiency of the custom prefetching technique is not as high as the internal prefetching of the CPU. However, the latency variance is reduced significantly, which is a desirable trade-off.

Furthermore, this approach aligns frequently used data by applying the built-in function in C++ [10]. Data across two cache lines means two loads or two stores operations. If the data is cache line aligned, it is possible to reduce one read or write operation, thus reducing the latency.

3.5 Automated Testing

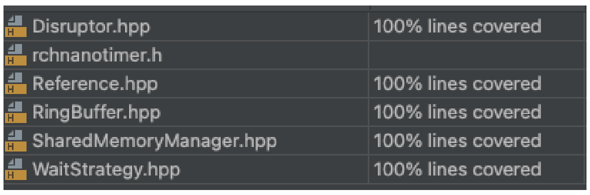
This section introduces techniques and results of testing adopted in development.

3.5.1 Continuous Integration (CI)

Continuous integration (CI) was adopted during development. A GitLab-ci.yml file was added to the root of the repository. Every time new commits are merged into the master branch, a pipeline consisting of building and running automated tests will be invoked, ensuring that every time new commits are merged, new changes will not break the system.

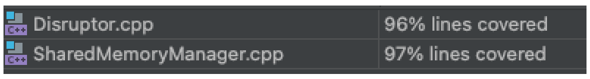
3.5.2 Testing Results and Code Coverage

The whole system is tested using the Google Test framework [5]. There are 20 tests in total, including both unit and integration tests. The test coverage is illustrated in Figure 3.2 and 3.3. It could be found that all header files are thoroughly tested, and more than 95% of source files are tested.



Disruptor.hpp	100% lines covered
rchnanotimer.h	
Reference.hpp	100% lines covered
RingBuffer.hpp	100% lines covered
SharedMemoryManager.hpp	100% lines covered
WaitStrategy.hpp	100% lines covered

Figure 3.2: Test coverage for all header files (rchnanotimer.h is a third-party library)



Disruptor.cpp	96% lines covered
SharedMemoryManager.cpp	97% lines covered

Figure 3.3: Test coverage for all source files

3.6 Software Engineering Practises

The project followed an agile scrum methodology. Sprint planning is done at each iteration with the supervisor to agree on the goals. Tickets are created and tracked on a Kanban board. Many catch-up meetings are carried out throughout the week to ensure nobody is blocked and report progress. This helps maintain the project's momentum and ensures accountability between team members. At the end of each sprint, a sprint review with the client is carried out to present results and receive feedback contributing to next week's sprint plan. From a development perspective, Gitlab for source code management. Each developer works on their tickets on separate branches and creates a merge request upon completion of the ticket. At least two other developers need to review the code and approve it to be merged on the main branch. This ensured high code quality and shared knowledge within the team. In contrast with other projects, the goal is to improve system performance rather than build new features. Hence, iteration cycles tend to be a cycle of development and benchmarking and constantly trying to reduce latency or increase stability and reliability.

Chapter 4 Evaluation

4.1 Comparison of Iterations

The project has been optimised several times through iterations. The most significant optimisations have been removing the locking mechanism from the implementation, utilising cache prefetching technique and isolating CPU cores. This section will compare the performances of iterations.

The first column in Table 4.1 and Table 4.2 is pure shared memory communication without modification or implementation. Since there is no overhead in pure shared memory communication, it serves as the baseline. Other columns are the performances of previous iterations where our implementations, such as Disruptor and ring buffer, were added. It is evident from the tables that the first iteration, which involves locking, is the slowest regrading write and read latency. The reason is that locking is a time-consuming operation that requires context switching when there is a blocked process. After the project is optimised as lock-free, the read and write latency both drop significantly, from an average of 4093.2ns to 152.6ns. The cache prefetching technique helped stabilise the latency as the standard deviation is much lower than other versions. Though its mean increased by a few nanoseconds, it was a positive trade-off.

Stats. \ Versions	shm Primitive	Original (with lock)	Lock Free	Cache Prefetching
Mean	108.315	4093.222	152.558	159.709
Min	89	94	87	108
Median	86	2265	131	159
90th Percentile	113	10111	163	170
99th Percentile	114	22290	207	176
Max	316	77038	11505	296
Standard Deviation	88.649	5683.473	376.842	119.641

Table 4.1: Write Latencies for Different Iterations (in Nanoseconds)

Stats. \ Versions	shm Primitive	Original (with lock)	Lock Free	Cache Prefetching
Mean	83.883	3031.892	85.001	91.879
Min	75	86	68	72
Median	79	2568	86	87
90th Percentile	80	9756	87	91
99th Percentile	99	20048	89	92
Max	4810	74964	316	158
Standard Deviation	149.662	5364.514	8.529	5.817

Table 4.2: Read Latencies for Different Iterations (in Nanoseconds)

4.2 Benchmarking in Isolated Environment

Optimised through the above iterations, the ultimate version of the IPC solution was subsequently benchmarked rigorously on the testing server. The configuration of the isolated environment follows the techniques described in Chapter 3.3.4. The system latencies were measured using a standard procedure (Table 4.3), where messages are sent one by one and under a burst writing scenario (Table 4.4). Besides, the system throughput on batch writing mode was collected, resulting in the numbers in Table 4.5.

For the measurements in Table 4.3, it is clear that the results generated in a clean environment are significantly better and more stable than the last columns in Table 4.1 and 4.2. It reflects a more real-world low-latency use case, where software executes on its dedicated hardware to interference from other processes. The system gives a latency close to the average for even two-nine percentile, implying high-performance predictability. The max values are due to cache warming that only occurs at the beginning of the executions. Regarding the burst messaging results in Table 4.4, the higher percentiles of writing latency turn out to be larger than the standard situation. It is caused by cache warming at the beginning of each burst of writing. The client suggested a solution: the application could spin or send empty data instead of blocking the process while there is no incoming message, keeping the cache warm. No substantial impact is made to the consumer latency in the burst scenario. Furthermore, the message bus appears to provide high throughput, as indicated in Table 4.5. However, the receiver halves the sender's bandwidth since only the latter was optimised for the batch messaging mode.

Stats. \ Type	Write	Read
Mean	66.374	28.961
Min	60	25
Median	66	30
90th Percentile	69	31
99th Percentile	72	31
Max	309	241
Standard Deviation	11.553	7.073

Table 4.3: Write and Read Latencies of the Final Version on the Testing Server (in Nanoseconds)

Stats. \ Type	Burst Write	Read
Mean	189.785	29.674
Min	62	23
Median	71	29
90th Percentile	175	97
99th Percentile	210	146
Max	26145	433
Standard Deviation	1335.710	39.995

Table 4.4: Burst Write and Read Latencies on the Testing Server (in Nanoseconds)

Metric \ Type	Write	Read
Throughput (messages per second)	1.385×10^8	6.159×10^7

Table 4.5: Burst Write and Read Throughputs on the Testing Server

4.3 System Profiling

To understand which components of the system contribute to most execution time, basic profiling was performed by a built-in profiler on macOS – Instruments. The profiler periodically inspects the call stack during profiling to determine which regions of code (i.e. functions) are being processed and increment the corresponding timers or counters. After a sufficiently long period of runtime, it terminates and estimates the percentages of execution time for all functions involved from the collected sample.

Figures 4.1 and 4.2 illustrate the profiling results of the best-performing version of the messaging system. In those charts, the inner pies represent the runtime portions of higher-level functions, the APIs exposed to the user. The outer rings break them down into finer-grained percentages of function invocations inside the system.

For producer, `Disruptor::Commit` and `Disruptor::ClaimIndex` dominate, which are responsible for maintaining the system metadata to enable being read by the consumer. The function `Disruptor::SetData` that physically write the incoming data into the data structure only takes up a small sector. It does not directly mean that the overhead of maintaining the messaging data structure is large. Considering

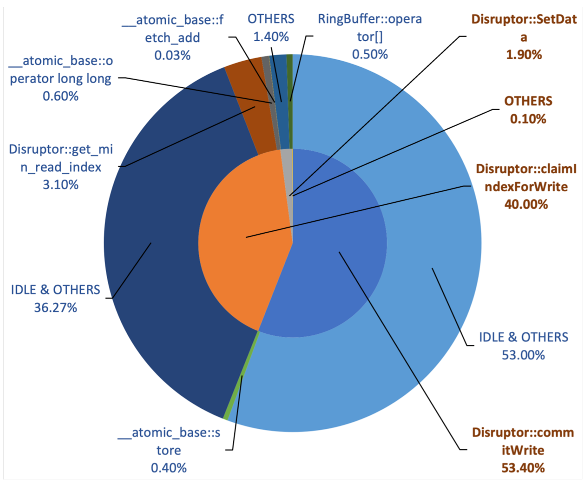


Figure 4.1: Profiling of Producer Process

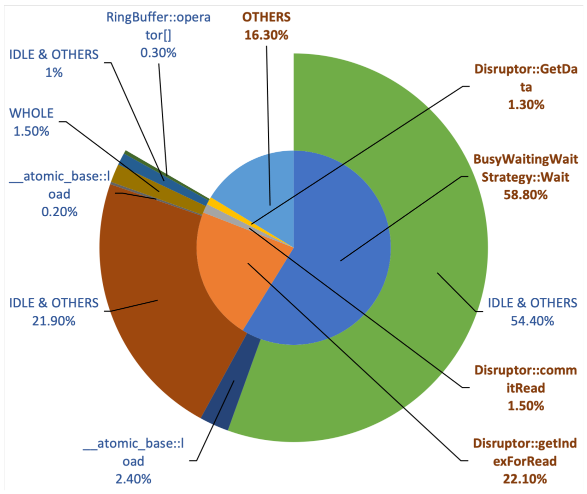


Figure 4.2: Profiling of Consumer Process

the outer ring in 4.1, most of the percentages were spent as process idle time, which was led by the necessary interaction between the process and consumer processes. More specifically, whenever the producer fills the shared memory space, it spins until the consumer reads previous entries to proceed. Increasing the buffer size of the `Disruptor` or the number of concurrent consumers could potentially help reduce the idle time.

The consumer may occasionally wait for the producer to write new data for reading. The API functions `BusyWaitingWaitStrategy::Wait` and `Disruptor::WaitFor` therefore occupies a large execution period in Figure 4.2.

4.4 Objective Evaluation

Table 4.6 summarises the results of objectives mentioned in section 1.1 as well as how they get verified.

4.5 Client Feedback

The client, qSpark, has been able to follow the development process of the IPC mechanism and have provided invaluable feedback for potential optimisations to be made based on their professional experience in the area. The qSpark has been supportive throughout the development process and satisfied with the project outcomes given the major pivot and limited time and experience the development team had. This project has successfully demonstrated the potential of IPC and a likely to be used by the client. However, further work is still required if it were to be integrated into their systems. Some advanced optimisation techniques can still be explored if the client chooses to continue working on this project and integrate it into their HFT strategies.

Objective ID	Verified By	Result	How
1	Testing	Fulfilled	The system passed a test where a consumer can read the data posted by a producer.
2	Testing	Fulfilled	The system passed a test that involves multiple producers and consumers writing and reading data.
3	Benchmarking	Fulfilled	The mean writing latency is 66ns surpassing the 100ns target.
4	Testing	Fulfilled	The system passed a test where a producer can send a 40-byte char array to a consumer.
5	Testing	Fulfilled	Same as 4.
6	Benchmarking	Fulfilled	The std. of writing latency is under 11.553ns below the 15ns required.
7	Benchmarking	Fulfilled	The producer can achieve high throughput in the batch writing mode supporting 1.385×10^8 messages per second.
8	Benchmarking	Fulfilled	The 99th percentiles of writing and reading latencies are closed to their means with 72ns latency.

Table 4.6: Objective Report

Chapter 5 Conclusion

Overall the project has been quite successful despite the setbacks of the pivot. All the objectives set out have been met, and further improvements have been made beyond the initial objectives. The final IPC mechanism can run at consistently low latency with low variance and few outliers. The client was impressed with the results, although there is still room for improvement.

5.1 Future Extensions

Performance-wise, more in-depth and advanced profiling can be carried out to identify areas of improvement. More advanced low-level optimisations can still be explored with the system to improve the latency and stability. The burst performance of the system is not the best due to cache miss issues. This can be mitigated by implementing cache warming techniques during runtime to increase stability.

The solution is intended to be used as a C++ library, so developer experience is essential. The library has been designed with usability in mind. Throughout the project, tweaks have been added to improve usability. For instance: shared key management and API simplification. Nevertheless, there are aspects such as error handling, interrupt handling and flexibility of parameters of the API that are strong candidates for future improvement. All these aspects are essential to facilitate the adoption of the library and maximise utility to the end-user.

References

[1] David Branco and Pedro Rangel Henriques. "Impact of GCC Optimization Levels in Energy Consumption During Program Execution". In: *Acta Electrotechnica et Informatica* 16 (Mar. 2016), pp. 20–26. DOI: 10.15546/aeei-2016-0004.

[2] Johannes Breckenfelder. "Competition among high-frequency traders, and market quality". In: (2019).

[3] Carl Cook. *When a microsecond is an eternity: High performance trading systems in c++*. 2017. URL: https://isocpp.org/blog/2018/09/cppcon-2017-when-a-microsecond-is-an-eternity-high-performance-trading-syst.

[4] Steven Feldman and Damian Dechev. "A Wait-Free Multi-Producer Multi-Consumer Ring Buffer". In: *SIGAPP Appl. Comput. Rev.* 15.3 (Oct. 2015), pp. 59–71. ISSN: 1559-6915. DOI: 10.1145/2835260.2835264. URL: https://doi.org/10.1145/2835260.2835264.

[5] Google. *GoogleTest - Google Testing and Mocking Framework*. URL: https://github.com/google/googletest.

[6] Morteza Kashyian, Seyedeh Leili Mirtaheri, and Ehsan Mousavi Khaneghah. "Portable Inter Process Communication Programming". In: *2008 The Second International Conference on Advanced Engineering Computing and Applications in Sciences*. 2008, pp. 181–186. DOI: 10.1109/ADVCOMP.2008.38.

[7] Leslie Lamport. "On interprocess communication". In: *Distributed computing* 1.2 (1986), pp. 86–101.

[8] Serge Leef. *Real Time Machine Learning (RTML)*. URL: https://www.darpa.mil/program/real-time-machine-learning.

[9] Laure Madonna. *The Ethics of High Frequency Trading - Seven Pillars Institute*. Apr. 2013. URL: https://sevenpillarsinstitute.org/wp-content/uploads/2017/11/Laure-Madonna-High-Frequency-Trading.pdf.

[10] Chris B. Sears. "The Elements of Cache Programming Style". In: *Proceedings of the 4th Annual Linux Showcase Conference - Volume 4*. ALS'00. Atlanta, Georgia: USENIX Association, 2000, p. 18.

[11] Shobhit Seth. *The World of High-Frequency Algorithmic Trading*. Jan. 2022. URL: https://www.investopedia.com/articles/investing/091615/world-high-frequency-algorithmic-trading.asp.

[12] Martin Thompson et al. *Disruptor: High performance alternative to unbounded queues for exchanging data between concurrent threads*. https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf. 2011.

[13] Steven P. Vanderwiel and David J. Lilja. "Data Prefetch Mechanisms". In: *ACM Comput. Surv.* 32.2 (June 2000), pp. 174–199. ISSN: 0360-0300. DOI: 10.1145/358923.358939. URL: https://doi.org/10.1145/358923.358939.

[14] Aditya Venkataraman and Kishore Kumar Jagadeesha. "Evaluation of interprocess communication mechanisms". In: *Architecture* 86 (2015), p. 64.

Appendix A API Documentation

bool InitRingBuffer(int size);

Initializes the ring buffer of the given size. The size should be the power of 2, in order to easily calculate the modular of the index by **&** operator. It calls **CreateShMem()** in **SharedMemoryManager** class to create the shared memory if there is no shared memory available. The shared memory should include both header and ring buffer. The header will record the information about the shared memory, such as the size of the shared memory, the number of attached processes etc. After the shared memory is created or if it is already available, the calling process will be attached to the shared memory. Then the process can access the header and the ring buffer by **ring_buffer_status_on_shared_mem_** pointer and **ring_buffer_** array respectively. Finally, the blocking strategy will also be initialized.

Parameters:

int size: The size of the ring buffer. It should be the power of 2.

Return value:

bool: The result of the initialization. **true** for success, **false** for unsuccess.

void ResetRingBufferState();

Reset all fields in the **ring_buffer_status_on_shared_mem_** to initial status. This function should be called only if the shared memory is created for the first time.

bool TerminateRingBuffer();

Detaches the process from the shared memory. Releases the shared memory if it is the last process attached to the shared memory.

Return value:

bool: The result of the termination. **true** for success, **false** for unsuccess.

bool SetData(int64_t index, OneBufferData* data);

Sets data to the specified place of the ring buffer by **memcpy()**.

Parameters:

int64_t index: The index of the ring buffer, where data should be written on.

OneBufferData* data: The data, producer wishes to write into the ring buffer.

Return value:

bool: The result of the data copying. **true** for success, **false** for unsuccess.

OneBufferData* GetData(int64_t index);

Get data from the ring buffer by the specified index.

Parameters:

int64_t index: The index of the ring buffer, where data should be read from.

Return value:

OneBufferData* : Data reading from the ring buffer.

bool RegisterConsumer (int id, int64_t* index_for_customer);

Registers the consumer to the shared memory. The registry including incrementing the counts of consumers, checking the capacity of the consumer array, setting consumer index to the readable place, updating the consumer index.

Parameters:

int id: The id number of the consumer. The consumer will be registered to the consumer array of index id.

int64_t* index_for_customer: The index of the ring buffer where the consumer wants to read data from.

Return value:

bool: The result of the registry. **true** if success. **false** if the number of consumers exceed the capacity of the consumer array.

int64_t claimIndexForWrite(int caller_id);

When the producer wants to write a new data in the ring buffer. The producer first needs to claim an index position to write on. This function will fetch the next available index pointed by the **next** pointer. If consumers have read the data from this index, this index will be returned to the producer. Otherwise, the claiming process will be blocked until the data has been read.

Parameters:

int caller_id: The id of the producer which wants to claim an index.

Return value:

int64_t: The available index for writing.

bool commitWrite(int user_id, int64_t index);

After the data is written to the ring buffer, the producer needs to commit the writing to update the **cursor**. This function ensures that the writing is committed sequentially by using some blocking strategies. For example, if multiple producers commit at the same time, the writing whose index is 1 greater than the cursor will be firstly committed.

Parameters:

int user_id: The id of the producer which wants to commit the writing.
int64_t index: The index of the ring buffer which was written to.

Return value:

bool: The result of the commitment. **true** for success, **false** for unsuccess.

int64_t getIndexForRead(int user_id, int64_t index);

Reads data from the specified index of the ring buffer for the consumer. If there is no available data, the reading will be blocking. Otherwise, the latest available index will be returned to the consumer.

Parameters:

int user_id: The id of the consumer which wants to read data.
int64_t index: The index of the ring buffer which consumers want to read from.

Return value:

int64_t: The latest available index for reading.

bool commitRead(int user_id, int64_t index);

Updates the index in the consumer array for specified consumer to commit the reading.

Parameters:

int user_id: The id of the consumer which wants to commit reading.
int64_t index: The index of the ring buffer which the consumer has read from.

Return value:

bool: The result of the commitment. **true** for success, **false** for unsuccess.

int64_t get_min_read_index();

Gets the smallest index of the data read by the consumer, in order to not overwrite the data where has not been read by the consumer. The producer will use this function to claim the index to write data.

Return value:

int64_t: The minimal index of the data read by the consumer.