The University of Hong Kong

Faculty of Engineering

Department of Computer Science

COMP7704
Dissertation Title
High-throughput and Low-latency Trading System Using Java

Submitted in partial fulfillment of the requirements for the admission to the degree of Master of Science in Computer Science

By
Yuen Siu Hung

Supervisor's title and name: Dr. Ronald H.Y. Chung
Date of submission: 01/05/2019

Abstract

This dissertation project studies the practices in developing high-throughput and low-latency electronic trading systems using Java and investigates how to minimize the overheads in inbound and outbound messages in the Financial Information eXchange (FIX) protocol. We developed a trading system from scratch without using any external libraries. Our trading system consisted of an order matching engine and a FIX protocol gateway. We tried several implementations of order list and market data stores in the order matching engine, and we compared the performance of the gateway with that of the popular opensource library QuickFIX/J. We benchmarked the engine, the gateway and the overall trading system in terms of latency, throughput and garbage collection statistics. In all benchmarks, we used two Java Virtual Machine for comparisons. With the settings we proposed, the throughput of our overall trading system achieved 130,236 orders per second and the latency was 7.678 microseconds per order. Our order matching engine individually achieved 491,344 orders per second as its throughput and 2.035 microsecond per order as its latency. Our FIX protocol gateway achieved much lower latency than QuickFIX/J. Our gateway only spent 1.867 microseconds in generating each execution report, while QuickFIX/J spent 3.969 microseconds per each execution report.

Declaration

I, Yuen Siu Hung, declare that this dissertation project “High-throughput and Low-latency Trading System Using Java” thereof represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Content

1. Background

This dissertation is an industrial based project working together with Global eSolutions (HK) Limited.

Nowadays electronic trading is the main stream trading method in the financial world. Many financial markets now choose to only provide electronic trading platforms and no longer provide pit trading to commodity traders [1]. One of the most attractive features of electronic trading to investors, exchanges and other market participants is the significant lowered costs. Electronic trading does not involve any floor brokers who were the key players for pit trading, so it lowers labor costs and thus commission fees. Another most attractive feature of electronic trading is the execution speed. The electronic trading enables investor to place orders to exchanges in the world within a fraction of seconds. This also allows them to trade global products and react to the ever-changing market conditions rapidly in anywhere using computers or mobile devices. As electronic trading provides faster trade executions, it makes day trading, which is difficult in pit trading, more and more popular [1]. Electronic trading also greatly minimizes the difficulties in supervising for rules and regulations. This is especially important in these years as number of rules and regulations is increasing after financial crises. The increasing speed of electronic order matching systems also has been proved that it has positive effects on efficiency of financial markets [2].

Efficiency, effectiveness and market integrity are the key elements of electronic trading [3]. To achieve these elements in trading system level, capacity, consistency and predictability are the three attributes software developers should concern and target to optimize.

Electronic trading system is somewhat simple: it is about sending and receiving messages and doing internal processing. However, what programming language should be chosen to implement it is always in dispute. Traditionally, trading systems were implemented on C, C++, GPU [4] or FPGA [5] to squeeze every ounce of performance of hardware. While they have high performance, the difficulties in programming are high and the development time are long.

Java, one of the most popular programming languages, has been used in many low latency applications to get benefits from its strengths and its big community [6]. Java is commonly known by many programmers. There are many well-developed libraries and deployment tools available. Java frees developers from writing code to manage memory and frees them from writing code checking for detecting run time errors. Therefore, Java is easy to write with and thus its development time can be short [7].

2. Objective

In capital markets, data-latency-race has been a word-wide movement to eliminate geographical, technological and psychological obstacles to increase fairness and transparency of markets [8]. Quantitative traders and statistical arbitrageurs, who works for major brokerages and hedge funds, compete to get their orders reaching market places first in order to trade financial instruments with the most favorable prices. The strong demand of low latency order executions in Wall Street does not only obsolete trading pits, but also creates opportunities for new Alternative Trading Systems (ATS) and Electronic Communication Networks (ECN) to compete with established exchanges and gives rise to new execution service vendors and system integrators who specialize in providing fast order transaction services or infrastructure.

Global eSolutions (HK) Limited (GES) is a financial trading platform vendor serving for the financial technology industry over 10 years [9]. Its customer base covers Hong Kong, United Kingdom, Japan, Malaysia, Indonesia, Australia, etc. In particular, GES TX is one of the trading solutions GES offers. It provides functions of both A Book and B book. The A Book function is used by brokers who route clients' orders to liquidity providers or multilateral trading facilities [10]. The B book function is used by market maker brokers who take the opposite side of their customers' orders and keep the orders internally. The programming language used in GES TX is Java.

To further enhance GES TX system, this dissertation project studies the best practices in developing high-throughput and low-latency trading systems using Java and investigates how to minimize the overheads in inbound and outbound messages.

3. Literature Review

3.1. Java Platform and its Limitations

In 1990s, Java was developed at Sun Microsystem, which was acquired by Oracle Corporation on 2009. The objective of developing Java was to make program functionable independent of the device they ran on and the slogan of Java was "Write once, run anywhere".

Java applications are compiled to byte-code which is in an intermediate, portable format. The Java byte-code programs are executed on a virtual machine called Java Virtual Machine (JVM). While Java byte-code programs remain the same across all hardware and operating systems, implementations of JVM depend on hardware and operating systems.

There are various implementations of JVM from different parties. In this dissertation, Oracle HotSpot JVM and Eclipse OpenJ9 JVM are studied. Oracle HotSpot JVM is maintained by the Oracle Corporation. The project is under GNU General Public License (GPL) version 2. It is the most popular JVM for desktop computers and servers. It has the client version and the server version. The client version is tailor-made for fast start-up and the server version loads longer time to produce highly optimized Just-In-Time compilations to achieve higher performance. Eclipse OpenJ9 JVM is an open source project mainly contributed by IBM. The project is under Eclipse Public License 2.0 and Apache License 2.0 [11] [12]. To migrate to OpenJDK with Eclipse OpenJ9, developers do not need to change their Java applications. Eclipse OpenJ9 utilizes the main components of the Eclipse OMR project which is also contributed by IBM. Fast startup time, small memory usage and high peak performance are important features to

cloud native workloads and desktop applications [13]. When comparing to Eclipse OpenJ9 JVM with Oracle HotSpot JVM, Eclipse OpenJ9 JVM has a 42% faster startup time and a 66% smaller memory footprint [14]. Eclipse OpenJ9 JVM supports Java releases from Java 8 to Java 11.

3.1.1. Just-in-time compilation

There are two possible ways to run byte-code in JVM: by using interpreter at run time or by using just-in-time compilation (JIT) at load time and run time [15]. Using interpreter to interpret byte-code is slower than running compiled native code. JIT is to compile byte-code into native code to directly run it on hardware to improve performance. However, the compilations on the fly costs a penalty to initial performance, so JIT compilers are designed to only compile frequently-called code.

JIT compiled code can be faster than static native code as JIT compilers can collect run time statistic to perform optimization, like in-lining of library methods and reordering of execution instruction, globally at run time [16] while static compilers are impossible to do that. All modern JVM implementations utilize JIT compilation approach to achieve the performance of native code after the initial startup phase of programs.

There are two types of JIT compilers, client compiler and server compiler. Client compiler and server compiler are also called C1 and C2 respectively. Client compiler compiles code earlier than server compiler does, so that client compiler can run code faster than server compiler in the early stage of program execution. Server compiler observes longer in the beginning to collect more statistics to perform better global optimization.

Since Java 7, tiered compilation feature is implemented in JVM and it allows server compiler to work together with client compiler to have faster performance in beginning

stage of execution. Tiered compilation involves 5 levels of executions, level 0 to level 5. Level 0 is interpreted code. Level 1 is execution of C1 compiled code without profiling. Level 2 is execution of C1 compiled code with light profiling. Level 3 is execution of C1 compiled code with full profiling. Level 4 is C2 compiled code using profile data collected in level 3. The normal execution path is level 0 to level 3 then to level 4. There are three special cases. Firstly, level 1 is used when the method to be compiled is simple. Secondly, when the compilation queue of C2 is full, code will be taken from the queue and will be compiled at level 2 and level 3. When the C2 is less busy, the code will be compiled by C2. Thirdly, if C1 is busy, level 3 will be skipped and the code will be passed from level 0 to level 4.

The limitation of JIT compilation is the cache size of compiled code. JVM caches all compiled code in memory, but the size of the cache is limited. If the cache is full, compilers will stop compiling byte-code to native code and the program will start to run lots of slow interpreted code. The initial and maximum size of the code cache can be set by *-XX:InitialCodeCacheSize* flag and *XX:ReservedCodeCacheSize* flag respectively.

3.1.2. Garbage collection

Garbage collector is one of the limitations of the Java platform that increase the difficulties in achieving ultra-low latencies. Memory in JVM is fully managed by the system itself that objects cannot be deallocated explicitly by code. JVM uses a component, called Garbage Collector, to find out objects that are not in use and reclaim their memory. In a Java program, there are two groups of thread: one group working for application logic and another group working for garbage collections.

Most of the collection mechanisms used in collectors involve generations of pauses to

application logic to make sure objects are not used by the application threads. Pauses that halt all application threads are called stop-the-world pauses. Minimizing them is one of the key considerations when tuning performance of a Java program. Generational garbage collectors split heap space into young generation and old generation. When young generation fills up, minor garbage collection, which has relatively fast and short stop-the-world pause, will occur. When old generation fills up, major garbage collection, which has relatively long stop-the-world pause, will occur. Major garbage collection should be avoided.

A. Garbage Collectors in Oracle HotSpot JVM

There are five garbage collector types in Oracle HotSpot Virtual Machine [17]. They are Serial Garbage Collector, Parallel Garbage Collector, Concurrent Mark Sweep (CMS) Garbage Collector, Garbage-First Garbage Collector (G1GC) and Z Garbage Collector (ZGC). To choose Garbage collector type, need to pass the corresponding option to the JVM when starting up applications.

I. Serial Garbage Collector

Serial Garbage Collector only uses one thread for garbage collections, and it pauses all application threads when it does garbage collection. It is suitable for a simple program in command line. By default, it is enabled on some operating system and hardware, and can be chosen with the option *-XX:UseSerailGC*.

II. Parallel Garbage Collector

Parallel Garbage Collector was the default garbage collector in Oracle HotSpot Virtual Machine before Java 9. It uses several threads for garbage collection to speed up the throughput of garbage collection. It also pauses all application threads when doing

garbage collection. It can be enabled by the option *-XX:UseParallelGC*.

III. Concurrent Mark Sweep Garbage Collector

Concurrent Mark Sweep Garbage Collector (CMS) is designed to have short garbage collection pauses and it shares process resources with the application when the application is running. It utilizes separate garbage collector thread to mark objects concurrently when the application is still running [18]. It has two steps in garbage collection. Firstly, it uses multiple threads to scan and mark unreferenced instances in heap. Secondly, it releases the memory of those instances. In each major collection, Concurrent Mark Sweep Garbage Collector pauses all application threads when marking referenced objects in the old generation space and when a change in the heap occurs in parallel during the garbage collection. Minor collections are executed like the parallel collector that they pause the application threads when they occur. Concurrent Mark Sweep Garbage Collector outperforms Parallel Garbage Collector by using more CPU to provide higher throughput. Concurrent Mark Sweep Garbage Collector does not do memory compaction in concurrent collection cycle [19]. Concurrent Mark Sweep Garbage Collector can be enabled by the option *-XX:+ UseConcMarkSweepGC*. It has been marked as deprecated since Java 9.

IV. Garbage-First Garbage Collector

Garbage-First Garbage Collector (G1GC) is the default garbage collector in Oracle HotSpot Virtual Machine since Java 9 [20]. It is designed for machines with multiple processors and a large memory size [21]. It divides memory space into many equal sized small regions and do garbage collections within those regions in parallel. It collects garbage from the region with the most garbage first. To avoid length of interruptions proportional to the heap size, operations, such as global marking, which

involve the whole heap are performed parallelly with the application threads. It can be enabled by the option *-XX:UseG1GC*.

V. Z Garbage Collector

Z Garbage Collector (ZGC) is a new garbage collector introduced in Oracle HotSpot Virtual Machine in Java 11 [22] and it is designed for low latency. It is announced as an experimental feature and only supports Linux 64-bit operating system. It guarantees that the pause time of each garbage collection is always not more than 10 milliseconds. Increasing heap space or the size of live-set does not increase ZGC's pause times, though increasing the root-set size, i.e. number of threads increases the time. It can handle heaps with few hundred megabytes up to several terabytes in size. Z Garbage Collector limits the impact of garbage collections on the application's response time by doing garbage collections concurrently while keeping application threads continue to execute. Z Garbage Collector compacts heap to reduce fragmentations [23] [24]. The core concept of Z Garbage Collector is the colored pointers. Z Garbage Collector utilizes 4 bits out of the 22 unused bits in 64-bit object pointers to store metadata. Just-in-time compiler will add a small segment of code, which is known as a load barrier, when an object reference needs to be retrieved from the heap. The load barrier utilizes the metadata stored in color pointer to check if it should take actions to correct the object. It is enabled by the option *-XX:+UseZGC*.

B. Garbage Collectors in Eclipse OpenJ9 JVM

1.1.1.1. Generational Concurrent Garbage Collector (GenCon)

Generational Concurrent garbage collector of the Eclipse OpenJ9 JVM is the default garbage collection policy of OpenJ9 JVM [25]. It uses generational garbage collection

and a concurrent mark phase to shorten the time length of garbage collection pauses [26]. It splits the heap into a nursery region and a tenure region. The nursery region is to store new objects allocated. When it is more than 90% full, a nursery collection will occur. The nursery region consists of the evacuate subspace and the survivor subspace. New objects are allocated to the evacuate subspace of the nursery and when the space is filled, objects will be evacuated to the survivor subspace of the nursery. The objects after surviving for certain number of nursery collections will be moved into the tenure region. Global collection, which removes unnecessary objects from the tenured region, will be triggered when the tenure region is full. The nursery garbage collector and the tenured garbage collector pause all application threads when they do collections. This policy is enabled by using the option *-Xcpolicy:gencon*.

1.1.1.2. Metronome Garbage Collector

Metronome garbage collector of Eclipse OpenJ9 is only available in certain hardware and operating system. This policy is designed to provide short and precise pause time by using incremental and deterministic garbage collector. While the standard garbage collection would pause application threads during marking and collecting phase, the metronome garbage collection works in small interruptible steps [27]. Metronome garbage collection policy guarantees applications can run for certain percentage of time in every 60 milliseconds and the garbage collector can run in the remaining percentage of time. There are two types of threads in the Metronome Garbage Collector: a single alarm thread and several garbage collection threads. The alarm thread regularly checks the amount of free heap memory and if any garbage collection is occurring. If not enough free memory and there is no garbage collection taking place, the alarm thread will trigger the garbage collection threads to do garbage collection. After finishing a

cycle of garbage collection, the Metronome Garbage Collector will check again if the free space is still insufficient. If the free space is still insufficient, it will start the garbage collection again. This garbage collector can be enabled by the option -Xgcpolicy:metronome.

1.1.1.3. Balanced Garbage Collector

Balanced garbage collector of Eclipse OpenJ9 is designed for applications which uses more than 64 mega-bytes on 64-bit operating systems. This garbage collector utilizes mark, sweep, compact, incremental and generational style garbage collection. It tries to match object allocation and survival rates to avoid global garbage collection [26]. It splits a heap into many regions which are assigned age numbers ranging from 0 to 24 [25]. This approach is to have regions managed individually to shorten the maximum pause time on large heap space and improve the efficiency of garbage collections. New objects are allocated in Eden Space which is formed by the regions of age 0 or 1. The size of Eden Space is by default 25% of the overall heap. While the regions of ages ranging from 2 to 23 are used as Survivor Space, the regions of age 24 are used as Tenured Space. Survivor Space is responsible to store live objects forwarded from Eden Space. This garbage collector can be enabled by the option -Xgcpolicy:balanced.

The balanced garbage collector utilizes a special representation for arrays. Arrays will be represented continuously in the heap space, like what GenCon garbage collector does, if they are small enough to be contained in a region. Large arrays are represented discontinuously in the heap by a spine that contains the object header and an array of pointers to arraylets. Each arraylet fills a whole region [25].

3.2. Tuning Java programs for High Performance

To achieve ultra-low latency performance of a Java application, developers need to consider many layers that are involved in the program execution chains. They are network, hardware, operating system, JVM and the application itself. This dissertation focuses on software development perspective.

3.2.1. Java Virtual Machine (JVM) tuning

As Java programs are running JVM, the configurations of JVM is important to the performance of the program executions. Configurations related to memory size and those related to garbage collector usually impact the performance significantly as they directly affect the process of garbage collections. Ideally an application should use mostly long-lived objects and creates short-live objects as few as possible to avoid fill up young generation of the heap, so that no garbage collection will occur. However, it is difficult to write programs using this approach. As a trade-off, developers can set the size of the young and old generation of the heap respectively according to the performance target to permit a small number of minor garbage collections to happen and avoid major garbage collections from happening.

3.2.2. Just-In-Time (JIT) compilation tuning

To optimize JIT compilations, developer can set the size of code cache to be large enough that never be full or give hints to JIT compilers to decide when to compile certain code. There are two ways to give hints to JIT compilers in OpenJDK and Oracle JDK: Compiler command file and Annotations.

3.2.3. Design principles and techniques

There are some design principles and implementation techniques to achieve high

performance in Java applications.

3.2.3.1. Keep It Stupid Simple (KISS)

Designing a simple architecture and writing simple code are the keys to produce high performance applications. Simple software can allow developers to understand it easily, so that developers can write higher quality of code and find performance bottleneck of the software easily.

To keep code simple, developers should break down their tasks into sub-task and break down big problems into small problems [28]. They should avoid creating many classes for one problem. They should never write any method with longer than 30-40 lines. If there are many conditions in a method, developers should split it into several small methods.

3.2.3.2. Pre-allocations and re-utilizations of objects

Garbage collection is the key concern in developing high performance Java application. Number of garbage collections is directly proportional to the number of objects allocated. To decrease number of garbage collections, applications should avoid creating unnecessary objects. This can be achieved by pre-allocating a set of objects in the startup phase of a program and re-utilizing those objects throughout the entire run time of the program. Pre-allocated objects are long-lived, and as they are stored in the old generation of the heap space, they will not impact the process of minor garbage collection. Object pre-allocation also improve CPU caching and memory access as data can have higher chance to be allocated in contiguous locations of main memory. Large objects should be avoided as frequent creations of large objects can lead to memory fragmentation issues and memory compaction operations [29].

3.2.3.3. Use of multi-threading

Parallelism can enable effective utilizations of processing power, and thus speed up processing time and increase the throughput. Developers can divide a task into smaller tasks, such as order matching, sending market data to feeders and aggregating data for reporting, and parallelize tasks to run simultaneously [30]. For tasks which require strict sequential executions, like order matching, they should be single-threaded.

However, the number of running thread should not exceed the number of physical cores available. When threads compete for CPU cycle, operating systems will do time-slicing to divide CPU cycles to each thread and this will slow down the application.

3.2.3.4. Use of lock-free approaches

Making an application multi-threaded does not only parallel tasks, but also potentially create competitions between threads for the read and write access to resources. If multiple threads try to write to the same resource at the same time, complicated and costly operations are required to co-ordinate those threads [31].

Locks are traditionally used to control the access of threads to certain portions of code and ensure the updated value of data visible to other threads. However, lock contention limits the scalability and performance of an application due to L2 cache miss and cache-to-cache transfer [32]. According to an experiment of lock conducted by LMAX [31], the time cost of a program goes up significantly after introducing a lock even when the lock is not contended. After adding two or more threads to contend the lock, the time cost increases by orders of magnitude. Lock is slow because it requires a context switch to the operating system kernel for arbitration to suspend other threads which are waiting on the same lock [30]. Context switch does not only add overheads to program execution, but also has two more disadvantages. Firstly, operating system will get the

control of execution during context switch and it probably decides to perform other tasks. Secondly, execution context will likely lose cached data and instructions. Trading system targeting to handle large trade volume of orders within a short period of time should avoid using locks to prevent frequent context switching from happening.

There are several lock-free techniques synchronizing access to resources, such as compare-and-swap (CAS) and busy spins. They do not involve the kernel arbitration and can provide a stable performance up to the saturation point [12].

3.2.3.5. Use of proper data structure

Developers should use data structure in the ways that are friendly to garbage collectors. Developers should initialize standard containers with correct size [33]. For example, HashMap should be sized correctly, as increasing size will lead to rehashing which takes some time. Data structure should be used correctly in the use cases [34]. For example, adding an element to the beginning of an *ArrayList* triggers copying the whole list to a new list. The old list will become garbage and need to be reclaimed by garbage collector.

3.2.3.6. Avoid Disk I/O in main transaction flow

Disk I/O means read and write operations on a physical disk. The speed of disk I/O depends on the performance of the physical disk [35]. When read or write operations occur, CPU must wait for those operations to complete [36]. Disk I/O also involves system calls each of which will lead to a context switch.

3.2.3.7. Avoid capturing state in lambdas

Lambda is efficient, but developers must pay attention when using them. To achieve low latency, developers need to make sure that a lambda does not capture any states

otherwise an allocation will occur to capture that state [37]. It will be fine if lambdas directly apply to data input to them.

4. Methodology

To find out the best practices in developing high-throughput and low-latency trading systems using Java and investigate how to minimize overheads of inbound and outbound message, the first step was to design and implement a trading system. The two major components of a trading system, an Order Matching Engine and a FIX Gateway, were developed in this dissertation by using some design principles and techniques stated in the literature review section. They were written in Java without using any 3rd party libraries, except those for testing, to keep the source code in smaller size, so that bottlenecks could be easily identified. The Order Matching Engine and the FIX Gateway could work together to provide an order matching service and a market data feeding service to external clients.

4.1. Order Matching Engine

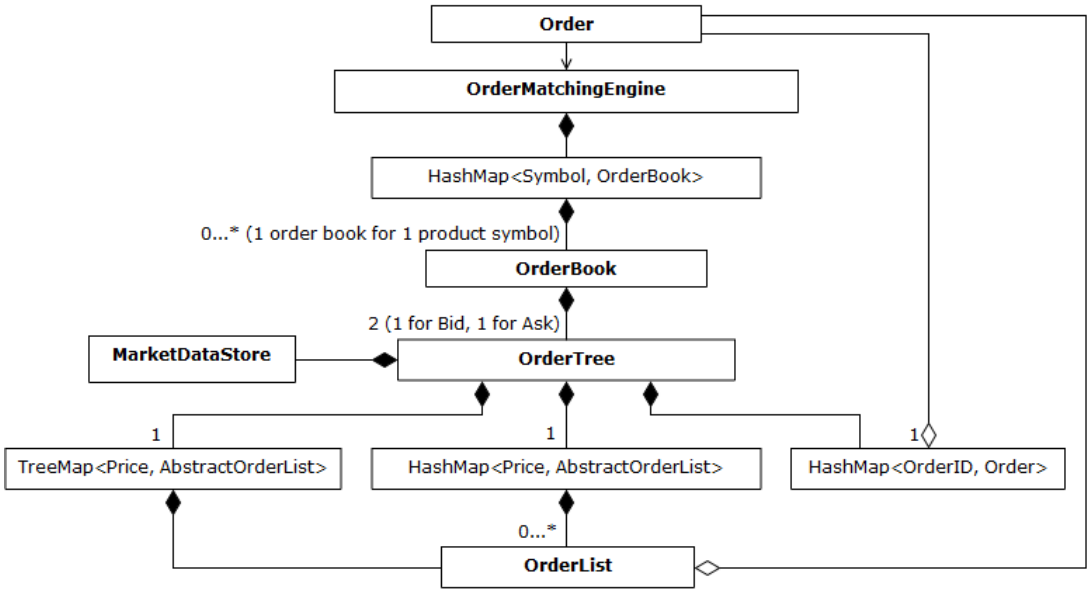

Figure 4.1 Architecture of Order Matching Engine

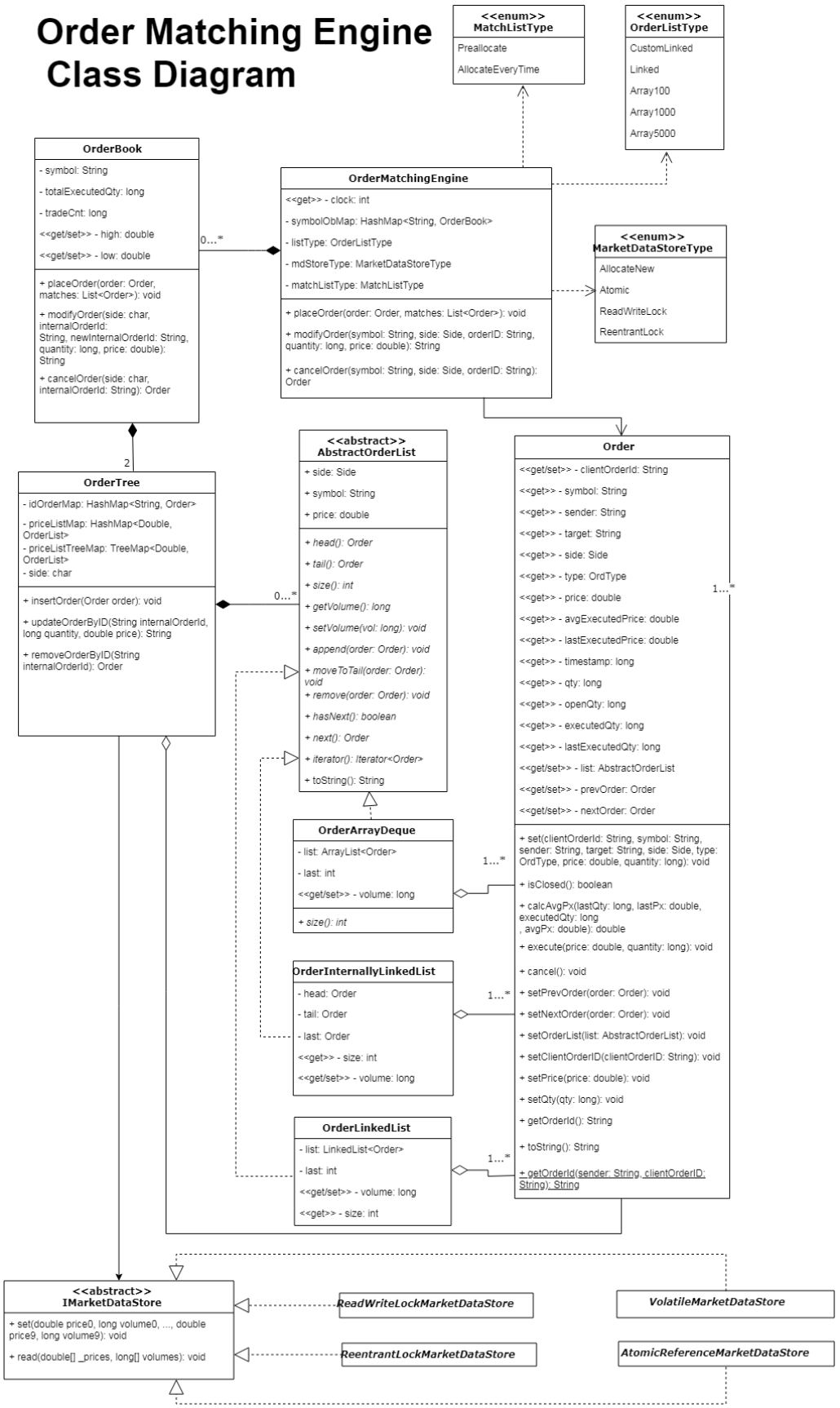

Figure 4.2 Class diagram of Order Matching Engine

The Order Matching Engine was developed with a reference to a GitHub project, "A fast java implementation of a limit order book" [38]. It provided the process of continuous double auction, in which potential buyers and sellers could raise their bid prices and offer prices by orders at any moments and the engine acted as an auctioneer to select prices to match the demands from buyers and sellers to clear orders.

The Order Matching Engine was designed to handle all order related operations in a single thread sequentially, so that number of context switches in CPU could be at the minimum value as to lower the chance to have cache miss and false sharing. All operations had to be executed fast in order to achieve low latency with only one thread. Data structures and algorithms with low time complexity were the key of success in the engine.

4.1.1. Main Class

The main class of the Order Matching Engine was *OrderMatchingEngine* class. It contained a hash map which mapped symbols to their own order books and provided methods for placing orders, modifying orders and cancelling orders. It would create a new order book when a symbol was used in a query and its order book was not found.

A hash map was chosen to store order books as it utilizes hashing technique which provides indexing and fast searches [39]. Hash map makes use of the method *hashCode()* of the keys to hash them to integers. The hashed integers are then be used in calculating the location of a bucket in the hash table of the hash map to place values of the keys [40]. When a value of a key need to be gotten from a hash map, the key will be hashed again to find the location of the bucket which stores the value. In this project, immutable object types, such as *String* and *Double*, were chosen to be the key, as hash

codes of immutable objects are cacheable and thus the time spent for hashing is very short. In an average case of a hash map, the *get*, *put* and *containsKey* operations in hash map are in constant time complexity, i.e. O(1).

4.1.2. Order Book

There was one order book for one symbol. Each order book contained one order tree for bid quotations, one order tree for offer quotations and several statistical variables such as high/low price and trade count.

4.1.3. Order Tree

Each order tree object consisted of an ID-to-Order hash map, a Price-to-Order-List hash map and a Price-to-Order-List tree map. The ID-to-Order hash map was used to map each order with its order identifier string as the key. It was mainly for the purpose of fast lookup of an order by its order identifier when a client requested to modify or cancel an order. All orders with the same limit price were placed in the same order list. Orders were sorted in the ordering of order arrival time. Earlier orders were in the earlier positions. The Price-to-Order-List hash map and the Price-to-Order-List tree map both were used to map order lists with their corresponding limit prices as the key. Limit prices in order objects were primitive, but when they were used as key for hash map and tree map, they would be converted to *Double* objects by *autoboxing*. *Double* objects are immutable and thus their hash codes can be cached like *String*. The Price-to-Order-List hash map was used for fast lookup of the corresponding order list by the limit price of a new order, so that the new order could append to the order list of the limit price quickly. The Price-to-Order-List tree map was an object of the class *java.util.TreeMap* which is backed by a red-black tree [41]. A red-black tree is a type of binary search tree which can rebalance itself, so that it can provide guaranteed O(log n) search times [42].

In *java.util.TreeMap*, keys are stored in a sorted order. The time complexity of *add*, *remove* and *containsKey* operations in *java.util.TreeMap* is always O(log n) where n is the number of nodes in the *java.util.TreeMap* [43]. The *java.util.TreeMap* stores the minimum key and the maximum key in the fields as the caches, so the Price-to-Order-List tree map could provide fast lookup time, O(1), for orders with the maximum prices in buy side and the minimum prices in sell side.

An order tree provided operations for inserting orders, updating order information and removing orders. These operations were to manage orders in those maps. A new order was inserted to the end of the order list of the price specified in the order. An order list was stored in the Price-to-Order-List hash map and the Price-to-Order-List tree map. It only contained orders sharing the same limit price. When a client chose to increase the quantity of an order and kept the price unchanged, the order would be moved to the end of the order list, so that it would be executed later than other existing orders. When a client chose to modify the limit price of an order, the order would be moved from the order list of the original price to the end of the order list of new limit price.

4.1.4. Order List

An order list was to store orders of the same limit price in the sequence of insertion order. Three types of order list were implemented: one using the *java.util.ArrayDeque*, one using the *java.util.LinkedList* and one using an Order-Internally-Linked List. The *java.util.ArrayDeque* was a standard collection which was backed by a resizable array and implemented the *java.util.Deque* interface. The *java.util.LinkedList* was a standard collection which implemented the *java.util.List* interface and the *java.util.Deque* interface. The Order-Internally-Linked List was a customized collection. Each order in the Order-Internally-Linked List stored object references of previous order and next

order if they exist to form a list. The total volumes of order lists were calculated based on the previous computed total volume values and the new quantity values of the orders during insertions, removals, modifications and executions of orders to speed up the computations and queries.

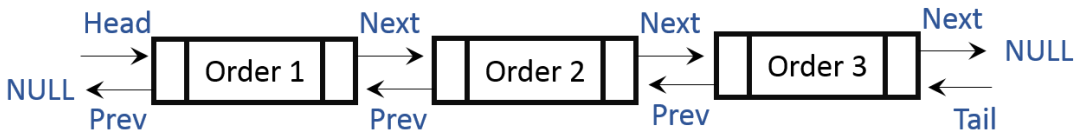

Figure 4.1.4A Structure of the Order-Internally-Linked List

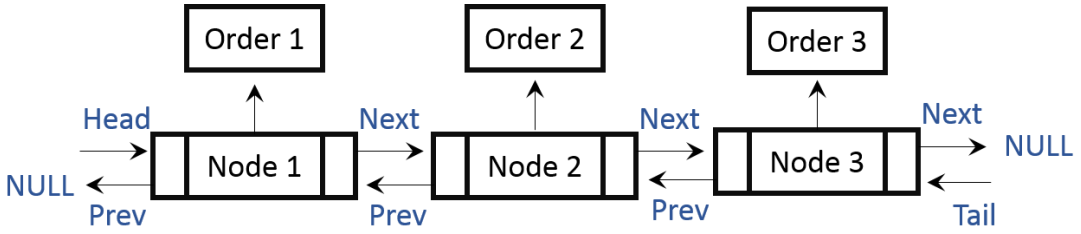

Figure 4.1.4B Structure of the Standard LinkedList

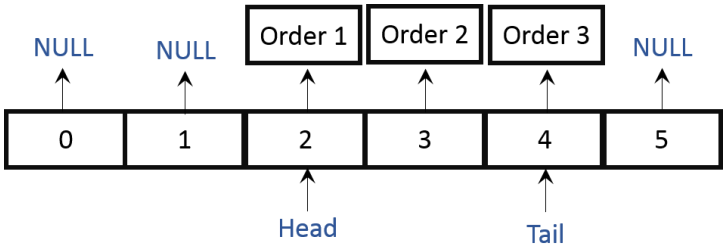

Figure 4.1.4C Structure of the Standard ArrayDeque

4.1.5. Order Matching Algorithm

Before an order entering to an order tree to wait for matching, the order book would try to match the order against the existing orders which were stored in sorted order lists in the order tree of the opposite side. This approach ensured orders were executed in the sequence of receiving time. A market order could execute with an order of the opposite

side without any price constraints. The limit price of a limit buy order was the maximum price that the client was willing to pay at. A new limit buy order would search the ask order tree to find and execute one or more sell orders which have limit prices lower than its limit price. The limit price of a limit sell order was the lowest price that the client was willing to sell at. A new limit sell order would search the bid order tree to find and execute one or more buy orders which have limit prices higher than its limit price. During executions, the prices of the orders from order trees were used as the executed prices.

4.1.6. Thread-safe Techniques for Market Data Cache

The main transaction flow of the Order Matching Engine was designed to be accessed by a single thread to ensure the number of context switches and data contentions to be minimum. Ideally there was only one thread responsible for both reading and writing operations on data, so that lock-free implementations could be easily employed in the main transaction flow.

However, the Order Matching Engine contained statistical market data information which needed to be disseminated to traders [24] [44] [45]. This required the Order Matching Engine to pass its data to another component in another thread which was responsible for the market data dissemination. A market data store was added to each order tree to cache top 10 levels of bid prices and ask prices. Several thread-safe techniques, such as *java.util.concurrent.locks.ReentrantReadWriteLock*, *java.util.concurrent.locks.ReentrantLock*, the *volatile* keyword and *java.util.concurrent.atomic.AtomicReference*, were implemented and could be selected to avoid the reader thread reading dirty data during a write operation of the writer thread. Both the *volatile* keyword and the *AtomicReference* were to two objects holding market

data. Both the *ReentrantLock* and the *ReentrantReadWriteLock* were to lock the segment of code to avoid writing when reading and to avoid reading when writing.

4.2. FIX Gateway

The FIX Gateway was the implementation of the Messaging Engine of the trading system. It was the contact point between traders and the trading system. It complied with FIX protocol 4.4 standard. FIX protocol stands for *Financial Information eXchange Protocol*, which is a global standard for financial institutions to exchange trade information [46].

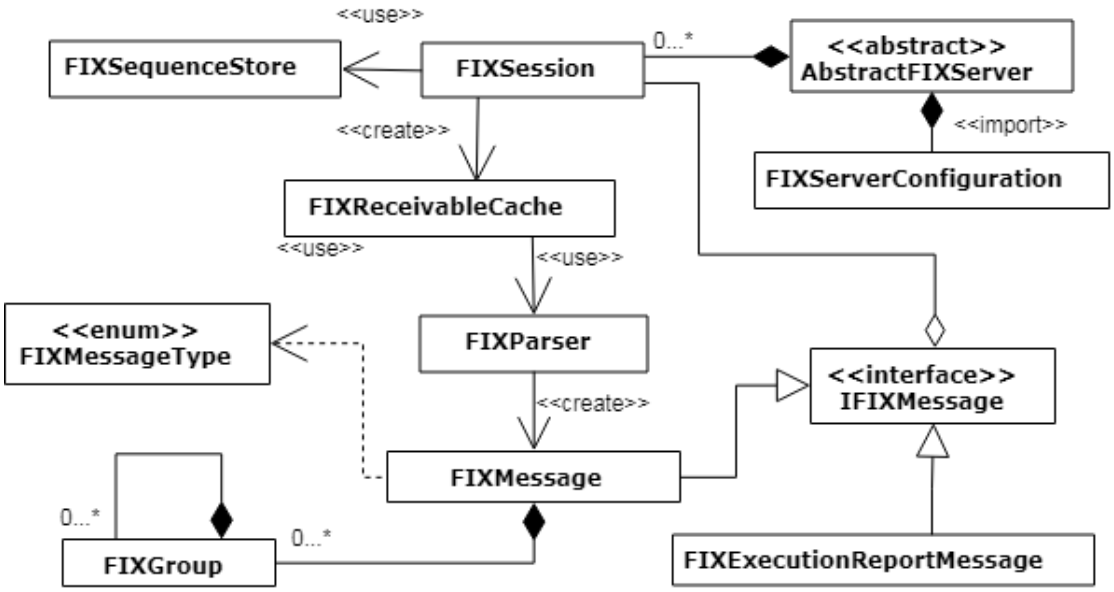

Figure 4.2(1) Architecture of FIX Gateway

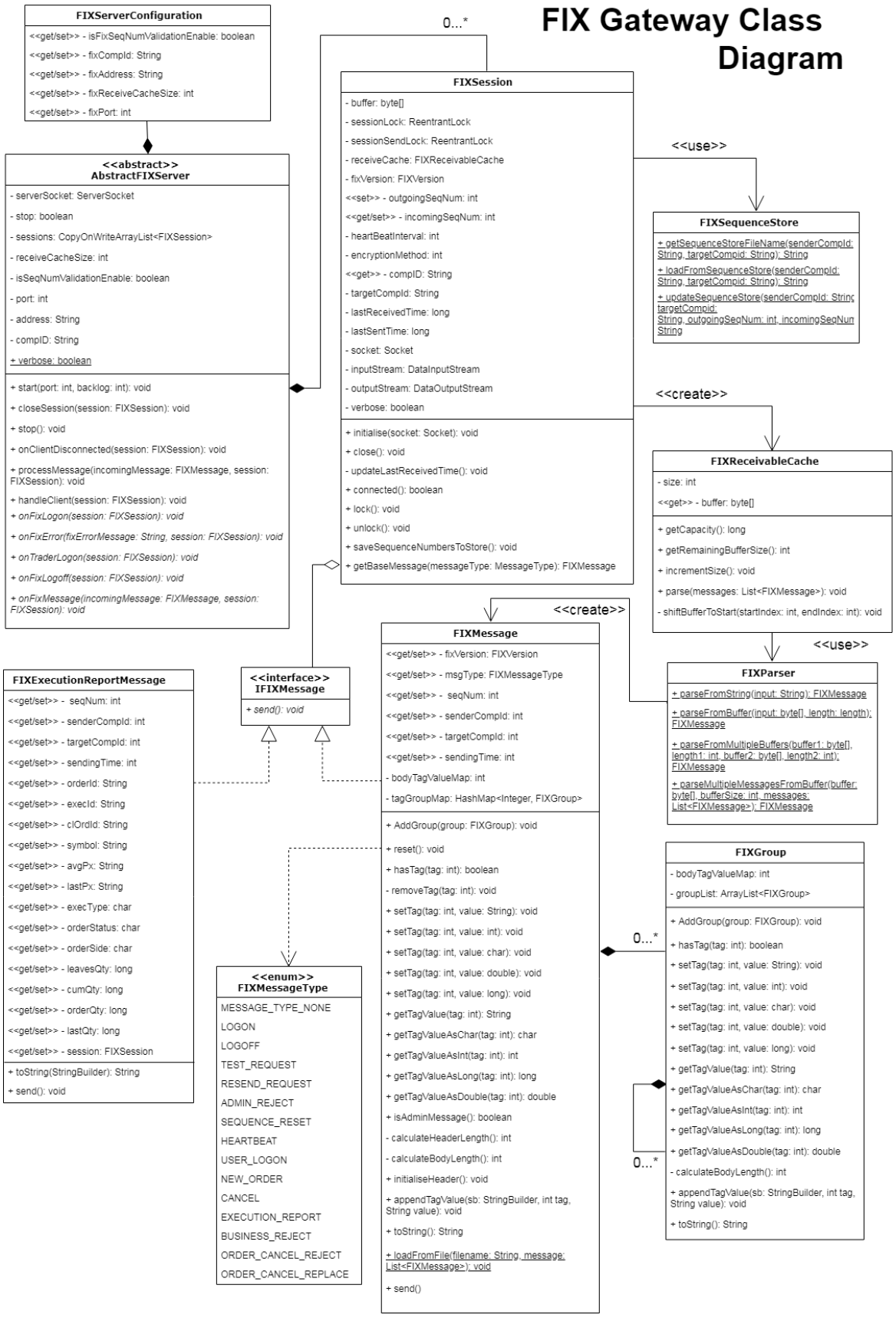

Figure 4.2(2) Class diagram of FIX Gateway

4.2.1. Connections and FIX Sessions

The main class of FIX Gateway was an abstract class, *AbstractFIXServer*. It was responsible to establish a server socket to listen for clients' connections. Each client's socket was wrapped in a FIX session object. All FIX sessions were stored in a *CopyOnWriteArrayList* in the *AbstractFIXServer*. *CopyOnWriteArrayList* is a thread-safe collection which copies the underlying array when a mutative operation of it is called [47]. The list storing FIX sessions had to be thread-safe as there were two threads in FIX Gateway interacting on it. One thread was responsible to manage FIX sessions and another thread was responsible to read binary data continuously from sockets of FIX sessions. Each alive FIX session kept getting binary stream from the socket and put received binary data to store in its own copy of the *FIXReceivableCache* object.

4.2.2. FIX Messages and Specifications

A FIX parser was developed and was used to unmarshall received binary data temporarily stored in the *FIXReceivableCache* object to FIX message objects. The FIX session objects could create a FIX message object which then converted to a string in binary to be sent to a client over networks. Each FIX message consisted of a hash map to store tag-value pairs. There were at least six message types supported in this FIX Gateway. They were *Logon*, *Heartbeat*, *New Order Single Request*, *Execution Report*, *Order Cancel Request* and *Order Cancel Replace Request*. A typical FIX message string was composed of header, body and trailer. Header contains tags of FIX version, body length, message type, target company identifier, sender company identifier and sequence number. The tags of FIX version, body length and message type were in the first, second and third position in a message string. The sequence number was to verify that if any message is duplicated or missed. Different messages could have different tags in their message body sections according to the message types. A checksum tag

was required in the trailer section of each message to allow to check if a message was corrupted.

As the Execution Report Messages were the most frequently generated reply messages during the run time of the trading system and the operations involving maps were slow [48], we created a tailor-made Java class, *FIXExecutionReportMessage*, for it. In this class, no mapping was used, and specific member variables were declared for those tags required by the Execution Report type.

The following sections show the formats of FIX messages.

4.2.2.1. Header

The header must exist in every message.

Tag ID	Tag Name	Required	Type	Description
8	BeginString	Yes	String	FIX.4.4 (Fixed in the 1st field of the message)
9	BodyLength	Yes	int	Use to get the length of the incoming message (Fixed in the 2nd field of the message)
35	MsgType	Yes	String	(Fixed in the 3rd field of the message)
49	SenderCompID	Yes	String	
56	TargetCompID	Yes	String	
34	MsgSeqNum	Yes	int	
52	SendingTime	Yes	String	

Table 4.2.2.1 FIX Message Header Format

4.2.2.2. Trailer

The header must exist in every message.

Tag ID	Tag Name	Required	Type	Description
10	CheckSum	Yes	int	Calculated by summing up all tags (excluding tag 10 itself) (Fixed in the last field of the message)

Table 4.2.2.2 FIX Message Trailer Format

4.2.2.3. Logon

Logon was used to establish a FIX session by a client and is used by a server to acknowledge a client that a logon is accepted [49].

Tag ID	Tag Name	Required	Type	Description
	<Message Header>	Yes		35=A
98	EncryptMethod	Yes	int	
108	HeartBtInt	Yes	int	Heartbeat Interval
	<Message Trailer>	Yes		

Table 4.2.2.3 FIX Message Type: Logon Format

4.2.2.4. Heartbeat

Heartbeat was used to monitor connection status [49].

Tag ID	Tag Name	Required	Type	Description
	<Message	Yes		35=0

	Header>			
112	TestReqID		int	Required if this heartbeat is replied for a TestRequest
	<Message Trailer>	Yes		

Table 4.2.2.4 FIX Message Type: Heartbeat Format

4.2.2.5. NewOrderSingle (New Order Request)

NewOrderSingle was used by clients to submit an order to the server for execution [49].

Tag ID	Tag Name	Required	Type	Description
	<Message Header>	Yes		35=D
55	Symbol	Yes	String	Representation of a security
11	ClOrdID	Yes	String	Client Order ID
38	OrderQty	Yes	long	Order Quantity
54	Side	Yes	char	1=Buy, 2=Sell
40	OrdType	Yes	char	1=Market, 2=Limit
44	Price		double	Required if 40=2
60	TransactTime	Yes	String	Timestamp of the transaction
	<Message Trailer>	Yes		

Table 4.2.2.5 FIX Message Type: NewOrderSingle Format

4.2.2.6. OrderCancelRequest

OrderCancelRequest was used by clients to cancel all remaining quantity of an order [49].

Tag ID	Tag Name	Required	Type	Description
	<Message Header>	Yes		35=F
41	OrigClOrdID	Yes	String	Original Client Order ID
11	ClOrdID	Yes	String	Client Order ID
55	Symbol	Yes	String	Representation of a security
54	Side	Yes	char	1=Buy, 2=Sell
	<Message Trailer>	Yes		

Table 4.2.2.6 FIX Message Type: OrderCancelRequest Format

4.2.2.7. OrderCancelReplaceRequest

OrderCancelReplaceRequest was used by clients to modify quantity and price of an order [49].

Tag ID	Tag Name	Required	Type	Description
	<Message Header>	Yes		35=G
41	OrigClOrdID	Yes	String	Original Client Order ID
11	ClOrdID	Yes	String	Client Order ID
55	Symbol	Yes	String	Representation of a security
54	Side	Yes	char	1=Buy, 2=Sell

38	OrderQty	Yes	long	Order Quantity
44	Price	Yes	double	
60	TransactTime	Yes	String	Timestamp of the transaction
	<Message Trailer>	Yes		

Table 4.2.2.7 FIX Message Type: OrderCancelReplaceRequest Format

4.2.2.8. ExecutionReport

It was used to report order status for the following requests: *NewOrderSingle* (D), *OrderCancelRequest* (F) and *OrderReplaceCancelRequest* (G).

Tag ID	Tag Name	Required	Type	Description
	<Message Header>	Yes		
37	OrderID	Yes	String	Identifier of an order
17	ExecID	Yes	String	Execution ID
150	ExecType	Yes	char	Execution Type 0=New, 4=Cancelled, 5=Replaced, 6=PendingCancel, 8=Rejected, E=PendingReplace, F=Trade
39	OrdStatus	Yes	char	Order Status 0=New, 1=Partially Filled,

				2=Filled, 4=Cancelled, 5=Replaced, 6=PendingCancel, 8=Rejected, E=PendingReplace
54	Side	Yes	char	1=Buy, 2=Sell
151	LeavesQty	Yes	long	Remaining Quantity
14	CumQty	Yes	long	Total filled Quantity
6	AvgPx	Yes	double	Average Price
11	ClOrdID	Yes	String	Client Order ID
38	OrderQty	Yes	long	Order Quantity
55	Symbol	Yes	String	Representation of a security
31	LastPx		double	Last Price, Required if 39=1 or 39=2
32	LastQty		long	Last Quantity, Required if 39=1 or 39=2
	<Message Trailer>	Yes		

Table 4.2.2.8 FIX Message Type: ExecutionReport Format

4.3. Overall Architecture

The flow of placing an order was as the following:

1. Client sends a *NewOrderSingle* request to the FIX Gateway.

2. The FIX Gateway unmarshalls the request into a Message object.

3. A utility converts the Message object to an Order object which is used by the Order Matching Engine.

4. The Order object is input into the place order function of Order Matching Engine. The Order Matching Engine does the process of order matching internally and then returns a list of matched orders.

5. A utility generates *Execution Report* reply messages from the matched order list.

6. The FIX Gateway sends those execution reports to the client through FIX Gateway.

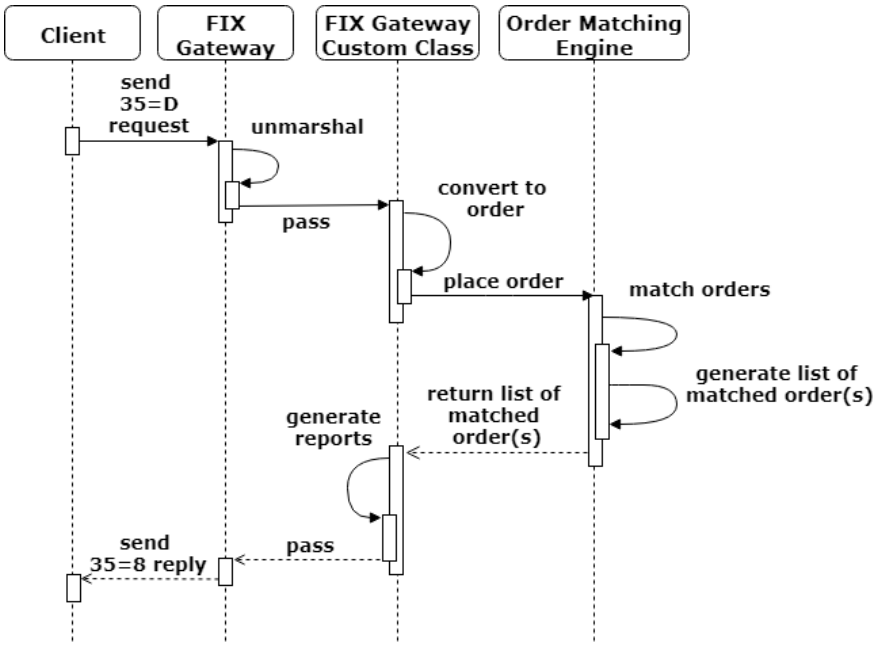

Figure 4.3 Sequence diagram showing the flow of placing an order by client

4.4. Choice of JVM and Garbage Collector

As Java applications run on Java Virtual Machines, the choice of JVM could directly affect the throughput and latency of the applications. In this dissertation, we compared

the performance of the trading system on Oracle HotSpot JVM and the Eclipse OpenJ9 JVM respectively.

Oracle HotSpot JVM and the Eclipse OpenJ9 JVM have different sets of garbage collectors. For Oracle HotSpot Java Virtual Machine, we chose either *Concurrent Mark Sweep Garbage Collector*, *Garbage-First Garbage Collector* or *Z Garbage Collector*. For the Eclipse OpenJ9 Java Virtual Machine, we selected three garbage collection policy, *GenCon*, *Metronome* and *Balanced*, to do benchmarks.

5. Testing

5.1. Unit Testing

Test cases of unit testing for the Order Matching Engine and the FIX Gateway were developed using JUnit 4 library to ensure the correctness of components. For the Order Matching Engine, functions for placing an order, modifying an order and cancelling an order were covered in test case. For the FIX Gateway, functions for unmarshalling binary data to a message, generating a market data message, handling a logon request, handling missing tags in various messages, handling a test message and a heart message were covered. The instruction coverage in classes are shown in the following tables.

Element	Instruction Coverage
AbstractOrderList	100%
BasicMarketDataStore	100%
AtomicReferenceMarketDataStore	100%
VolatileMarketDataStore	100%
OrderArrayDeque	100%
OrderLinkedList	100%
ReentrantReadWriteLockMarketData	94.2%
ReentrantLockMarketDataStore	93.9%
OrderInternallyLinkedList	91.3%
OrderTree	89.3%
OrderBook	84.2%
OrderMatchingEngine	81.1%
ImplementSelector	68.2%

Order	64.5%

Table 5.1.1 Instruction coverage of Order Matching Engine in unit test cases

Element	Instruction Coverage
FIXParser	98.9%
FIXReceivableCache	95.8%
FIXUtility	94.6%
FIXConst	93.5%
FIXMessage	93.4%
FIXExecutionReportMessage	93.1%
FIXGroup	90.9%
FIXSession	75.1%
FIXServerConfiguration	72.9%
AbstractFIXServer	42.4%
FIXSequenceStore	36.9%

Table 5.1.2 Instruction coverage of FIX Gateway in unit test cases

6. Experimental Results

6.1. Experimental settings

Both the Order Matching Engine and the FIX Gateway were benchmarked with JMH [51]. The server used for benchmarks was provided by Global eSolutions (HK) Limited. It was a virtual machine running on VMware with Centos 6.9 64 bit as the operating system. The total memory was 7,872 megabytes. The number of CPU core available was 2. We used the data file of testing orders provided by the supporting company, Global eSolutions (HK) Limited, to do benchmarks.

Benchmark tests executed in the throughput mode for the Order Matching Engine and the sampling mode for the FIX Gateway. Each benchmark test ran 10 measured iterations and each measured iteration lasted for 10 seconds. Each benchmark test ran 10 warm-up iterations and 10 measured iterations. Each warm-up iteration lasted for 10 seconds. Each measured iteration lasted for 60 seconds. The warm-up iterations were used to provide enough time to ensure Just-In-Time optimizations finished before the measured iterations, so that measured iterations could measure stable executions of the program.

Oracle HotSpot JVM and Eclipse OpenJ9 JVM were used in the benchmarks of the Order Matching Engine and the benchmarks of FIX Gateway for comparisons. For benchmark tests using Oracle HotSpot, the Concurrent Mark Sweep garbage collector, Garbage-First garbage collector and Z Garbage Collector were used. For benchmark tests using Eclipse OpenJ9 JVM, Balanced policy, Generational Concurrent policy (GenCon) and Metronome policy were used. For the memory setting of Java Virtual machine, the min heap size parameter (-Xms) and the max heap size parameter (-Xmx)

were both set to be the same value, 3936 megabytes, to eliminate the need for dynamic memory allocation at run-time. We turned on the server compiler of Oracle HotSpot JVM in the benchmarks with the parameter (-server), however we could not find any parameter for server compiler in the Eclipse OpenJ9 JVM to use.

JMH version	1.21
VM version	JDK 11.0.2, Java HotSpot (TM) 64-Bit Server VM, 11.0.2+9-LTS JDK 11.0.2, Eclipse OpenJ9 VM, openj9-0.12.1
VM options	For Oracle HotSpot Java Virtual Machine: 1. -server -Xms3936m -Xmx3936m -XX:+UnlockExperimentalVMOptions -XX:+UseZGC 2. -server -Xms3936m -Xmx3936m -XX:+UseConcMarkSweep 3. -server -Xms3936m -Xmx3936m -XX:+UseG1GC For Eclipse OpenJ9 Java Virtual Machine: 1. -Xms3936m -Xmx3936m -Xgcpolicy:metronome 2. -Xms3936m -Xmx3936m -Xgcpolicy:gencon 3. -Xms3936m -Xmx3936m -Xgcpolicy:balanced
Warmup	10 iterations, 10 s each
Measurement	10 iterations, 60 s each
Threads	1 thread, will synchronize iterations

Table 6.1 Specifications of benchmarks

6.2. Benchmark of Order Matching Engine

The order list implementations used in the Order Matching Engine and the different thread-safe techniques used in the market data store of the engine were benchmarked with the order placement function of the engine. The order placing function was the main function of the engine.

6.2.1. Benchmark of data structures of order lists

The available implementations of the order lists were the Order-Internally-Linked List, the Standard LinkedList and the Standard ArrayDeque. Order-Internally-Linked List was a highly-customized doubly linked list formed by linking orders with embedded order object references for previous and next orders in each order object. The order list structure using the Standard LinkedList was backed by a java.util.LinkedList object of the standard Java Library. The order list structure using the Standard ArrayDeque was backed by the *java.util.ArrayDeque* object of the standard Java Library. They were benchmarked to find out their effects on the performance of the order placement function of the Order Matching Engine. The thread-safe technique used in market data in these benchmarks were the AtomicReference approach. The results were as the following.

1. **Average Throughput and Average Latency**

Average Throughput (order/s) Average Latency (us/order)	**Order-Internally-Linked List**	**Standard LinkedList**	**Standard ArrayDeque**	***Max % improvement of throughput (= (Largest -***

				Smallest)/ Smallest)
Z Garbage Collector (HotSpot JVM)	454,228 order/s 2.201 us/order	397,255 order/s 2.517 us/order	263,233 order/s 3.799 us/order	*(454,228-263,233)/ 263,233 = 72.6%*
Concurrent Mark Sweep Collector (HotSpot JVM)	579,932 order/s 1.724 us/order	578,593 order/s 1.728 us/order	495,486 order/s 2.018 us/order	*(579,932-495,486)/ 495,486 = 17.0%*
Garbage-First Garbage Collector (HotSpot JVM)	443,125 order/s 2.256 us/order	447,039 order/s 2.237 us/order	427,007 order/s 2.342 us/order	*(443,125-427,007)/ 427,007 = 3.8%*
Max % improvement of throughput within HotSpot JVM by changing garbage collector (= (Largest - Smallest)/ Smallest)	*(579,932-443,125)/ 443,125 = 30.9%*	*(578,593-397,255)/ 397,255 = 45.6%*	*(495,486-263,233)/ 263,233 = 88.2%*	

GenCon policy (OpenJ9 JVM)	397,321 order/s 2.517 us/order	385,824 order/s 2.592 us/order	368,991 order/s 2.710 us/order	*(397,321-368,991)/ 368,991 = 7.7%*
Balanced policy (OpenJ9 JVM)	312,918 order/s 3.196 us/order	314,746 order/s 3.177 us/order	296,945 order/s 3.368 us/order	*(314,746-296,945)/ 296,945 = 6.0%*
Metronome policy (OpenJ9 JVM)	79,439 order/s 12.588 us/order	74,005 order/s 13.513 us/order	71,222 order/s 14.040 us/order	*(79,439-74,005)/ 74,005 = 7.3%*
Max % improvement of throughput within OpenJ9 JVM by changing GC policy (= (Largest - Smallest)/ Smallest)	*(397,321-79,439)/ 79,439 = 400.2%*	*(385,824-74,005)/ 74,005 = 421.4%*	*(368,991-71,222)/ 71,222 = 418.1%*	

Table 6.2.1(1) Throughput and latency of order placement function of Order Matching Engine with different data structures (Remark: Higher throughput mean better results and lower latency mean better results)

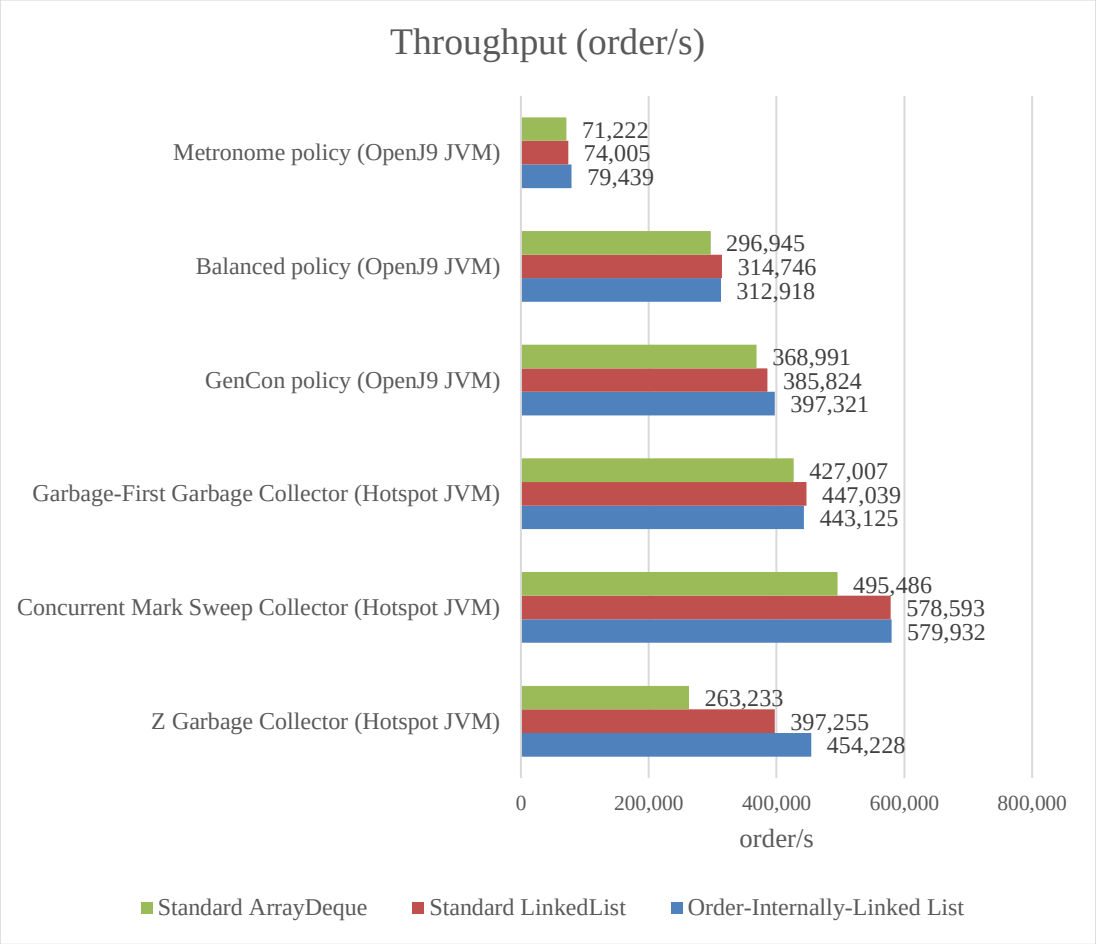

Chart 6.2.1(1) Throughput of order placement function of Order Matching Engine with different data structures (Remark: Higher throughput mean better results)

In the five benchmarks of out of the total six benchmarks, the Order-Internally-Linked List had the highest average throughput among the order list data structures. In the total six benchmarks, the order list using the standard ArrayDeque had the lowest throughput among the order list data structures.

For the Order-Internally-Linked List, the setting of Oracle HotSpot Virtual Machine with Concurrent Mark Sweep Garbage Collector had the highest throughput, which was about 580,000 orders per second, and the setting of Oracle HotSpot Virtual Machine with Z Garbage Collector had the second highest throughput, which was about 454,000 orders per second.

In the benchmarks running on Oracle HotSpot Java Virtual Machine, the maximum improvement in throughputs by changing garbage collectors were about 30.9%~88.2%, and the maximum improvement in throughputs by changing order list data structure were 3.8%~72.6%.

In the benchmarks running on the Eclipse OpenJ9 Java Virtual Machine, the maximum improvement in throughputs by changing garbage collectors were 400.2%~421.4%, and the maximum improvement in throughputs by changing order list data structure were 6%~7.7%.

2. Object Allocation Rate

Object Allocation Rate (byte/order)	**Order-Internally-Linked List**	**Standard LinkedList**	**Standard ArrayDeque**
Z Garbage Collector (HotSpot JVM)	798 byte/order	8,828 byte/order	2,271 byte/order
Concurrent Mark Sweep Collector (HotSpot JVM)	604 byte/order	4,712 byte/order	1,551 byte/order
Garbage-First Garbage Collector (HotSpot JVM)	616 byte/order	4,372 byte/order	1,193 byte/order
GenCon policy (OpenJ9 JVM)	794 byte/order	6,044 byte/order	1,638 byte/order
Balanced policy	752 byte/order	5,545 byte/order	1,790 byte/order

(OpenJ9 JVM)			
Metronome policy (OpenJ9 JVM)	784 byte/order	10,465 byte/order	5,913 byte/order

Table 6.2.1(2) Object Allocation Rate per order placement of Order Matching Engine with different order list implementations (Remark: Lower values mean better results)

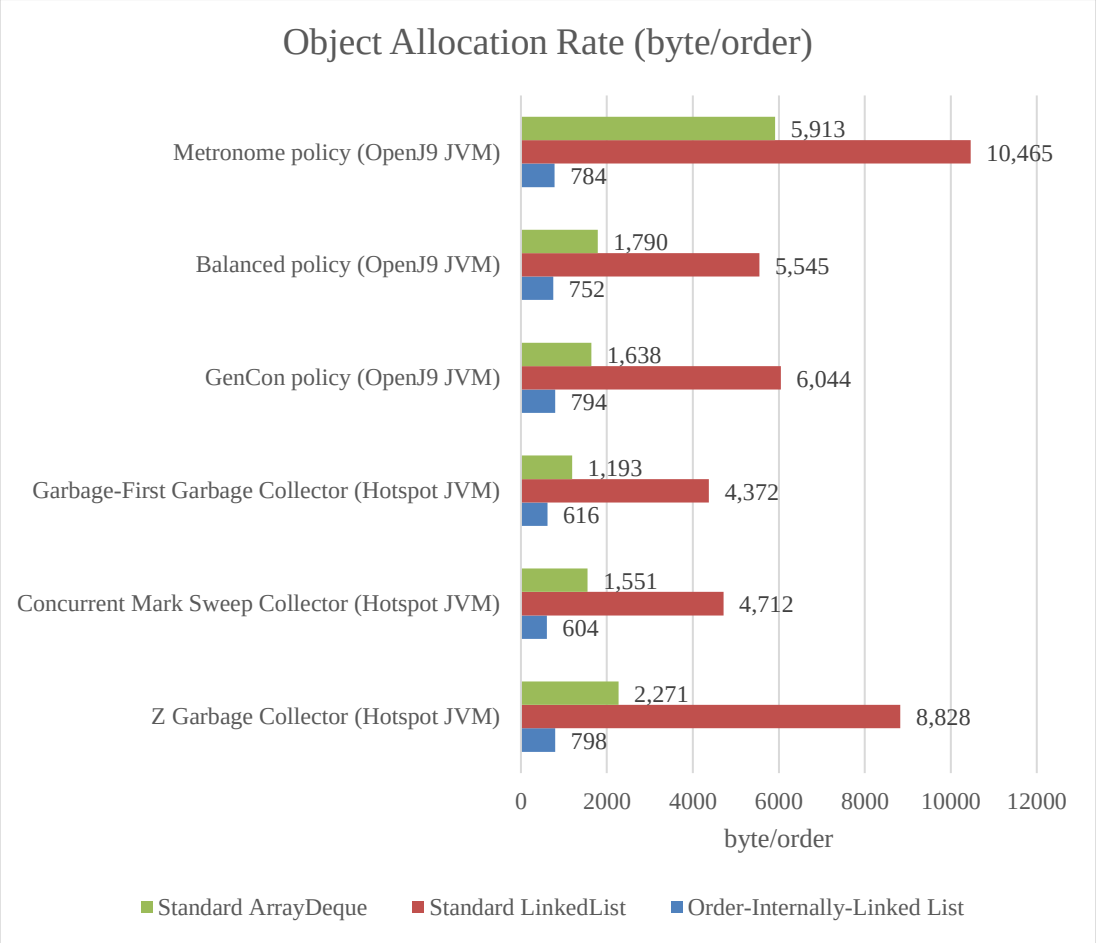

Chart 6.2.1(2) Object Allocation Rate per order placement of Order Matching Engine with different order list implementations (Remark: Lower values mean better results)

The order-Internally-Linked list implementation allocated the lowest memory size, which was 604~798 bytes per order placement operation among order list implementations. The implementation using the standard LinkedList allocated the most

memory size, which was 4,372~10,465 bytes, per order placement operation.

3. Garbage Collection Time and Count

Average Time Spent per Garbage Collection (ms/count) **Garbage Collection Count (count)** **GC time (ms)**	**Order-Internally-Linked List**	**Standard LinkedList**	**Standard ArrayDeque**
Z Garbage Collector (HotSpot JVM)	0.6 ms/count 90 counts 54 ms	0.6 ms/count 90 counts 48 ms	0.533 ms/count 90 counts 54 ms
Concurrent Mark Sweep Collector (HotSpot JVM)	24 ms/count 103 counts 2,521 ms	26 ms/count 108 counts 2,806 ms	23 ms/count 123 counts 2,777 ms
Garbage-First Garbage Collector (HotSpot JVM)	79 ms/count 22 counts 1,742 ms	117 ms/count 14 counts 1,638 ms	89 ms/count 25 counts 2,220 ms
GenCon policy (OpenJ9 JVM)	56 ms/count 14 counts 820 ms	64 ms/count 14 counts 889 ms	54 ms/count 16 counts 868 ms
Balanced policy (OpenJ9 JVM)	143 ms/count 9 counts 1,287 ms	151 ms/count 10 counts 1,508 ms	139 ms/count 11 counts 1,534 ms

Metronome policy	3 ms/count	3 ms/count	3 ms/count
(OpenJ9 JVM)	4,650 counts	4,609 counts	4,674 counts
	14,028 ms	13,889 ms	14,069 ms

Table 6.2.1(3) Garbage collection statistics of the benchmark of order placement of Order Matching Engine with different order list implementations (Remark: Lower values mean better results)

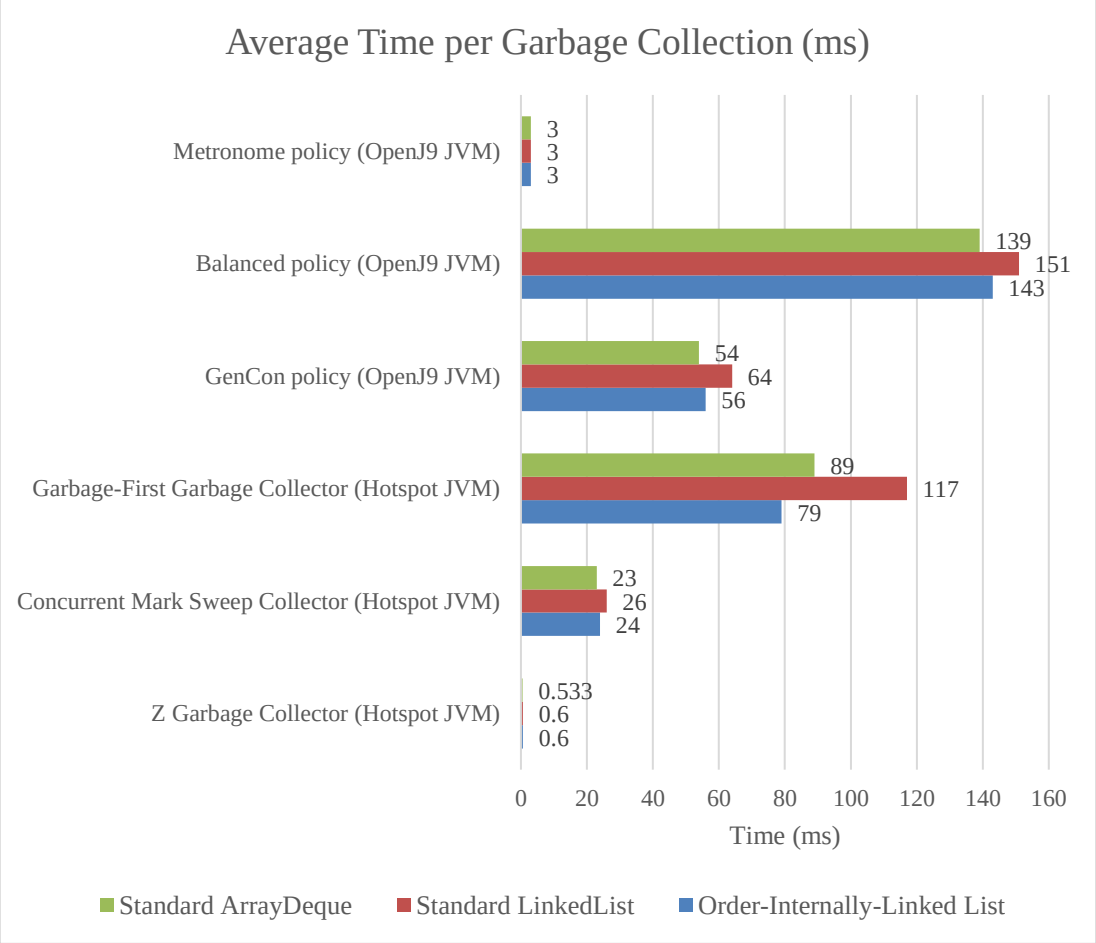

Chart 6.2.1(3A) Average time per garbage collection during the benchmark of order placement of Order Matching Engine with different order list implementations (Remark: Lower values mean better results)

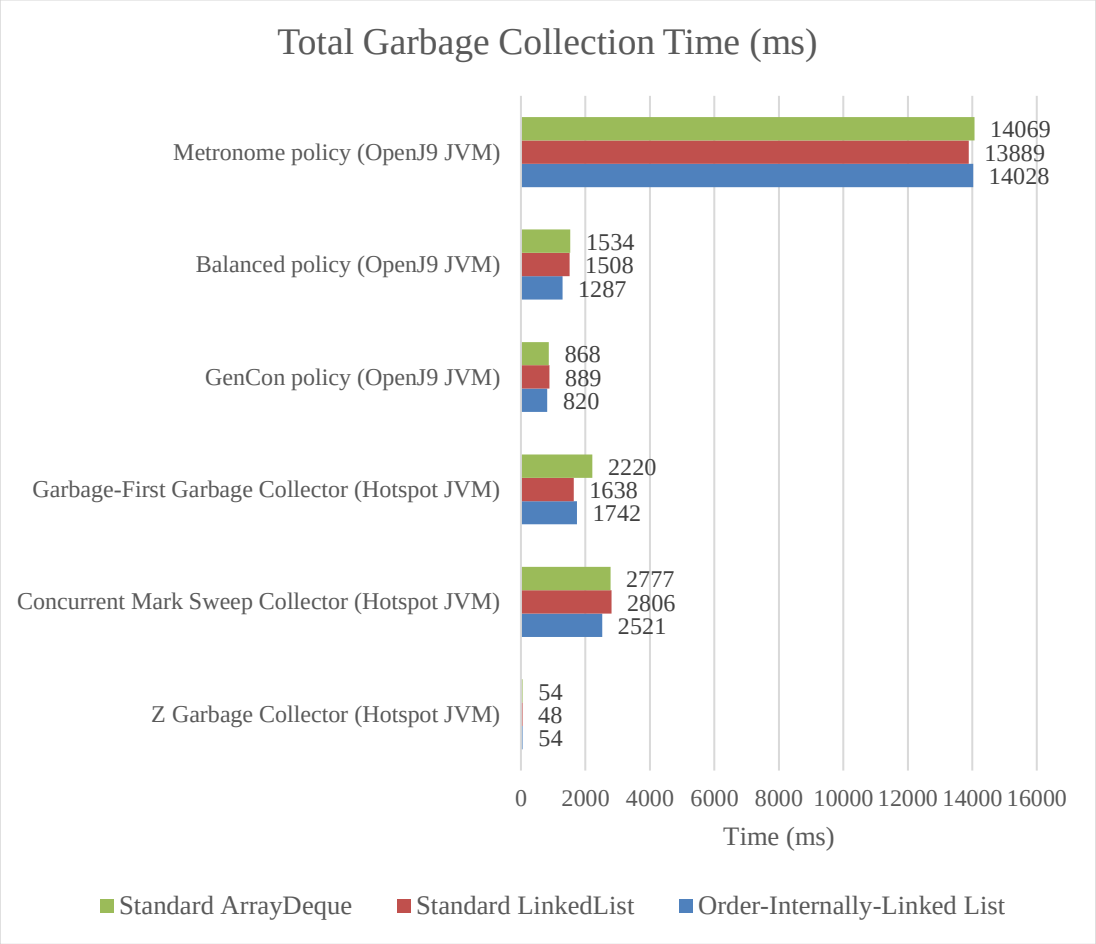

Chart 6.2.1(3B) Total garbage collection time during the benchmark of order placement of Order Matching Engine with different order list implementations (Remark: Lower values mean better results)

The Z Garbage Collector of Oracle HotSpot JVM spent the shortest time, which was 48~54 milliseconds, in garbage collection. The setting had the second lowest garbage collection time was the balance policy of the Eclipse OpenJ9 Java Virtual Machine, but its garbage collection time was already 820~889 milliseconds.

The Z Garbage Collector of Oracle HotSpot JVM also had the lowest average time spent per garbage collection, which was 0.533~0.6 milliseconds per count. The setting had the second lowest average time spent per garbage collection was the metronome policy of the Eclipse OpenJ9 Java Virtual Machine, and its garbage collection time was

3 milliseconds.

6.2.2. Benchmark of thread-safe techniques for Market Data

The four thread-safe techniques used for market data were benchmarked with the order placement function to find out their effects on the function. The techniques were *AtomicReference*, *ReentrantReadWriteLock*, *ReentrantLock* and the *volatile* keyword. *AtomicReference*, *ReentrantReadWriteLock* and *ReentrantLock* were classes from the package *java.util.concurrent*. The order list implementation used in these benchmarks were the Order-Internally-Linked List. The results were as the following.

1. Throughput

Average Throughput (order/s)	**Atomic-Reference**	**Volatile**	**Reentrant-Read-Write-Lock**	**Reentrant-Lock**
Z Garbage Collector (HotSpot JVM)	445,605 order/s	449,084 order/s	491,344 order/s	483,865 order/s
Concurrent Mark Sweep Collector (HotSpot JVM)	579,932 order/s	579,177 order/s	596,177 order/s	613,809 order/s
Garbage-First Garbage Collector (HotSpot JVM)	443,125 order/s	404,084 order/s	480,879 order/s	490,210 order/s
GenCon policy (OpenJ9 JVM)	397,321 order/s	391,936 order/s	387,615 order/s	389,291 order/s
Balanced policy (OpenJ9 JVM)	312,917 order/s	318,919 order/s	306,387 order/s	315,153 order/s
Metronome policy (OpenJ9 JVM)	79,438 order/s	80,865 order/s	78,728 order/s	78,807 order/s

Table 6.2.2(1) Average time spent in placing order operations with different synchronization techniques used in the Market Data Store of Order Matching Engine (Remark: Higher values mean better results)

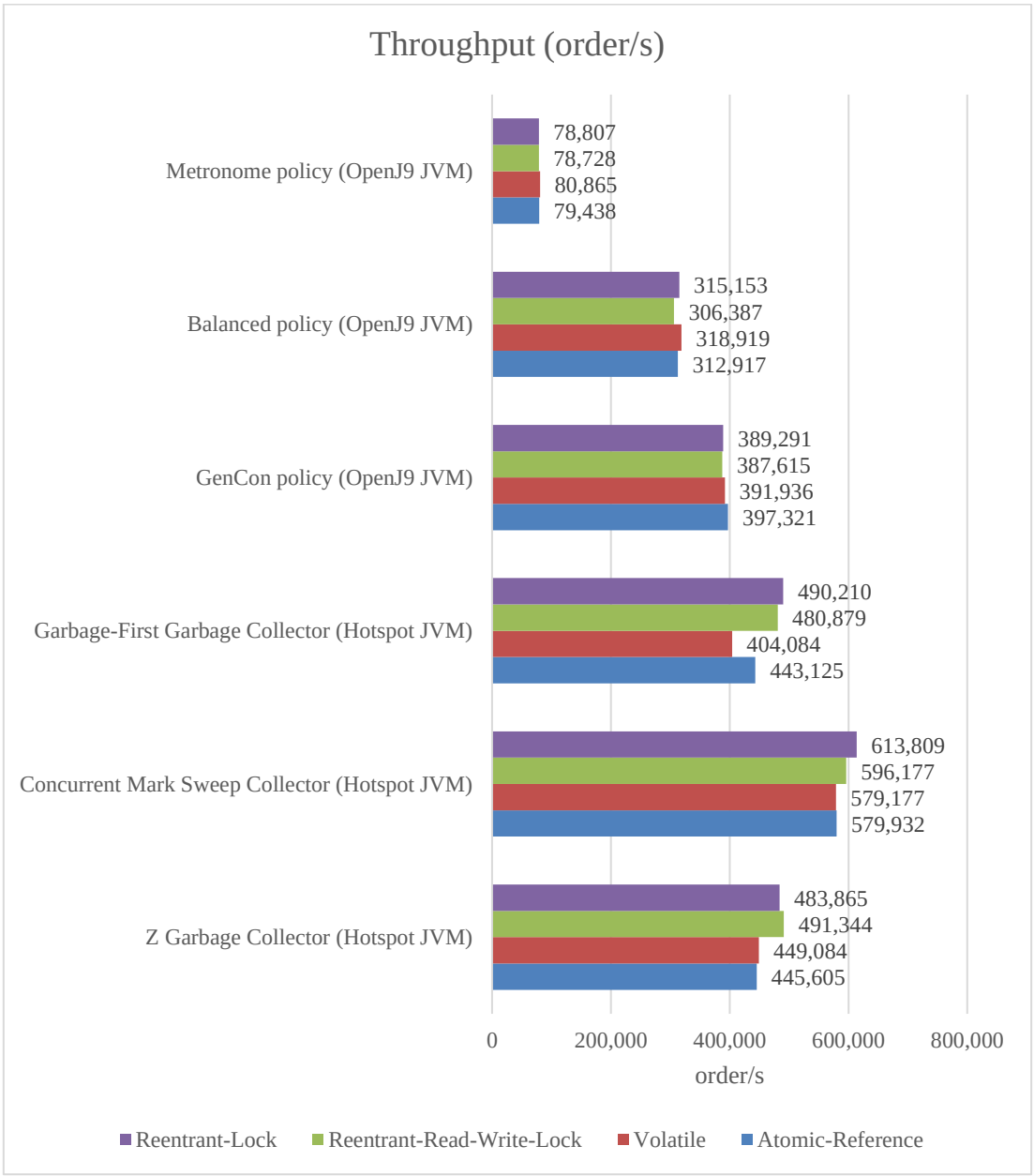

Chart 6.2.2(1) Average time spent in placing order operations with different synchronization techniques in the Market Data Store of Order Matching Engine (Remark: Higher values mean better results)

For the benchmarks running on Oracle HotSpot Java Virtual Machine, the throughputs of the Order Matching Engine using lock techniques had higher throughput than those using lock-free techniques. However, for the benchmarks running on the Eclipse OpenJ9 Java Virtual Machine, the throughputs of those using lock-free techniques were

higher.

In the five benchmarks out of the total six benchmarks, the engines using *Reentrantlock* had slightly higher throughput than those using *ReentrantReadWriteLock*.

In the benchmarks running on Oracle HotSpot Java Virtual Machine, the maximum percentage improvement in throughputs by changing garbage collectors were 24%~43.3%, and the maximum percentage improvement in throughputs by changing thread-safe techniques were 6%~21.3%. The Concurrent Mark Sweep Garbage Collector generally had higher throughput than the Garbage-First Garbage Collector and the Z Garbage Collector. The Z Garbage Collector had similar throughput level of the Garbage-First Garbage Collector.

In the benchmarks running on the Eclipse OpenJ9 Java Virtual Machine, the maximum percentage improvement in throughputs by changing garbage collectors were 384.7~400.2%, and the maximum percentage difference in throughputs by changing thread-safe techniques were 2.5%~4.1%. The GenCon policy generally had higher throughput than the Balanced policy and the Metronome policy. The Metronome policy had much lower throughput than other two policies.

2. Object Allocation Rate

Object Allocation Rate (byte/order)	**Atomic-Reference**	**Volatile**	**Reentrant-Read-Write-Lock**	**Reentrant-Lock**
Z Garbage Collector	798 byte/order	803 byte/order	803 byte/order	803 byte/order

(HotSpot JVM)				
Concurrent Mark Sweep Collector (HotSpot JVM)	611 byte/order	598 byte/order	611 byte/order	598 byte/order
Garbage-First Garbage Collector (HotSpot JVM)	632 byte/order	637 byte/order	636 byte/order	612 byte/order
GenCon policy (OpenJ9 JVM)	797 byte/order	812 byte/order	759 byte/order	759 byte/order
Balanced policy (OpenJ9 JVM)	792 byte/order	779 byte/order	799 byte/order	779 byte/order
Metronome policy (OpenJ9 JVM)	816 byte/order	820 byte/order	819 byte/order	814 byte/order

Table 6.2.2(2) Object Allocation Rate per order placement of Order Matching Engine with different synchronization techniques in the Market Data Store (Remark: Lower values mean better results)

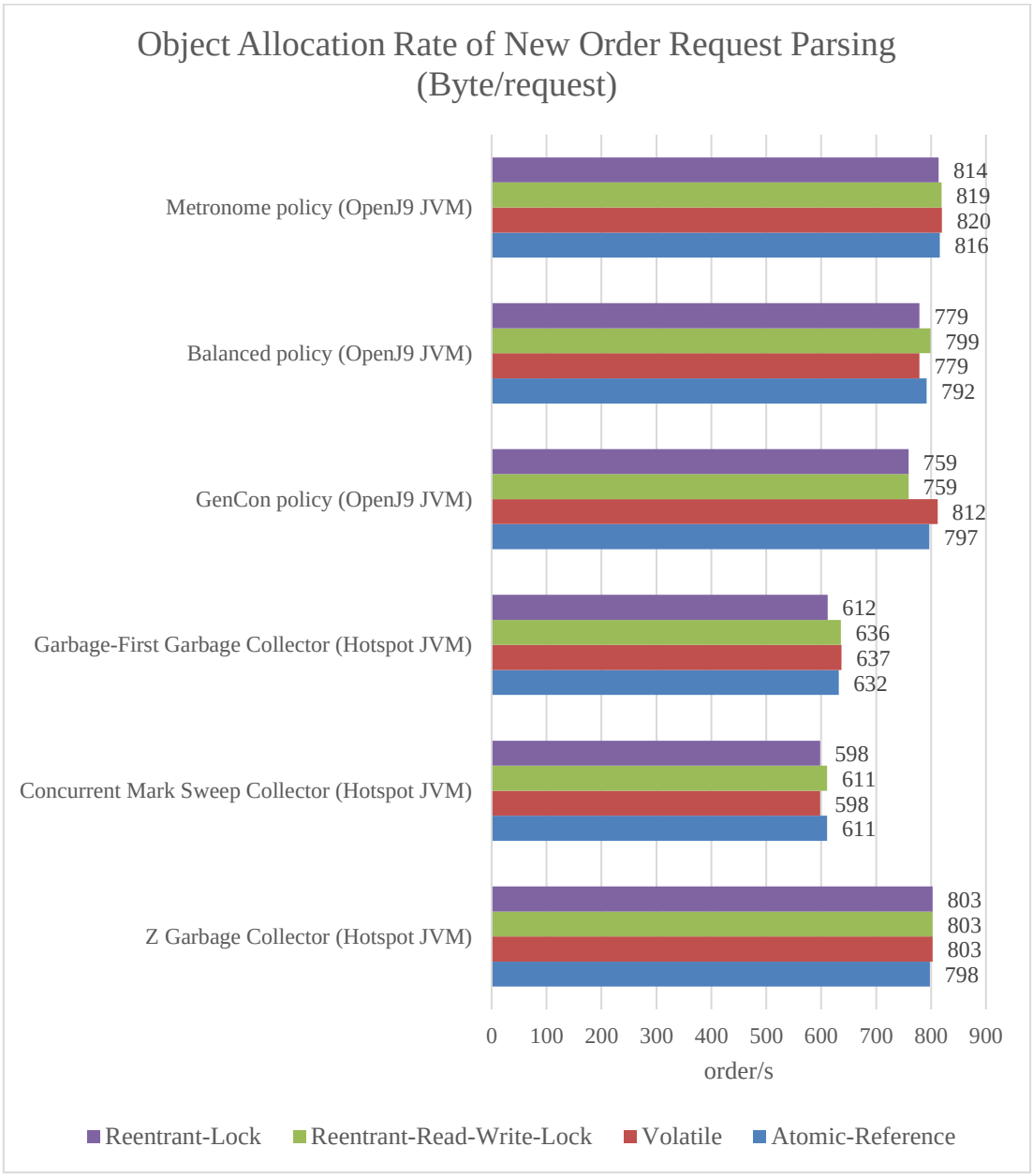

Chart 6.2.1(2) Object Allocation Rate per order placement of Order Matching Engine with different synchronization techniques in the Market Data Store (Remark: Lower values mean better results)

Within the same garbage collector settings, object allocation rates of order placement operations with different synchronization techniques in the Market Data Store had no significant variations.

When running in the Oracle Hotspot JVM, the object allocation rates of order placement

operations using Z Garbage Collector was higher than those using Garbage-First Garbage Collector and Concurrent Mark Sweep Garbage Collector.

When running in the Eclipse OpenJ9 JVM, the object allocation rates of order placement operations using those chosen garbage collection policies had no significant variations.

3. Garbage Collection Time and Count

Average Time Spent per Garbage Collection (ms/count) **Garbage Collection Count (count)** **GC time (ms)**	**Atomic-Reference**	**Volatile**	**Reentrant-Read-Write-Lock**	**Reentrant-Lock**
Z Garbage Collector (HotSpot JVM)	0.6 ms/count 15 counts 9 ms	0.533 ms/count 15 counts 8 ms	0.667 ms/count 15 counts 10 ms	0.5 ms/count 14 counts 7 ms
Concurrent Mark Sweep Collector (HotSpot JVM)	24 ms/count 103 counts 2,521 ms	25 ms/count 103 counts 2,558 ms	25 ms/count 107 counts 2,660 ms	25 ms/count 106 counts 2,625 ms
Garbage-First Garbage Collector (HotSpot JVM)	105 ms/count 18 counts	115 ms/count 14 counts	74 ms/count 27 counts 2,000 ms	97 ms/count 18 counts 1,750 ms

	1,882 ms	1,611 ms		
GenCon policy	59 ms/count	62 ms/count	59 ms/count	54 ms/count
(OpenJ9 JVM)	14 counts	14 counts	13 counts	13 counts
	820 ms	871 ms	773 ms	707 ms
Balanced policy	143	147	140	140
(OpenJ9 JVM)	ms/count	ms/count	ms/count	ms/count
	9 counts	9 counts	9 counts	9 counts
	1,287 ms	1,327 ms	1,264 ms	1,260 ms
Metronome policy	3 ms/count	3 ms/count	3 ms/count	3 ms/count
(OpenJ9 JVM)	4,650 counts	4,617 counts	4,645 counts	4,643 counts
	14,028 ms	13,911 ms	14,041 ms	13,969 ms

Table 6.1.2(3) Garbage collection statistics of the benchmark of order placement of Order Matching Engine with different synchronization techniques in the Market Data Store (Remark: Lower values mean better results)

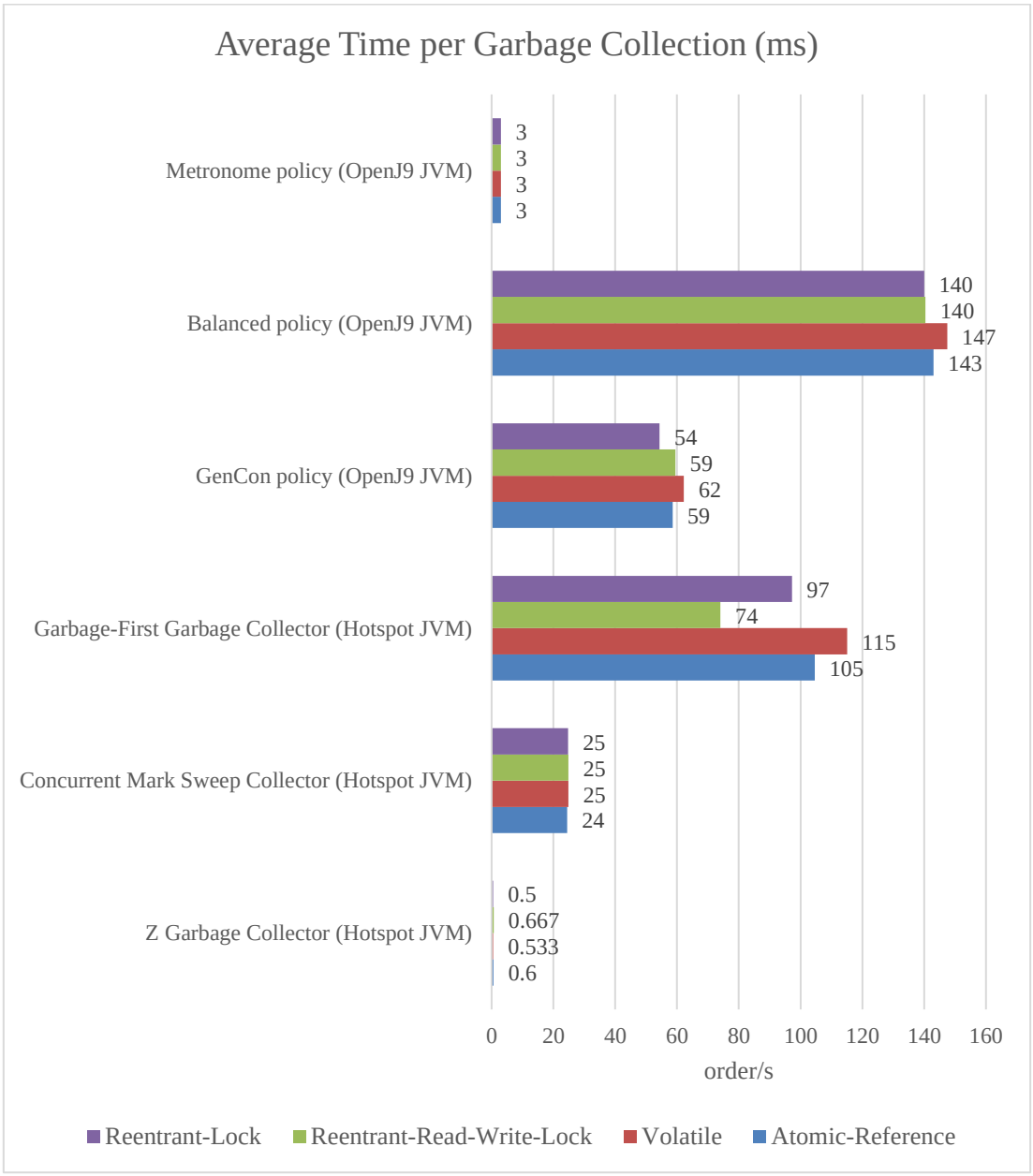

Chart 6.2.2(3A) Average time per garbage collection of order placement of Order Matching Engine with different synchronization techniques in the Market Data Store (Remark: Lower values mean better results)

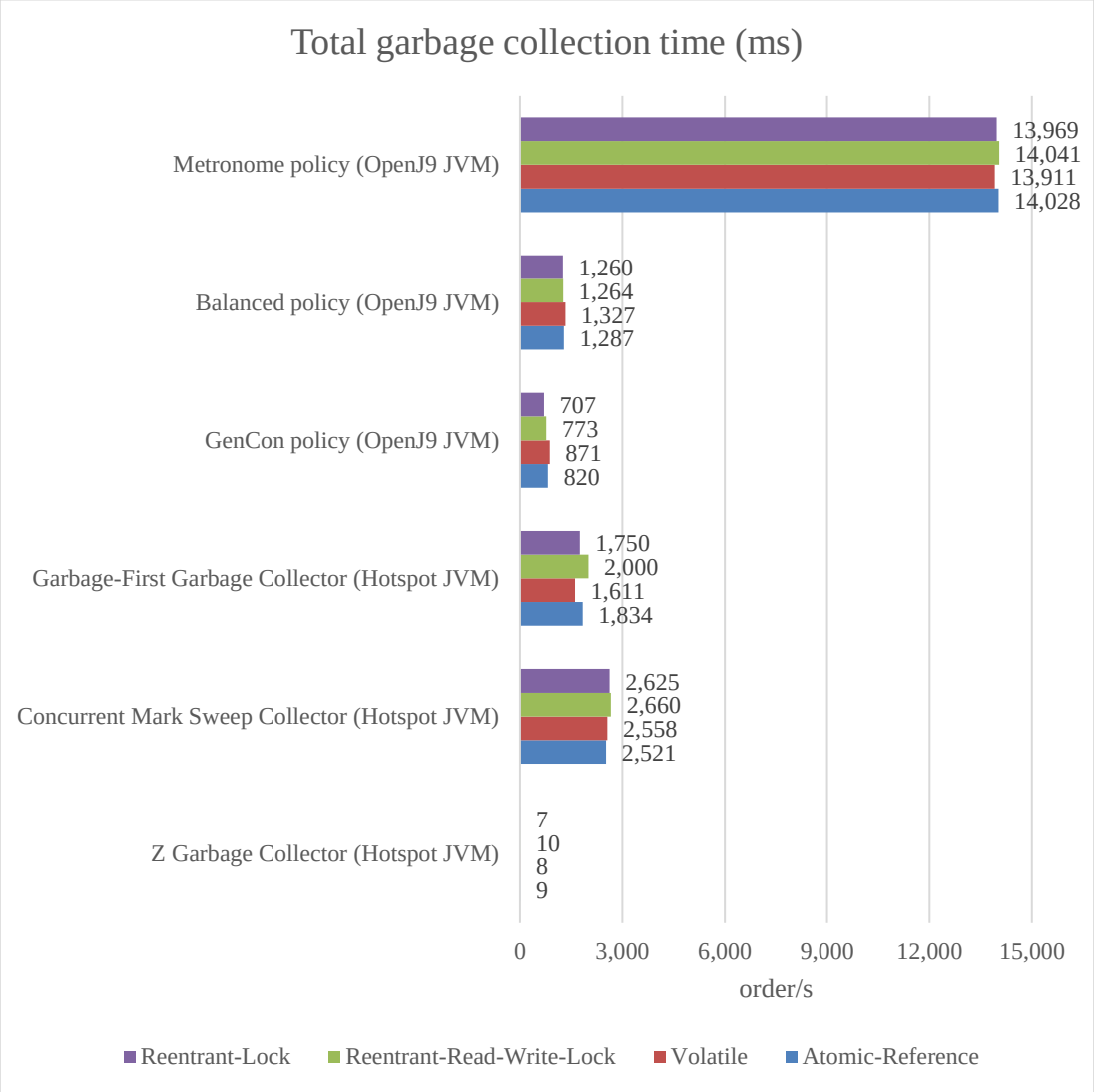

Chart 6.2.2(3B) Total garbage collection time during the benchmark of order placement of Order Matching Engine with different synchronization techniques in the Market Data Store (Remark: Lower values mean better results)

Z Garbage Collector of Oracle HotSpot JVM had the lowest time spent, 7~10 milliseconds, in garbage collections with relatively high throughput rate. Metronome policy had the highest time spent, 13,911~14,041 milliseconds, in garbage collections with the lowest throughput rate. Although Concurrent Mark Sweep Garbage Collector of Oracle HotSpot JVM had the highest throughput rate, it also had the second highest garbage collection time, 25,21~2,660 milliseconds.

6.3. Benchmark of FIX Gateway

We selected the two major functions, New Order Request Parsing and Execution Report Generation, of the FIX Gateway to do the benchmark. These two functions are the most common functions called in the trading execution chain of every order. The same functions of the popular open source library QuickFIX/J were also benchmarked with the same tests for comparisons.

6.3.1. Benchmark of New Order Request Parsing

The New Order Request Parsing function of the FIX Gateway and QuickFIX/J were benchmarked with the same set of testing messages. The operation was to parse a string of new order request to create an order object.

The results were as the following.

1. Average Latency

Average Latency (us/request)	**FIX Gateway**	**QuickFIX/J**
Z Garbage Collector (HotSpot JVM)	1.998 us/request	1.953 us/ request
Concurrent Mark Sweep Collector (HotSpot JVM)	2.078 us/request	2.276 us/request
Garbage-First Garbage Collector (HotSpot JVM)	1.657 us/request	2.112 us/request
GenCon policy (OpenJ9 JVM)	3.495 us/request	3.458 us/request
Balanced policy (OpenJ9 JVM)	4.771 us/request	4.834 us/request
Metronome policy (OpenJ9 JVM)	5.362 us/request	5.747 us/request

Table 6.3.1(1) Average Latency of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

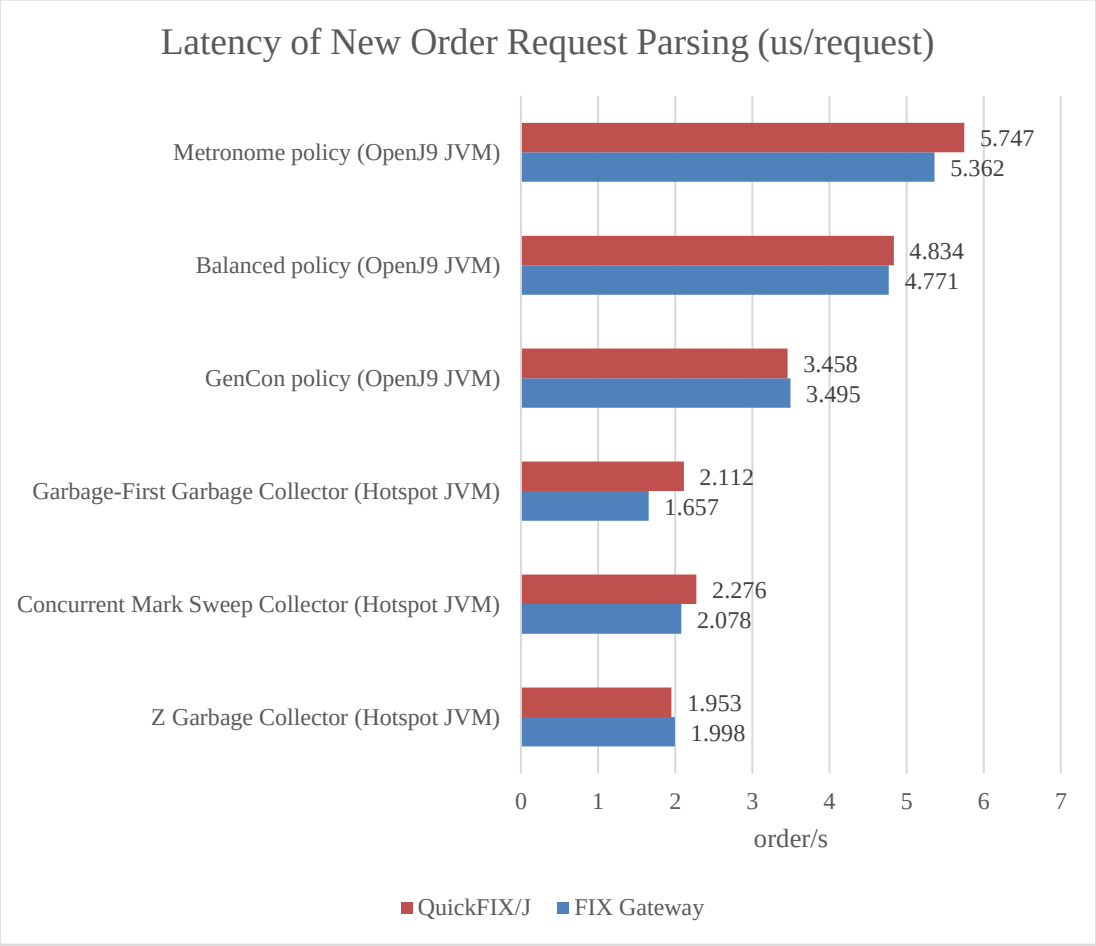

Chart 6.3.1(1) Average Latency in New Order Request Parsing of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

Our FIX Gateway had slightly lower average time spent in New Order Request Parsing than QuickFIX/J in four benchmarks out of six. In the two benchmarks in which FIX Gateway had longer average latency, the differences were only 40~45 nanoseconds which was very insignificant.

The average time spent in New Order Parsing function of both FIX Gateway and QuickFIX/J running in Oracle HotSpot JVM were all more than 40% shorter than those running in Eclipse OpenJ9 Java Virtual Machine. This suggested that Oracle HotSpot JVM had lower latency in program executions.

2. Object Allocation Rate

Object Allocation Rate (byte/request)	**FIX Gateway**	**QuickFIX/J**
Z Garbage Collector (HotSpot JVM)	3,632 byte/request	5,128 byte/request
Concurrent Mark Sweep Collector (HotSpot JVM)	2,587 byte/request	3,643 byte/request
Garbage-First Garbage Collector (HotSpot JVM)	2,587 byte/request	3,667 byte/request
GenCon policy (OpenJ9 JVM)	2,560 byte/request	3,930 byte/request
Balanced policy (OpenJ9 JVM)	2,558 byte/request	3,908 byte/request
Metronome policy (OpenJ9 JVM)	2,506 byte/request	3,747 byte/request

Table 6.3.1(2) Object allocation rate of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

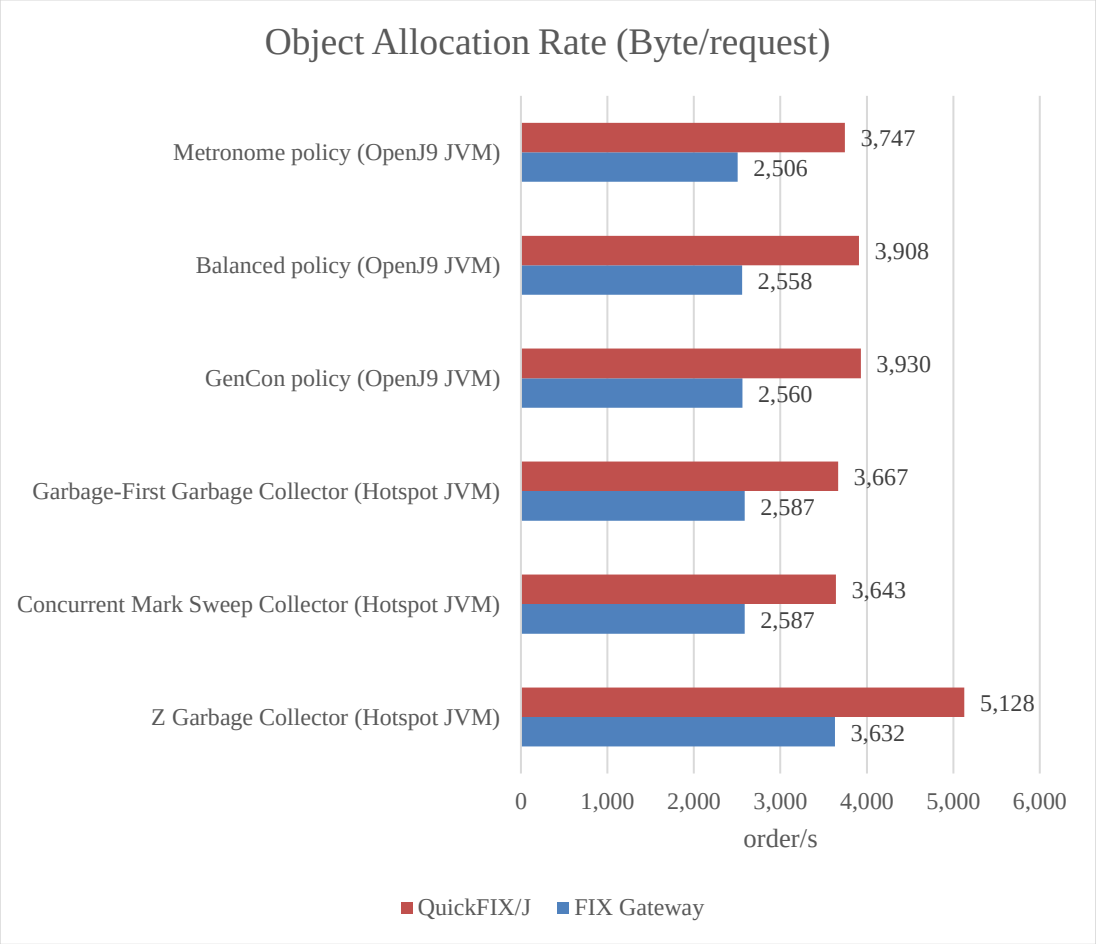

Chart 6.3.1(2) Object Allocation Rate of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

Over all benchmarks, the object allocation rates of New Order Request Parsing function of the FIX Gateway were only 65%~70% of that of the QuickFIX/J.

When comparing garbage collectors, the object allocation rate of the benchmarks running on Oracle HotSpot JVM with Z Garbage Collector had higher values than other benchmarks running on other settings. This might suggest that Z Garbage Collector would increase the size of objects.

3. Garbage Collection Time and Count

Average Time Spent per Garbage Collection (ms/count) **Garbage Collection Count (count)** **GC time (ms)**	**FIX Gateway**	**QuickFIX/J**
Z Garbage Collector (HotSpot JVM)	0.576 ms/count 1,911 counts 1,100 ms	0.593 ms/count 2,006 counts 1,189 ms
Concurrent Mark Sweep Collector (HotSpot JVM)	12 ms/count 10,568 counts 124,349 ms	12 ms/count 13,653 counts 159,273 ms
Garbage-First Garbage Collector (HotSpot JVM)	4 ms/count 711 counts 2,741 ms	4 ms/count 804 counts 3,107 ms
GenCon policy (OpenJ9 JVM)	0.386 ms/count 856 counts 330 ms	0.357 ms/count 1334 counts 476 ms
Balanced policy (OpenJ9 JVM)	8 ms/count 558 counts 4,759 ms	8 ms/count 841 counts 6,673 ms
Metronome policy (OpenJ9 JVM)	3 ms/count 27,564 counts 82,672 ms	3 ms/count 39,354 counts 117,571 ms

Table 6.3.1(3) Garbage collection statistics of the benchmark of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

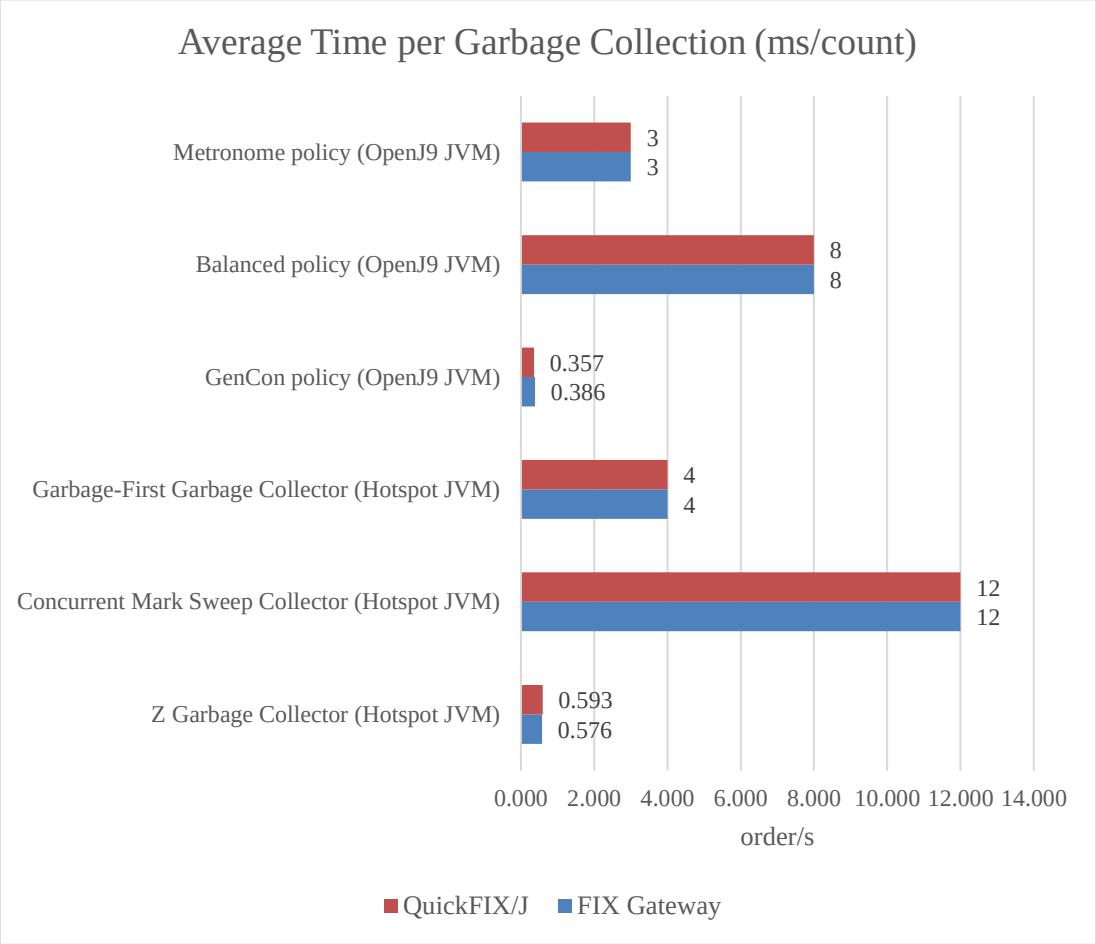

Chart 6.3.1(3A) Average time per garbage collection during the benchmark of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

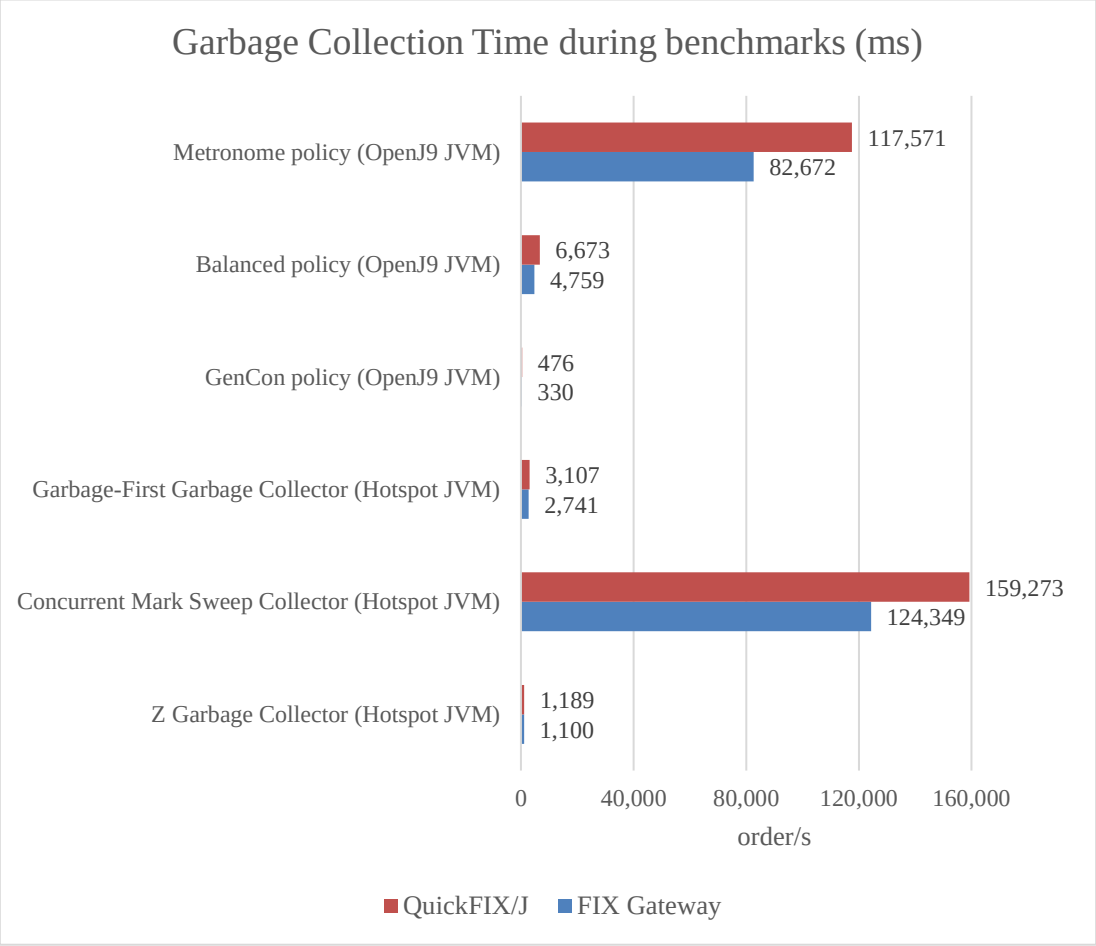

Chart 6.3.1(3B) Total garbage collection time during the benchmark of New Order Request Parsing by FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

Benchmarks of new order request parsing operation using FIX Gateway all had shorter time spent in garbage collection than those using QuickFIX/J. The average time spent per garbage collection were the same no matter using FIX Gateway or QuickFIX/J within the same garbage collection setting.

The GenCon policy of the Eclipse OpenJ9 JVM and the Z Garbage Collector of the Oracle Hotspot JVM were the top two garbage collection settings having the lowest garbage collection time. The Concurrent Mark Sweep Garbage Collector of the Oracle Hotspot JVM and the Metronome policy of the Eclipse OpenJ9 JVM had the highest

garbage collection time.

6.3.2. Benchmark of Execution Reports Generation

The execution report generation function of our FIX Gateway and QuickFIX/J were measured with the same set of input parameters.

Tag ID	Value
8 (FIXVersion)	FIX.4.4
49 (SenderCompID)	TS
56 (TargetCompID)	CLIENT1
35 (MsgType)	8
37 (OrderID)	O123456789
17 (ExecID)	1111
150 (ExecType)	F
11 (ClOrderID)	C056
39 (OrderStatus)	1 (Partially Filled)
54 (Side)	1 (Buy)
151 (LeavesQty)	100
14 (CumQty)	1900
6 (AvgPx)	143.4
38 (OrderQty)	2000
55 (Symbol)	AAPL
32 (LastQty)	1000
31 (LastPx)	143.42

Table 6.3.2 Parameters used for benchmarking the execution report generation functions of FIX Gateway and QuickFIX/J

The results were as the following.

1. Average Latency

Average Latency (microsecond/ report)	**FIX Gateway**	**QuickFIX/J**
Z Garbage Collector (HotSpot JVM)	1.867 us/report	3.969 us/report
Concurrent Mark Sweep Collector (HotSpot JVM)	1.851 us/report	3.196 us/report
Garbage-First Garbage Collector (HotSpot JVM)	1.882 us/report	3.430 us/report
GenCon policy (OpenJ9 JVM)	3.695 us/report	5.214 us/report
Balanced policy (OpenJ9 JVM)	5.069 us/report	6.980 us/report
Metronome policy (OpenJ9 JVM)	5.555 us/report	8.381 us/report

Table 6.3.2(1) Average time spent of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

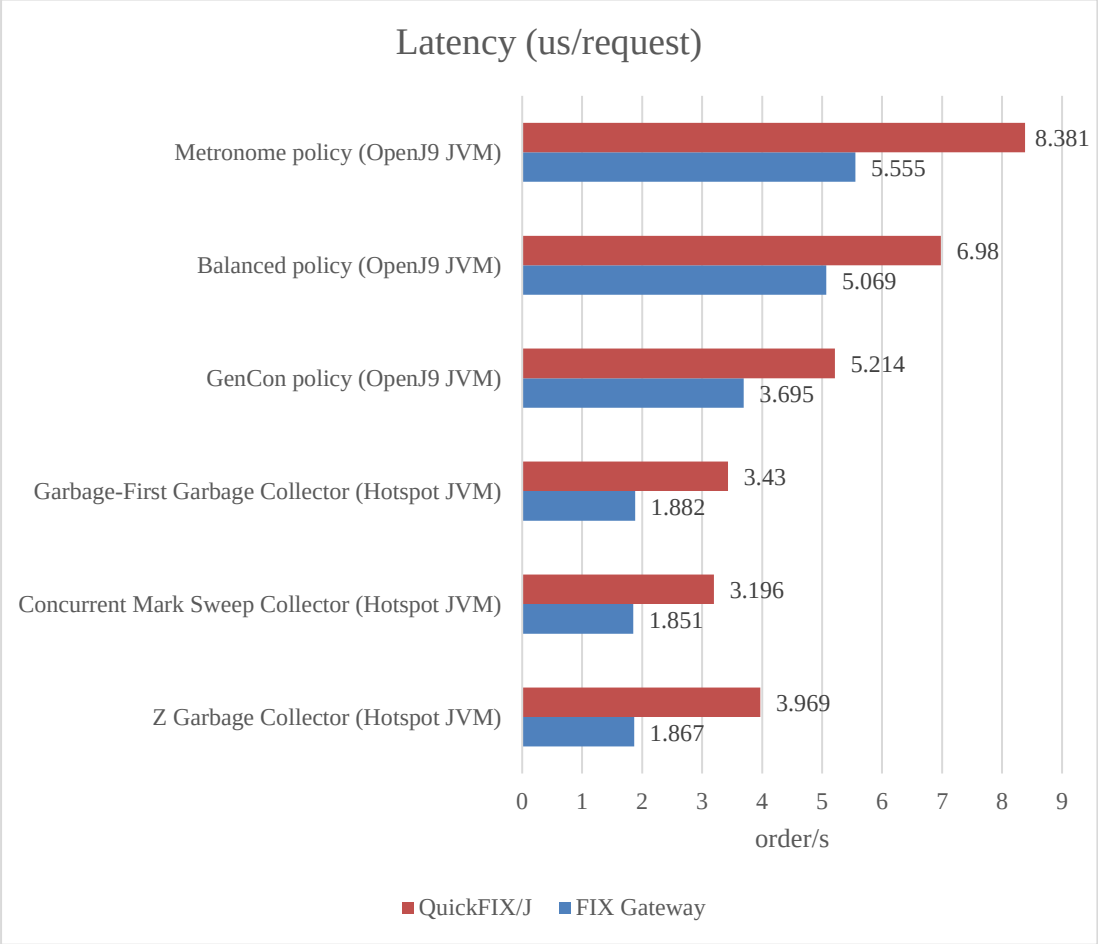

Chart 6.3.2(1) Average Latency of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

In all the benchmarks, the latencies on the execution report generation functions of our FIX Gateway are 47%~73% shorter than those of the QuickFIX/J.

The latencies on the operation of generating an Execution Report by FIX Gateway running on Oracle HotSpot JVM were 1.851~1.882 microseconds, but those running on the Eclipse OpenJ9 JVM were 3.695~5.55 microseconds.

2. Object Allocation Rate

Object Allocation Rate (byte/report)	FIX Gateway	QuickFIX/J

Z Garbage Collector (HotSpot JVM)	1,592 byte/report	4,544 byte/report
Concurrent Mark Sweep Collector (HotSpot JVM)	1,272 byte/report	2,920 byte/report
Garbage-First Garbage Collector (HotSpot JVM)	1,248 byte/report	3,016 byte/report
GenCon policy (OpenJ9 JVM)	1,796 byte/report	3,601 byte/report
Balanced policy (OpenJ9 JVM)	1,859 byte/report	3,660 byte/report
Metronome policy (OpenJ9 JVM)	1,845 byte/report	3,639 byte/report

Table 6.3.2(2) Object Allocation Rate per the execution report generation functions of FIX Gateway and QuickFIX/J

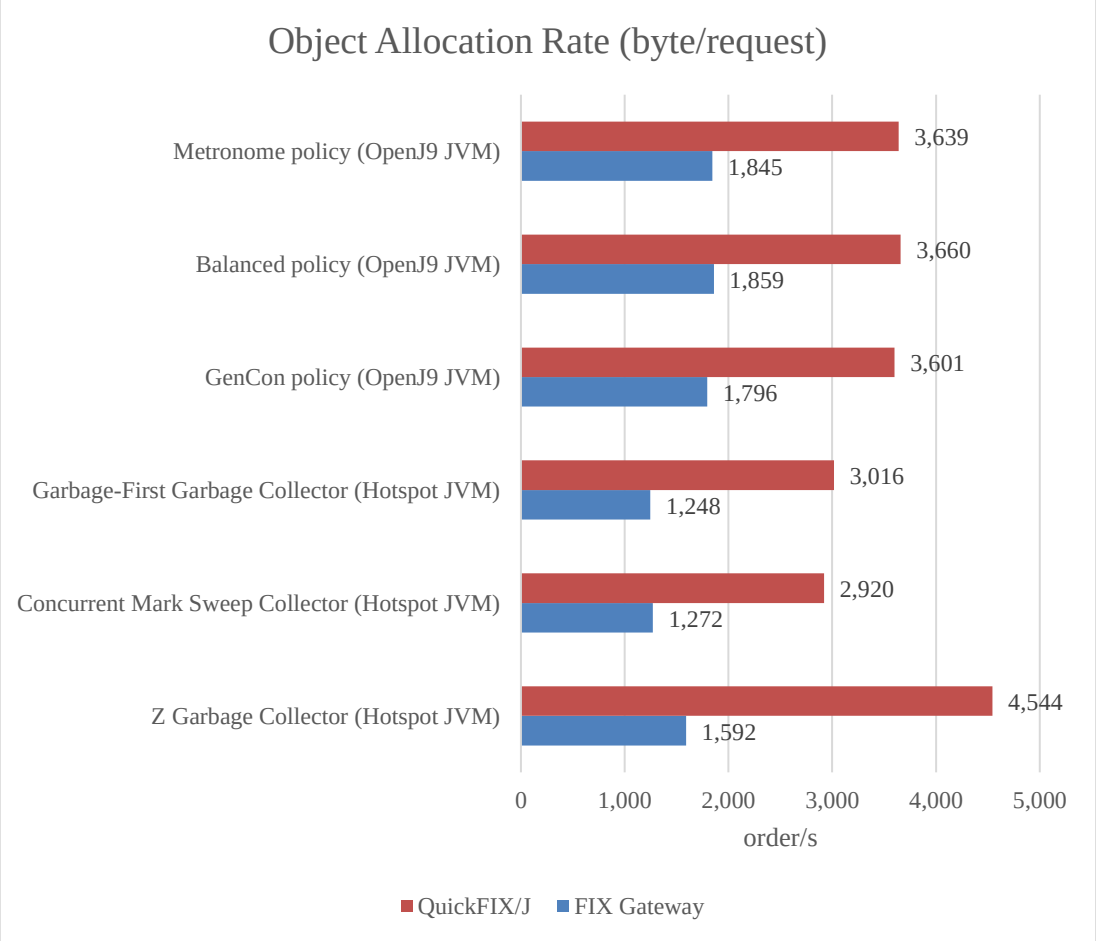

Chart 6.3.2(2) Object Allocation Rate of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

The execution report generation function of the FIX Gateway allocated 1600~3000 bytes less than that of the QuickFIX/J in all JVM and garbage collector pairs, so we could expect fewer garbage collections will occur when using FIX Gateway.

3. Garbage Collection Time and Count

Average Time Spent per Garbage Collection (ms/count) **Garbage Collection Count (count)** **GC time (ms)**	**FIX Gateway**	**QuickFIX/J**
Z Garbage Collector (HotSpot JVM)	0.532 ms/count 427 counts 227 ms	0.585 ms/count 733 counts 429 ms
Concurrent Mark Sweep Collector (HotSpot JVM)	11 ms/count 5768 counts 66111 ms	12 ms/count 7846 counts 90264 ms
Garbage-First Garbage Collector (HotSpot JVM)	4 ms/count 304 counts 1225 ms	4 ms/count 409 counts 1643 ms
GenCon policy (OpenJ9 JVM)	0.427 ms/count 569 counts 243 ms	0.410 ms/count 830 counts 340 ms

Balanced policy (OpenJ9 JVM)	9 ms/count 379 counts 3546 ms	8 ms/count 562 counts 4748 ms
Metronome policy (OpenJ9 JVM)	3 ms/count 19057 counts 56,666 ms	3 ms/count 26835 counts 79965 ms

Table 6.3.2(3) Garbage collection statistics of the benchmark of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

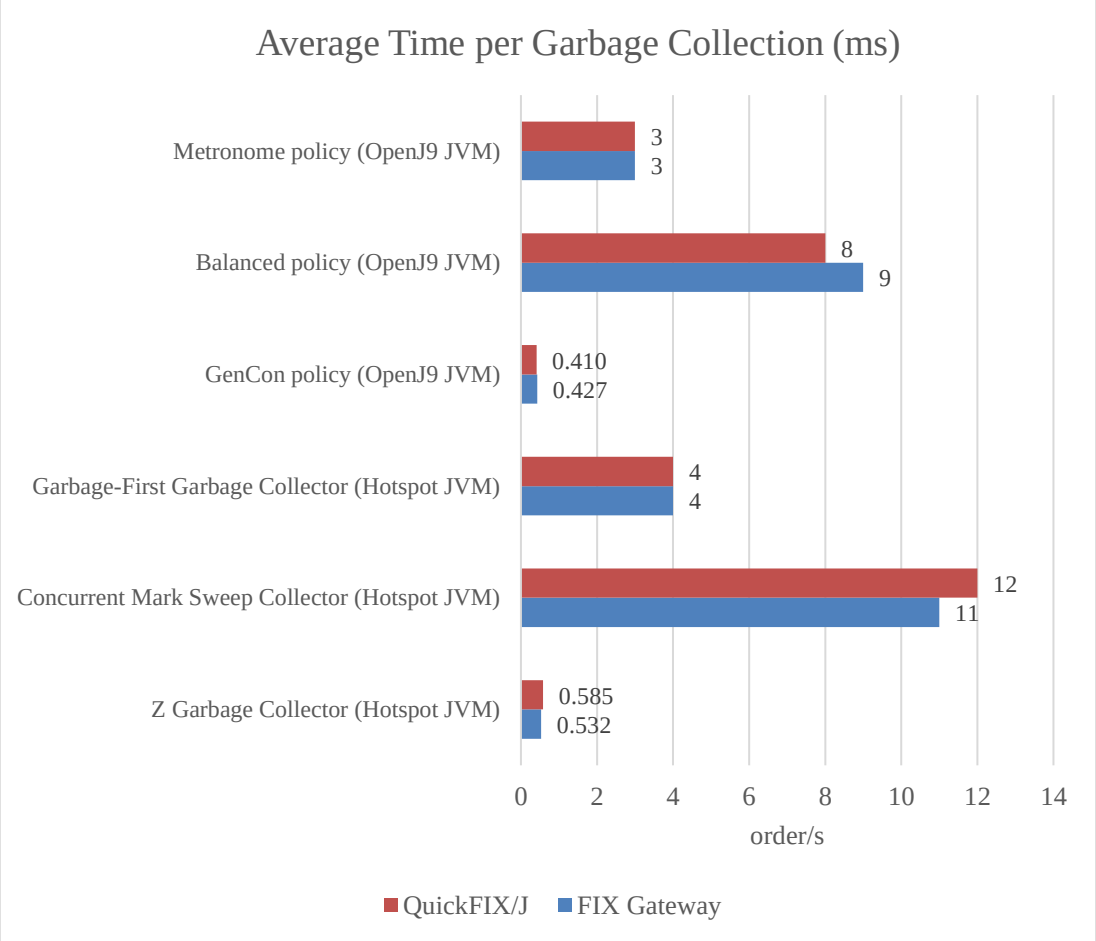

Chart 6.3.2(3A) Average time per garbage collection during the benchmark of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark:

Lower values mean better results)

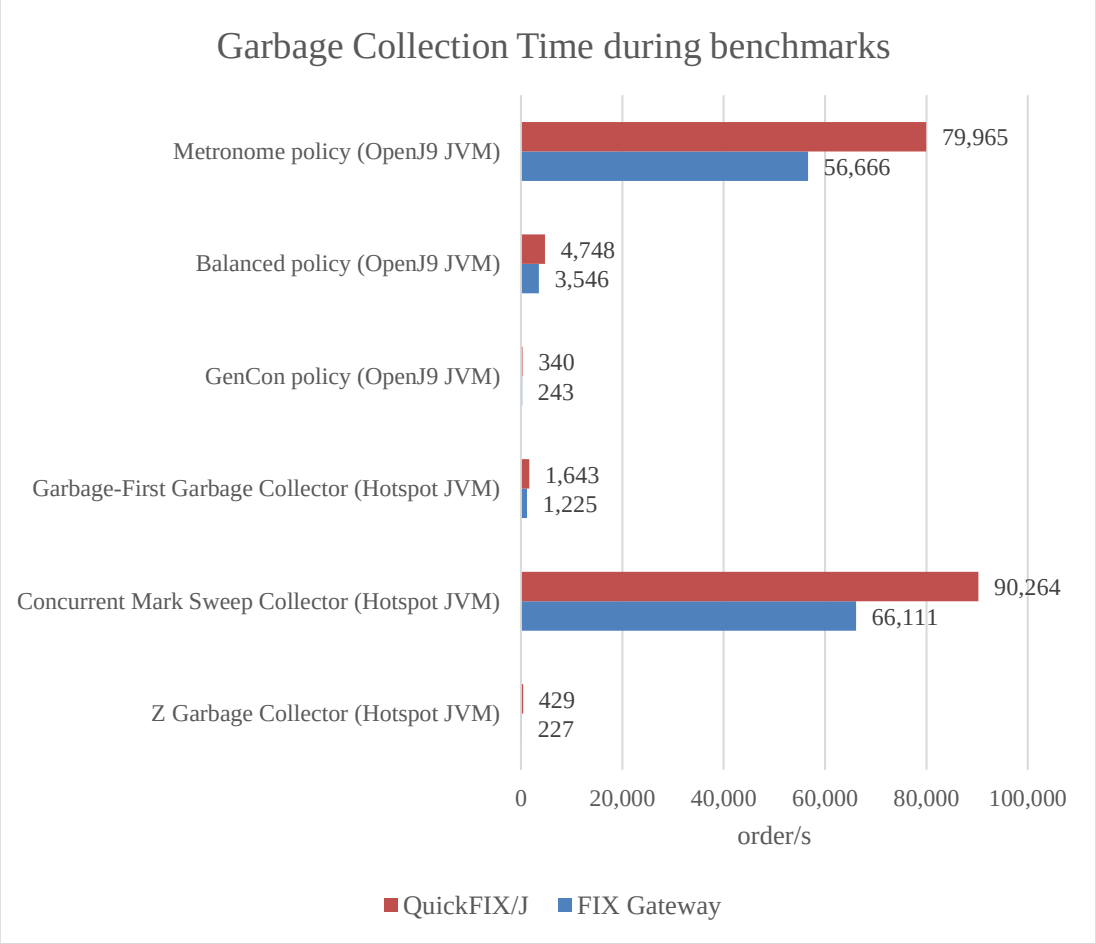

Chart 6.3.2(3B) Total garbage collection time during the benchmark of the execution report generation functions of FIX Gateway and QuickFIX/J (Remark: Lower values mean better results)

Benchmarks of execution report generating operations using FIX Gateway all had shorter time spent in garbage collection than those using QuickFIX/J. The average time spent per garbage collection were the same no matter using FIX Gateway or QuickFIX/J within the same garbage collection setting.

The GenCon policy of the Eclipse OpenJ9 JVM and the Z Garbage Collector of the Oracle Hotspot JVM were the top two garbage collection settings having the lowest

garbage collection time. The Concurrent Mark Sweep Garbage Collector of the Oracle Hotspot JVM and the Metronome policy of the Eclipse OpenJ9 JVM had the highest garbage collection time.

6.4. Benchmark of the Overall Trading System

The Trading System formed by combining the Order Matching Engine and the FIX Gateway were benchmarked.

The Order flow in the benchmarked trading system was as the following:

1. Parse a New Order Request and convert it into an order by the FIX Engine, either the FIX Gateway or the QuickFIX/J.
2. Pass the order generated in step 1 into the Order Matching Engine for matching.
3. Generate one or more execution reports.
    a. If the order matched successfully, generate execution reports with either filled statuses or partially filled statuses for all matched orders.
    b. If the current order is market order and it is not fully filled, generate an execution report with a cancelled status for it.
    c. If the current order is limit order and it is not matched by any orders, generate an execution report with a new status for it.

We chose the following settings to do benchmarks for comparisons.

	Trading System A	**Trading System B**	**Trading System C**	**Trading System D**
Java Virtual Machine	Oracle HotSpot	Oracle HotSpot	Oracle HotSpot	Oracle HotSpot
Garbage Collector	Z Garbage Collector	Z Garbage Collector	Z Garbage Collector	Z Garbage Collector

FIX Engine	FIX Gateway	QuickFIX/J library	FIX Gateway	FIX Gateway
Order List Structure	Order-Internally-Linked List	Order-Internally-Linked List	Standard ArrayDeque	Order-Internally-Linked List
Thread-safe Technique for Market Data	Reentrant-Lock	Reentrant-Lock	Reentrant-Lock	Atomic-Reference

Table. 6.4(1) Settings of the Trading Systems benchmarked

The Trading System A used the components which produced the highest throughputs and lowest latencies in the previous benchmarks. The Trading System B, C and D were control setups. The Trading System B had the same setting of the Trading System A except having QuickFIX/J as the FIX engine. The Trading System C had the same setting of the Trading System A except having the Standard ArrayDeque as the Order List implementation. The Trading System D had the same setting of the Trading System A except having the AtomicReference approach as the thread-safe technique for market data.

The result was as the following.

6.4.1. Throughput, latency and object allocation rate

	Trading System A	**Trading System B**	**Trading System C**	**Trading System D**
Average Throughput (order/ second)	130,236 order/s	77,651 order/s	112,817 order/s	113,266 order/s
Average Latency	7.678	12.878	8.864	8.829

(us/order)	us/order	us/order	us/order	us/order
Object Allocation Rate (byte/order)	7,015 byte/order	13,598 byte/order	7,535 byte/order	6,242 byte/order
Garbage Collection Count	321 counts	346 counts	165 counts	323 counts
Total Garbage Collection Time (ms)	209 ms	235 ms	149 ms	219 ms
Average Time Spent per Garbage Collection (ms/count)	0.651 ms/count	0.679 ms/count	0.903 ms/count	0.678 ms/count

Table. 6.4.1(1) Benchmark Results of our Trading System with different settings

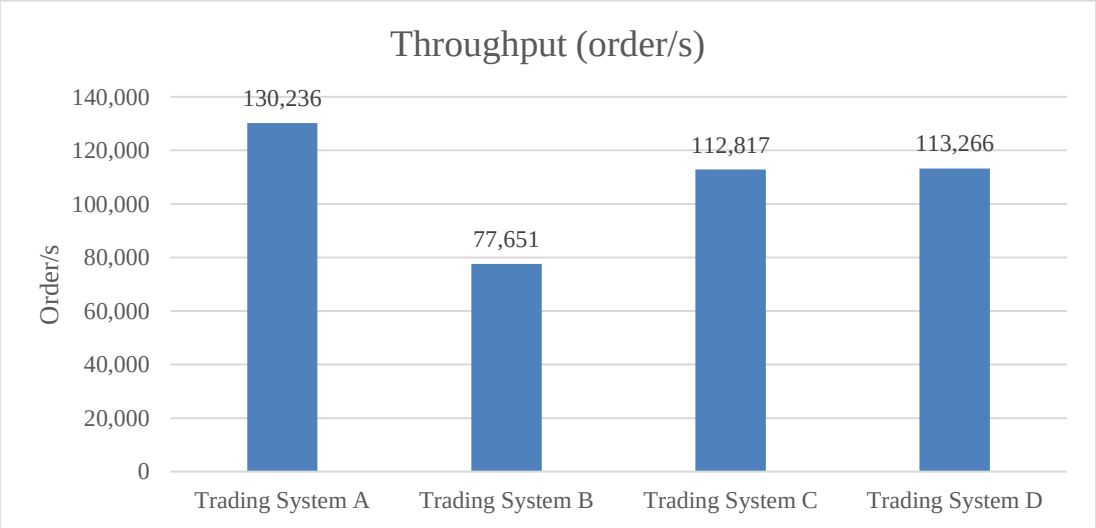

Chart. 6.4.1(1) Throughput of our Trading System with different settings (Remark: Higher values mean better results)

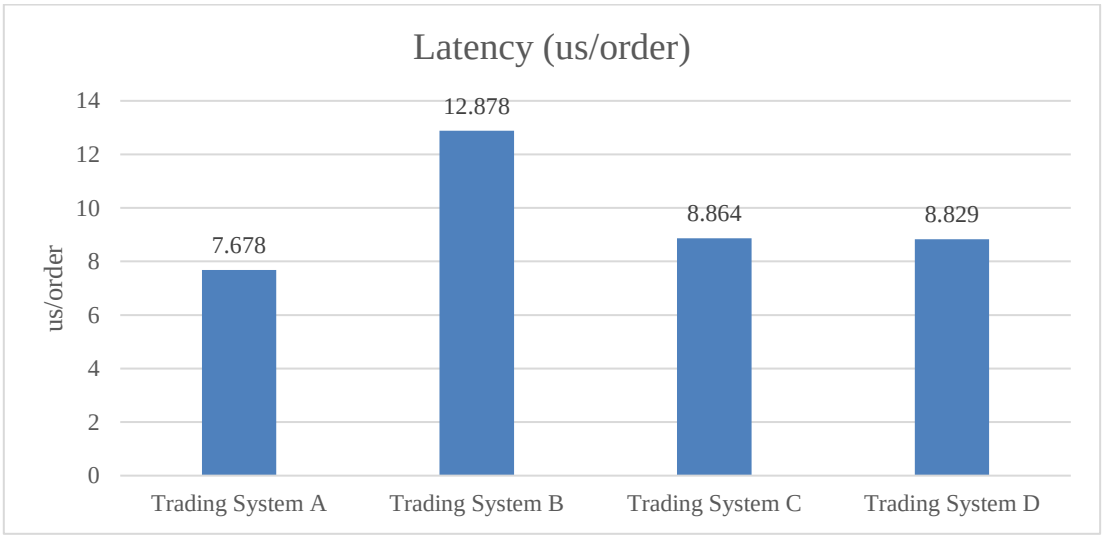

Chart. 6.4.1(2) Latency of our Trading System with different settings (Remark: Lower values mean better results)

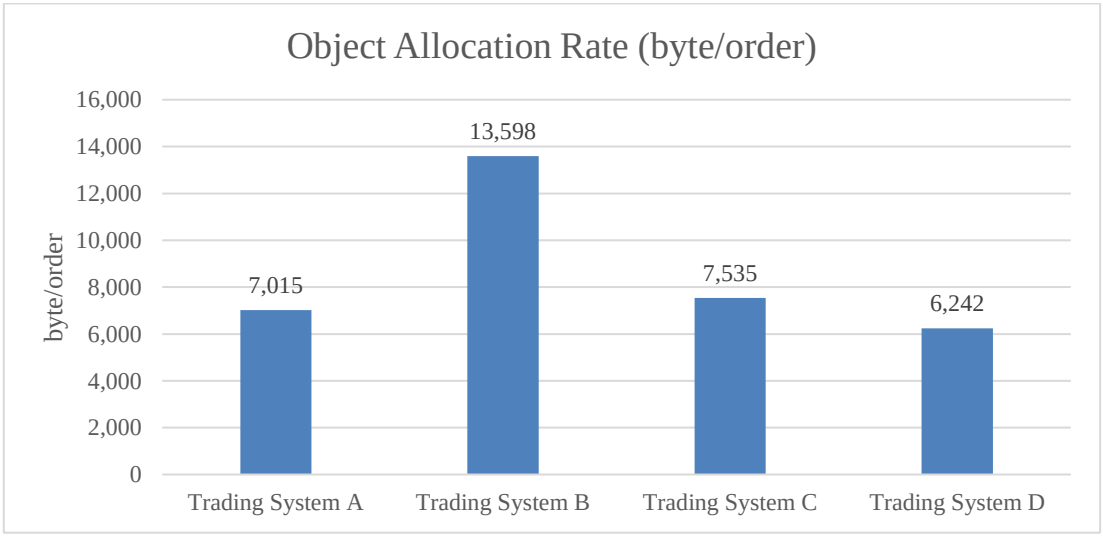

Chart. 6.4.1(3) Object Allocation Rate of our Trading System with different settings (Remark: Lower values mean better results)

The Trading System A had the highest throughput and the lowest latency among all the setups. The Trading System B, which used QuickFIX/J, had 40.4% reduction of the throughput. The Trading System C, which used the Standard ArrayDeque, had 13.4% reduction of the throughput. The Trading System D, which used the AtomicReference,

had 13.0% reduction of the throughput.

6.4.2. CPU-intensive methods in Trading System A and B

Rank	CPU Usage	Method	Component
1	9.9%	*FIXExecutionReportMessage.toMessageString*	FIX Gateway
2	9.5%	*FIXMessage.setTag*	FIX Gateway
3	8.8%	*java.util.HashMap.resize*	FIX Gateway
4	7.0%	*java.lang.AbstractStringBuilder.append*	FIX Gateway
5	4.7%	*java.util.regex.Pattern$BmpCharProperty.match*	FIX Gateway

Table. 6.4.2(A) Top 5 methods occupied the longest CPU time in Trading System A

Rank	CPU Usage	Method	Component
1	11.1%	*java.text.DecimalFormat.subformat*	QuickFIX/J
2	10.8%	*java.text.DecimalFormat.format*	QuickFIX/J
3	7.7%	*quickfix.Field.toString*	QuickFIX/J
4	4.5%	*quickfix.Message.parseHeader*	QuickFIX/J
5	4.2%	*quickfix.Message.toString*	QuickFIX/J

Table. 6.4.2(B) Top 5 methods occupied the longest CPU time in Trading System B

All the top five methods occupied the longest CPU time in the Trading System A and B were called from the FIX engine, which was either our FIX Gateway, or the QuickFIX/J. The top five methods occupied around 38%~40% CPU time in each trading system.

7. Discussion

7.1. Factors

According to the benchmarks we produced, we ranked the importance of factors affecting the performance of our Java trading system as the following:

Rank	Factor
1	Choice of Java Virtual Machines
2	Choice of Garbage Collectors
3	Implementation of Messaging (FIX) Engine
4	Implementation of Data Structure in Order List
5	Choice of synchronization techniques

Table. 7.1 Impact of factors affecting the throughput and the latency of the Java trading system

7.2. Choice of Java Virtual Machines

In the overall benchmarks, Oracle HotSpot JVM allowed higher throughput and lower latency in program executions than the Eclipse OpenJ9 JVM. For example, in the benchmarks of Execution Report Generation, the average time spent per operations of FIX Gateway running in Oracle HotSpot JVM were only 1.851~1.882 microseconds, but those running in Eclipse OpenJ9 JVM were 3.695~5.555 microseconds, which were about 96%~200% increase.

As different implementations of JVM generally follow the same specification of the same Java version, we could change to use a well-implemented JVM to significantly improve the application performance with no cost of re-development. However, we should pay attention to the potential differences, as our results of the benchmark of thread-safe techniques on market data suggested that different JVM implementations might have different execution behaviors on the thread-safe techniques.

We would recommend choosing Oracle HotSpot JVM over the Eclipse OpenJ9 Java Virtual Machine.

7.3. Choice of Garbage Collectors

The choices of Garbage Collectors directly affected the garbage collection time spent during the runtime of the Java application as different garbage collectors had their own garbage collection mechanisms, such as marking and collecting. Longer garbage collection time would add longer pauses to the application threads and thus increase the latencies of order executions. In the benchmark of Order Matching Engine with order list structures and thread-safe techniques, changing garbage collectors when running on Oracle HotSpot Virtual Machine could increase the throughput by 24.0%~88.2%, and changing those when running on the Eclipse OpenJ9 JVM could increase the throughput by 384.7%~421.4%.

We also found that the Z Garbage Collector of Oracle HotSpot JVM had the extremely short total garbage collection time and the lowest average time spent per garbage collection in the benchmarks of Order Matching Engine with order list structures. The average time spent per garbage collection of the Z Garbage Collector was only 0.533~0.6 milliseconds, while those of the Concurrent Mark Sweep Garbage Collector and Garbage-First Garbage Collector were 23~26 milliseconds and 79~117 milliseconds respectively. All garbage collection mechanisms contain stop-the-world phases. In the trading system, if an order arrived at the time of garbage collection work occurring, the waiting time of the order due to the stop-the-world phases would add to the latency of the order. Z Garbage Collector is the best choice of garbage collector in Oracle HotSpot JVM.

7.4. Implementation of the Messaging (FIX) Engine

Messaging engine was responsible for parsing inbound messages and generating outbound messages. In our Java trading system, the messaging engine was FIX Gateway which complied with the standard FIX Protocol 4.4.

Each inbound order request must be parsed by the Messaging Engine before inputting the order information into the Order Matching Engine. Each order might have zero, one or multiple fills. Each fill would be converted into an outbound execution report message by the Messaging Engine.

7.4.1. Effect of latency in Messaging Engine

The time spent on an order placement could be deduced by the following calculations:

Let P be the time spent in order request parsing, M be the time spent in Order Matching Engine, E be the time spent in generating execution report and n be the number of fills

There were two possible situations of an order:

Situation 1: an order could not be filled immediately

Total time spent

= Time spent in FIX order request parsing by FIX Engine

+ Time spent in Order Matching Engine

+ Time spent in generating FIX Execution Report with "New" or "Cancel" status

$= P + M + E$

Situation 2: an order could not be filled immediately

Total time spent

= Time spent in FIX order request parsing

+ Time spent in Order Matching Engine

*+ number of fills * (Time spent in generating execution report with "Filled" / "Partially Filled" status * involved parties (i.e. 2))*

= P + M + 2nE

From the above equations, we could see that the FIX Engine occupied two variables affecting the time spent in the order placement. In the situation 2, the greater number of fills of an order, the more execution reports generated by the FIX engine, and thus the higher impacts of latency of the execution report generation operation to the overall latency of an order placement operation.

From the previous benchmark results running on Oracle HotSpot JVM with the Z Garbage Collector, the order placement function using Order-Internally-Linked List took 2.035 microseconds in average.

According to the benchmarks of the QuickFIX/J running on Oracle HotSpot JVM with Z Garbage Collector, the new order single parsing and the execution report generation by QuickFIX/J took 1.953 microsecond and 3.969 microsecond in average respectively. Therefore, the time spent in an order placement of situation 1 would be (1.953 + 2.035 + 3.969 =) 7.957 microseconds, and the time spent in an order placement of situation 2 would be (1.953 + 2.035 + 2n* 3.969=) 3.988 + 7.938n microseconds.

According to the benchmarks of the FIX Gateway running on Oracle HotSpot JVM with Z Garbage Collector, the new order single parsing and the execution report

generation by FIX Gateway took 1.99 microsecond and 1.87 microsecond in average respectively. Therefore, for trading system running on Oracle HotSpot JVM with Z Garbage Collector and FIX Gateway, the time spent in an order placement of situation 1 would be (1.998 + 2.035 + 1.87 =) 5.903 microseconds, and the time spent in an order placement of situation 2 would be (1.998 + 2.035 + 2n* 1.87=) 4.033+3.74n microseconds.

	Time spent per order placement		
Number of fills	**Trading System using QuickFIX/J**	**Trading System using FIX Gateway**	**Latency Reduction by FIX Gateway**
0	7.957 us	5.903 us	25.81%
1	11.926 us	7.773 us	34.82%
2	19.864 us	11.513 us	42.04%
3	27.802 us	15.253 us	45.14%
4	35.74 us	18.993 us	46.86%
5	43.678 us	22.733 us	47.95%
6	51.616 us	26.473 us	48.71%
7	59.554 us	30.213 us	49.27%
8	67.492 us	33.953 us	49.69%
9	75.43 us	37.693 us	50.03%
10	83.368 us	41.433 us	50.30%

Table 7.4.1 Time spent per order placement operation of the trading system using either QuickFIX/J or FIX Gateway with the number of fills ranging from 0 to 10

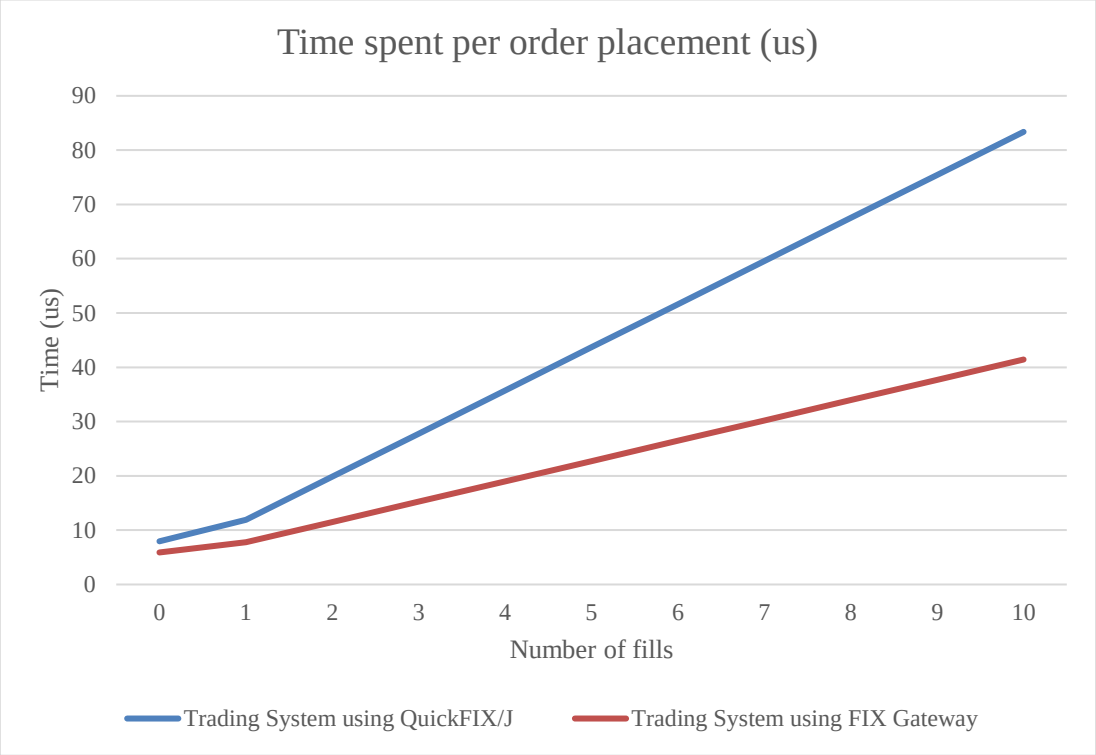

Chart 7.4.1 Time spent per order placement operation of the trading system using either QuickFIX/J or FIX Gateway with the number of fills ranging from 0 to 10

With the number of fills increasing, the difference in time spent of an order placement between the Trading System using FIX Gateway and the Trading System using QuickFIX/J increases. Using our implementation of FIX Gateway in our trading system could decrease the overall latency of order placement operation 25%~50%.

From the benchmark result of the overall trading system, replacing FIX Gateway by QuickFIX/J resulted in 40.4% throughput reduction.

7.4.2. Possible reasons of FIX Gateway outperforming QuickFIX/J in Execution Report Generation

For execution report generation, our FIX Gateway had >47% lower latency and >50% lower object allocation rate than QuickFIX/J.

During the process of generating execution reports, QuickFIX/J created many wrapping objects of the class *quickfix.StringField*, which held tag-value pairs of fields. The number of *quickfix.StringField* objects was equal to the number of fields in header, body and trailer in the FIX message. Those wrapping objects were then inserted to a *java.util.TreeMap* object of a *quickfix.FieldMap* object in a *quickfix.Message* object. There were two *quickfix.FieldMap* objects, which were created for header and body respectively, in each *quickfix.Message* object.

The approach of QuickFIX/J had three disadvantages.

Firstly, frequent object creations would heavily increase the burden of object cycling [51]. More objects allocated would lead to more garbage collections as less free memory would be available. Object allocations and garbage collections unavoidably generated some overheads in CPU and memory and thus slowed down the application performance.

Secondly, the insert operation of *java.util.TreeMap* had an average time complexity of O(log n) as it used red-black tree to search the position for the key to insert to maintain the sorting order of keys [41]. An execution report could involve up to thirteen fields in the body. With this time complexity, fields would be inserted at slower and slower rate for each execution report message.

Thirdly, we suspected that as maps involved complicated structures and thus had a greater number of lines of Java code, Just-In-Time (JIT) compilation would be more difficult to optimize them effectively when comparing with the approach simply using member variables in a class.

We designed our FIX Gateway to have a custom class *FIXExecutionReportMessage* to

hold tag-value pairs of fields for execution report generations. In this custom class, fields of the header and the body were all declared as member variables in either string type, char type or long type. During an execution report generation, values of fields were assigned to their corresponding member variables directly and the generated message string was built by appending hard-coded tags and assigned values to a re-allocated *StringBuilder* object in a pre-defined ordering of Java code.

With this approach in FIX Gateway, we successfully minimized the object allocation rate by not creating any wrapper objects for fields and re-using *StringBuilder* objects. This approach also stored char and long values directly with their original types instead of converting them to *String* type to minimize the chances of creating unnecessary String objects. The field value setting operation also only had O(1) time complexity as only variable assignments were involved and no sorting was involved. This architecture was much simpler than that of QuickFIX/J and we believed that it made Just-In-Time compilation easier to optimize.

7.5. Implementation of Data Structures in Order List

We developed three order list implementations in the Order Matching Engine. As the Order Matching Engine had to follow the time price priority among orders, there were three frequent operations on an order list. Firstly, head orders of order lists of best prices would be frequently queried to match with new orders. Secondly, head orders would be removed from order lists after a matching if they had no remaining open quantity anymore. Thirdly, new orders would be appended to the ends of order lists if it still had remaining open quantity.

Double-ended queue type was the suitable data structure for this scenario, as it had time complexity O(1) for the head element removal operation. We implemented and benchmarked three implementations of the order list having the double-ended queue logic: Order-Internally-Linked List, one backed by the Standard LinkedList and one backed by the Standard ArrayDeque.

From the benchmark results, the implementation using the Standard ArrayDeque had the lowest throughput among the implementations. We believed that it was because of the resize operation of the ArrayDeque. The Standard ArrayDeque was backed by a resizable array. When the array was not large enough to hold new elements, an array with double-sized would be created and old elements would be copied to the new array. These would make the old array became a garbage to be collected in the next garbage collection and the copy operation also would slow down the process.

From the benchmark results, the Order-Internally-Linked had the highest throughput among the implementations. Although the Order-Internally-Linked List had the similar structure of the Standard LinkedList, it had much lower object allocation rate than the Standard LinkedList. It was because the Order-Internally-Linked List did not allocate

extra objects, while the Standard LinkedList had to allocated extra objects, i.e. Node objects, to wrap each element to maintain the list.

According to the previous benchmark, for the Order Matching Engine alone running on Oracle HotSpot JVM using the Z Garbage Collector, the percentage improvement in throughput by changing the Order Matching Engine from using the ArrayDeque to the Order-Internally-Linked List was 72.6%.

But for the overall trading system combined by the Order Matching Engine and the FIX Gateway, the percentage improvement was only (130,236-112,817)/112,817=15.4% in the benchmark result, as the proportion of time spent in the FIX Gateway in the order placement function of the overall system was large.

7.6. Choice of synchronization techniques

From the benchmark result of the thread-safe techniques used in market data, lock techniques, i.e. ReentrantLock and ReentrantReadWriteLock, had higher throughput than the lock-free techniques, i.e. AtomicReference and volatile keyword. This contradicted the belief that lock-free techniques would have higher throughput than lock techniques. We believed that it was because the read and write contentions on market data were not heavy enough to get benefits from the lock-free techniques. In the benchmark program, we only set one market data reader thread to retrieve ten levels of market data in one millisecond interval, which was relatively infrequent when comparing with orders placing in microsecond-interval. The market data retrieval task was also simple and fast to finish, so that those locks did not lock the data long. As we followed the single writer principle [52], which was the concept of only using one thread to write data, in developing Order Matching Engine, the engine got the benefit of the biased locking property [53] of the ReentrantLock and the ReentrantReadWriteLock. Biased Locking allows the same thread to reacquire the lock quickly when the lock is uncontended.

In the benchmark results, the ReentrantReadWriteLock had lower throughput than the ReentrankLock. We believed that it was because the complexity of ReentrantReadWriteLock was higher and it did not suit our Single-Reader-Single-Writer scenario of our trading system. The ReentrantReadWriteLock would be more suitable for scenarios having Multiple-Reader-Single-Writer as its read lock could allow multiple threads reading at the same time when there was no thread writing. ReentrankLock involved simpler operations in acquiring mutex when comparing to ReentrantReadWriteLock [52] [53], thus using ReentrantLock led to the highest

throughput of the order placement function of the Order Matching Engine.

The AtomicReference approach utilized the Compare-And-Swap technique which was an expensive type of CPU instruction. The volatile approach used the keyword, volatile, to force the program to write data to the main memory, and not only to the CPU cache, and force to read data from the main memory, and not from the CPU cache. These made the two approaches slower in Oracle HotSpot Java Virtual Machine.

However, the Order Matching Engines using lock techniques had slightly lower throughput when running in the Eclipse OpenJ9 JVM. It might be due to the implementation of the Eclipse OpenJ9 JVM. This indicated that different JVM implementations might have different program behaviors.

8. Conclusion

When developing high-throughput and low-latency trading systems using Java, there are many areas a developer should concern. Software architecture simplicity, object pre-allocations, use of proper data structures, use of synchronization techniques, avoiding I/O operations and parallelizing tasks are some of the factors affecting the performance of a Java application. Implementing customized classes to replace the use of standard libraries also could improve the performance of a trading system significantly. Developers should seriously decide the choices of thread-safe classes and data containers according to the actual scenarios their applications running in.

In this dissertation project, we successfully developed a trading system from scratch without using any external libraries. It consisted of an Order Matching Engine and a FIX Gateway and it achieved high throughput and low latency. From our benchmark results, we found that our trading system running on Oracle HotSpot JVM with Z Garbage Collector could have the lowest time spent in garbage collection without sacrificing high throughput and low latency. With this setting, our trading system achieved 130,236 orders per second and 7.678 microseconds per order. Our Order Matching Engine individually achieved 491,344 orders per second and 2.035 microsecond per order. Our FIX protocol gateway achieved much lower latency than QuickFIX/J. Our gateway only spent 1.867 microseconds in generating each execution report, while QuickFIX/J spent 3.969 microseconds per each execution report.

There were several findings in our benchmark results.

1. Lower object allocation rates generally led to higher throughputs and lower latencies.

2. More suitable data structure led to higher throughputs and lower latencies.

3. Locks with reentrant property led to higher throughput than Atomic Reference and volatile keyword in our trading system and settings.

4. The choice of JVM and garbage collectors impacted throughput of operations, latency of operations, garbage collection count and garbage collection time.

For further investigations, we suggest to further decrease the latency of New Order Request parsing of the FIX Gateway and study how to add more servers or instances to increase scalability of the trading system.

9. References

[1] C. Kowalski, "Electronic Vs. Pit Trading," The Balance, [Online]. Available: https://www.thebalance.com/electronic-trading-versus-pit-trading-809246. [Accessed 27 1 2019].

[2] T. Mizuta, Y. Noritake, S. Hayakawa and K. Izumi, "Affecting Market Efficiency by Increasing Speed of Order Matching Systems on Financial Exchanges – Investigation using Agent Based Model," 2016.

[3] A. Studnitzer, "Electronic Trading: Is Speed Really What's Most Important?," CME Group, [Online]. Available: http://openmarkets.cmegroup.com/4478/electronic-trading-is-speed-really-whats-most-important. [Accessed 27 1 2019].

[4] K. Rungraung and P. Uthayopas, "A high performance computing for AOM stock trading order matching using GPU," in *2013 International Computer Science and Engineering Conference (ICSEC)*, Nakorn Pathom, 2013.

[5] H. Fu, C. He, H. Ruan, I. Greenspon, W. Luk, Y. Zheng, J. Liao, Q. Zhang and G. Yang, "Accelerating Financial Market Server through Hybrid List Design," *FPGA '17 Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 289-290, 2017.

[6] G. Tene, "Java Without the Jitter: Achieving Ultra-Low Latency," white paper, Azul Systems, 2013.

[7] Cinnober Financial Technology AB, "The benefits of using Java as a high-performance language for mission critical financial applications," white paper, Stockholm, Sweden, 2012.

[8] R. Martin, "Wall Street's Quest To Process Data At The Speed Of Light," InformationWeek magazine, 2007. [Online]. Available: https://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-light/d/d-id/1054287?. [Accessed 28 1 2019].

[9] Global eSolutions (HK) Limited, "GES Profile," [Online]. Available: http://www.ges.com.hk/en-us/ges-profile.html. [Accessed 27 1 2019].

[10] Forex-Central.net, "What is the A Book and B Book that forex brokers use?," [Online]. Available: http://www.forex-central.net/A-book-B-book.php. [Accessed 27 1 2019].

[11] "OpenJ9 - FAQ," Eclipse, [Online]. Available: https://www.eclipse.org/openj9/oj9_faq.html. [Accessed 5 3 2019].

[12] L. Zanivan, "New Open Source JVM optimized for Cloud and Microservices," Medium, 7 2 2018. [Online]. Available: https://medium.com/criciumadev/new-open-source-jvm-optimized-for-cloud-and-microservices-c75a41aa987d.

[13] D. Heidinga, "OpenJDK with Eclipse OpenJ9: No worries, just improvements," IBM, 10 1 2019. [Online]. Available: https://developer.ibm.com/blogs/openjdk-with-eclipse-openj9-no-worries-just-improvements/.

[14] Dan Heidinga;Sue Chaplain, "Eclipse OpenJ9; not just any Java Virtual Machine," IBM, 2018. [Online]. Available: https://www.eclipse.org/community/eclipse_newsletter/2018/april/openj9.php.

[15] S. Oaks, "Java Performance: The Definitive Guide," 1 ed., Sebastopol, California: O'Reilly Media, 2014, p. 74.

[16] D. Sharon and T. Niranjani, "Effective usage of customizable extensions on JAVA for optimized high performance computation," in *2014 IEEE National Conference on Emerging Trends In New & Renewable Energy Sources And Energy Management (NCET NRES EM)*, Chennai, 2014.

[17] Oracle, "HotSpot Virtual Machine Garbage Collection Tuning Guide," [Online]. Available: https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html#GUID-C7B19628-27BA-4945-9004-EC0F08C76003. [Accessed 27 1 2019].

[18] "Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide - Concurrent Mark Sweep (CMS) Collector," Oracle, 2018. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html.

[19] C. Hunt, M. Beckwit, P. Parhar and B. Rutisson, "Garbage First Overview," Pearson, 2016. [Online]. Available: http://www.informit.com/articles/article.aspx?p=2496621&seqNum=4. [Accessed 27 1 2019].

[20] H. Grgic, B. Mihaljević and A. Radovan, "Comparison of Garbage Collectors in Java Programming Language," IEEE, Opatija, Croatia, 2018.

[21] Oracle, "Garbage-First Garbage Collector," [Online]. Available: https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm. [Accessed 27 1 2019].

[22] I. Clark, "Main - ZGC - OpenJDK," Oracle, 14 1 2019. [Online]. Available: https://wiki.openjdk.java.net/display/zgc/Main. [Accessed 27 1 2019].

[23] P. Lidén and S. Karlsson, "The Z Garbage Collector," 2018. [Online]. Available: http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf. [Accessed 27 1 2019].

[24] E. Österlund, "The Z Garbage Collector - Scalable Low-Latency GC in JDK 11," Oracle, 16 11 2018. [Online]. Available: http://cr.openjdk.java.net/~pliden/slides/ZGC-Devoxx-2018.pdf. [Accessed 27 1 2019].

[25] K. Briggs, "Memory management in Eclipse OpenJ9," IBM, 11 1 2019. [Online]. Available: https://developer.ibm.com/articles/garbage-collection-tradeoffs-and-tuning-with-openj9/.

[26] "OpenJ9 -Xgcpolicy," Eclipse Foundation, [Online]. Available: https://www.eclipse.org/openj9/docs/xgcpolicy/. [Accessed 5 3 2019].

[27] "Metronome Garbage Collection policy (AIX, Linux only)," IBM, [Online]. Available:

https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_gc_mgc_intro.html. [Accessed 5 3 2019].

[28] F. Hanik, "The KISS principle," Apache, [Online]. Available: https://people.apache.org/~fhanik/kiss.html. [Accessed 27 1 2019].

[29] W. S. Dhruv Mohindra, "OBJ52-J. Write garbage-collection-friendly code," Carnegie Mellon University, [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/java/OBJ52-J.+Write+garbage-collection-friendly+code. [Accessed 27 1 2019].

[30] A. Lixandru, "Efficient Architectures for Low Latency and High Throughput Trading Systems on the JVM," in *Informatica Economica*, Romania, 2013.

[31] M. Thompson, D. Farley, M. Barker, P. Gee and A. Stewart, "High performance alternative to bounded queues for exchanging data between concurrent threads," technical paper, LMAX Exchange, 2011.

[32] K.-Y. Chen, J. M. Chang and T.-W. Hou, "Multithreading in Java: Performance and Scalability on Multicore Systems," in *IEEE Transactions on Computers*, 2011.

[33] N. Steingarten, "5 Tips for Reducing Your Java Garbage Collection Overhead," OverOps, [Online]. Available: https://blog.overops.com/5-tips-for-reducing-your-java-garbage-collection-overhead/. [Accessed 27 1 2019].

[34] S. Ritter, "How to Tune and Write Low-Latency Applications on the Java Virtual Machine," Oracle, 9 4 2011. [Online]. Available:

https://www.oracle.com/assets/virtual-machine-java-404173-ru.pdf. [Accessed 27 1 2019].

[35] Z. Zhuang, C. Tran, H. Ramachandra and B. Sridharan, "Eliminating OS-caused Large JVM Pauses for Latency-sensitive Java-based Cloud Platforms," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, San Francisco, CA, 2016.

[36] H. James, "Linux server performance: Is disk I/O slowing your application?," 2018. [Online]. Available: https://haydenjames.io/linux-server-performance-disk-io-slowing-application/. [Accessed 27 1 2019].

[37] A. Blewitt, "Interview with Martin Thompson on High Performance Java," InfoQ, [Online]. Available: https://www.infoq.com/interviews/thompson-high-performance-java. [Accessed 27 1 2019].

[38] A. Booth, "A fast java implementation of a limit order book," 2014. [Online]. Available: https://github.com/DrAshBooth/JavaLOB. [Accessed 27 1 2019].

[39] Oracle, "HashMap (Java SE 11 & JDK 11)," [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html. [Accessed 27 1 2019].

[40] L. Gupta, "How HashMap works in Java," HowToDoInJava, [Online]. Available: https://howtodoinjava.com/java/collections/hashmap/how-hashmap-works-in-java/. [Accessed 27 1 2019].

[41] Oracle, "TreeMap (Java SE 10 & JDK 10)," [Online]. Available:

https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html. [Accessed 27 1 2019].

[42] J. Morris, "Data Structures and Algorithms Course Notes," The University of Auckland, [Online]. Available: https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html. [Accessed 27 1 2019].

[43] University of Wisconsin-Madison, "Red-Black Trees," [Online]. Available: https://pages.cs.wisc.edu/~deppeler/cs400/readings/Red-Black-Trees/. [Accessed 27 1 2019].

[44] Y. Yu, "The Limit Order Book Information and the Order Submission Strategy: A Model Explanation," in *2006 International Conference on Service Systems and Service Management*, Troyes, 2006.

[45] C. He, H. Fu, W. Luk, W. Li and G. Yang, "Exploring the Potential of Reconfigurable Platforms for Order Book Update," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, Belgium, 2017.

[46] FIX Trading Community, "What is FIX?," [Online]. Available: https://www.fixtrading.org/what-is-fix/. [Accessed 27 1 2019].

[47] Oracle, "CopyOnWriteArrayList (Java SE 10 & JDK 10)," Oracle, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/CopyOnWriteA

rrayList.html. [Accessed 27 1 2019].

[48] R. Rose, "Coding for Ultra Low Latency," 23 5 2015. [Online]. Available: http://submicro.blogspot.com/2015/05/coding-for-ultra-low-latency.html.

[49] Onix Solutions, "FIX 4.4: Messages by MsgType – FIX Dictionary," [Online]. Available: https://www.onixs.biz/fix-dictionary/4.4/msgs_by_msg_type.html. [Accessed 27 1 2019].

[50] K. Beineke, S. Nothaas and M. Schöttner, "Efficient Messaging for Java Applications Running in Data Centers," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Washington, DC, 2018.

[51] Oracle, "OpenJDK: jmh," [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/. [Accessed 27 1 2019].

[52] Oracle, "jdk7u-jdk/ReentrantReadWriteLock.java at master · openjdk-mirror/jdk7u-jdk," [Online]. Available: https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/locks/ReentrantReadWriteLock.java. [Accessed 27 1 2019].

[53] Oracle, "jdk7u-jdk/ReentrantLock.java at master · openjdk-mirror/jdk7u-jdk," [Online]. Available: https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/locks/ReentrantLock.java. [Accessed 27 1 2019].

[54] Oracle, "ReadWriteLock (Java SE 10 & JDK 10)," Oracle, [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/locks/ReadWriteLock.html. [Accessed 27 1 2019].

[55] D. Inführ, "A first look into ZGC," 3 1 2018. [Online]. Available: https://dinfuehr.github.io/blog/a-first-look-into-zgc/. [Accessed 27 1 2019].

[56] Oracle, "JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)," [Online]. Available: https://openjdk.java.net/jeps/333. [Accessed 27 1 2019].

[57] A. Ocal, "liteExchange," [Online]. Available: https://github.com/sharma1981/liteExchange. [Accessed 27 1 2019].

[58] M. Thompson, "Biased Locking, OSR, and Benchmarking Fun," [Online]. Available: https://mechanical-sympathy.blogspot.com/2011/11/biased-locking-osr-and-benchmarking-fun.html. [Accessed 27 1 2019].

[59] M. Thompson, "Java Lock Implementations," 19 11 2011. [Online]. Available: https://mechanical-sympathy.blogspot.com/2011/11/java-lock-implementations.html.

[60] M. Thompson, "Single Writer Principle," 22 9 2011. [Online]. Available: https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html.