Efficient Architectures for Low Latency and High Throughput Trading Systems on the JVM

Alexandru LIXANDRU
Bucharest University of Economic Studies
alex.lixandru@gmail.com

The motivation for our research starts from the common belief that the Java platform is not suitable for implementing ultra-high performance applications. Java is one of the most widely used software development platform in the world, and it provides the means for rapid development of robust and complex applications that are easy to extend, ensuring short time-to-market of initial deliveries and throughout the lifetime of the system. The Java runtime environment, and especially the Java Virtual Machine, on top of which applications are executed, is the principal source of concerns in regards to its suitability in the electronic trading environment, mainly because of its implicit memory management. In this paper, we intend to identify some of the most common measures that can be taken, both at the Java runtime environment level and at the application architecture level, which can help Java applications achieve ultra-high performance. We also propose two efficient architectures for exchange trading systems that allow for ultra-low latencies and high throughput.

Keywords: *Trading Systems, Software Architectures, High Performance, Low Latency, High Throughput, Java Virtual Machine*

1 Introduction

The technological platforms used by securities exchanges have always been subject to increased exigencies coming, on one side, from the need to meet the demands of the trading firms, and, on another side, from the ever-changing financial environment. Trading platforms are at the core of exchanges and they have a direct impact on the competitiveness of the market place. For this reason, they need to be secure, scalable, and failure-tolerant and to perform well, in order to efficiently handle an ever-increasing transaction volume [1].

It is not only the volumes of transaction which are continuously growing, but also the demands for better access to markets and faster execution times. As algorithmic trading begins to be widely used by more and more trading firms, the exchanges' trading systems need to be continuously improved to keep up with the new requirements. Performance is probably the most dynamic coordinate of the electronic trading processing chain. Exchanges which were performing well a few years ago might not be able to handle the algorithmic trading and high frequency trading activity from nowadays.

Moreover, recent changes in the trading regulations has led to the emergence of new trading venues, in the form of multilateral trading facilities (MTF) or alternative trading systems (ATS). These new venues created added value for investors through reduced transaction costs and access to equity markets worldwide [6]. Alternative trading systems were notable for high trading speeds, making their platforms attractive to high frequency traders, for innovative fee structures and trading incentives and for enabling their customers to choose among customized market models.

These made the new venues highly attractive and caused an intensification of competition for order flow [7]. Regular exchanges had to find ways to keep their customer base, by offering discounts, improving their service offerings and by optimizing their trading platforms.

In this landscape, the characteristics of the software platforms offered by the trading venues have become one of the key decision factors for the trading firms when choosing a market place.

The pervasive impact of the trading systems on an exchange's service level and profit and

DOI: 10.12948/issn14531305/17.3.2013.06

loss statement has put a lot of pressure on software engineers and architects to optimize these platforms for ultra-high performance. Special attention was directed towards all of the aspects which have an impact on the system's reliability: availability, volume capacity, execution speed, scalability, fault tolerance and recoverability. In order to achieve a high level of performance, software engineers typically resorted to low-level technologies, highly specialized software solutions and even hardware-accelerated software components, deployed on high performance clusters of servers.
However, these overly complex architectures created challenges for developers and impacted the time-to-market periods. For this reason, more and more exchanges turned to using mainstream software technologies.
The motivation for our research starts from the common belief that the Java platform is not suitable for implementing and running ultra-high performance applications. Java is one of the most widely used software development platform in the world, and it provides the means for rapid development of robust and complex applications that are easy to extend, ensuring short time-to-market of initial deliveries and throughout the lifetime of the system [8]. The Java runtime environment, and especially the Java Virtual Machine, on top of which applications are executed, is the principal source of concerns in regards to its suitability in the electronic trading environment, mainly because of its implicit memory management [16].
In this paper, we intend to identify some of the most common measures that can be taken, both at the Java runtime environment level and at the application architecture level, which can help Java applications achieve ultra-high performance. We also propose two efficient architectures for exchange trading systems that allow for ultra-low latencies and high throughput.

2 The Electronic Trading Requirements

For a trading venue to be competitive and attract a high number of participants it has to keep up with the latest technological improvements and to offer fast execution times and fast dissemination of trade confirmations. Speed and response time are among the most conclusive characteristics which define the performance profile of a trading venue, and they are critical in making a clear distinction between regular trading platforms and high-performance ones.
One of the factors that have the biggest impact on the speed of a system is the latency. Latency is a measure of the delay experienced by the components of a system during their processing of a request. Latency exists at every stage of the trading execution chain [9]. The following diagram shows the layers where different types of latency can occur.

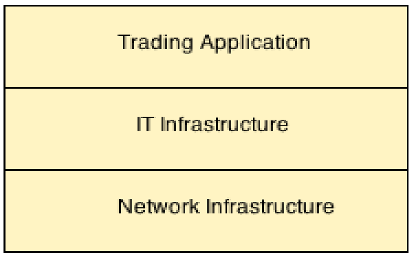

Fig. 1. Different layers of latency

Latency in the network layer is inherent, mainly because of the physical distance between the trading firms and the trading venue. Other factors which contribute to the network latency are the type of the communication media, the network architecture and the network protocols used. Choosing the most efficient network equipment and the appropriate communication protocols does help with reducing the network latency, but in most cases delays cannot be completely eliminated.
The IT infrastructure layer covers everything which is not related to network or the trading applications. This includes the hardware platform (the physical machine with all its devices), the operation system and all auxiliary software (virtual machines, messaging middleware, and databases). Along with the trading application layer, this is the area where latency can be reduced the most if proper solutions are employed. System architects and developers make every effort to identify the components which generate the highest de-

DOI: 10.12948/issn14531305/17.3.2013.06

lays, since optimizing those components will have the most impact on overall system latency.

Latency is also not constant and can be influenced by many factors external to the trading infrastructure [9]. Transaction volumes fluctuate based on regular events, such as market openings, and on outstanding events, such as news reports, announcements or corporate events. The execution of an algorithm and even the matching of two orders might trigger an explosion of messages routed towards the electronic trading system of an exchange. Spikes in message rates tend to impact the latency on systems which are not highly deterministic. This phenomenon is called jitter. Keeping jitter at a minimum is another challenge for a trading system, especially because these spikes appear at the moments which are the most visible for the trading participants and which have the biggest impact on their operations.

Another characteristic which is often used to assess the performance of a system is the number of requests it can process during a given time interval. This is represented by throughput, which is defined as the aggregated capacity of the exchange [11]. The ability of the system to execute large amounts of instructions gives the trading venues a competitive advantage by allowing the participants to get more order executions during the same unit of time.

The maximum throughput of a system usually has a direct influence on the latency. The following graph shows how the two measures vary in relation to each other.

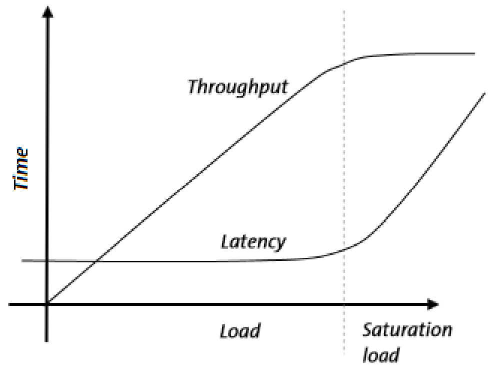

Fig. 2. The influence of maximum throughput on latency

As the number of messages received into the trading system continues to increase, throughput will increase in a linear fashion and the latency will stay relatively constant until the system reaches saturation load. From this moment throughput remains constant while latency begins to increase. If the load continues to increase, the system performance will degrade up to a point when participants will see the system as stalled. For this reason, many trading platforms have in place mechanisms that prevent the system from getting overloaded when the input message rates become too high. This protection is usually activated before the system reaches saturation and even if it limits the maximum throughput, it is critical for keeping the entire system stable. In practice, trading platforms increase throughput by distributing the processing logic on multiple machines, given that transactions on different instruments are independent. Nevertheless, designing the trading system so that it achieves the highest possible throughputs is still a priority for most trading venues, whether they employ distributed computing or not.

Achieving high performance and keeping it stable throughout the entire trading day are a necessary but not a sufficient condition. For many trading firm, there are other system requirements which weigh as much as the performance criteria when a decision is to be made whether the firm will activate on that trading venue or not. Scalability and functional expandability are two of them. Scalability allows the trading system to grow as markets grow, while expandability refers to the ability to add and integrate new functional components within the system. These two factors are particularly important in the emerging markets where new trading products and instruments are being constantly added.

Fast recoverability is also a key factor in becoming a competitive trading venue. In case of a system failure, the trading activity needs to be resumed as fast as possible, without any data loss. A trading system should be capable of starting up in a matter of seconds and should recreate the entire state from before the failure so that it can restart its operations. This leads to other requirements in regards to data persistence, data replication and hot-standby and data consistency. Although not all of these are enforced for all trading processes or markets, they are adopted by most exchanges.

Other key requirements for high-performance trading systems are deterministic behavior, which allows for predictability of performance, visibility which allows for identification of performance issues and other threats and fast time-to-market, which allows the trading system with keep up with the ever-evolving market places [10].

Traditionally, trading platforms have been deployed on high performance hardware, sometimes using specialist processing components, like Field Processor Gate Arrays (FPGAs), Graphics Processing Units (GPUs) and similar technology. These usually had high costs and were creating challenges for developers and architects. However, thanks to recent advances in the processing technology, great performances can now be achieved on commodity hardware, allowing trading venues to move away from highly specialized hardware. This allowed trading systems to achieve better visibility and faster time-to-market, but also favored the adoption and use of mainstream software technologies, like the Java platform.

3 Java Technology and Its Limitations

Java technology is one of the most widely used solutions for enterprise-level applications in the world. It features excellent developer tools as well as key characteristics, like portability, resilience and scalability, which make it a good platform candidate for almost any type of application. Many high performance applications have been written in Java to take advantage of its ecosystem. However, for solutions that require ultra-low latency, Java technology has some inherent limitations.

The nature of the Java platform itself and its execution model impose concerns in choosing it for ultra-high performance systems. Java applications are executed on top of a runtime environment, by a component called the Java Virtual Machine (JVM). This approach allows for portability, since the application source code is not compiled to machine code specific to a certain processor architecture, but to an intermediary portable format called *byte-code* [2]. The byte code is then executed by virtual machines specifically created for different hardware and software platforms. Most JVM implementations have two modes for running the byte code: by interpreting it and by compiling it to native machine code on the fly, a process called *just-in-time* compilation (JIT). When being interpreted, the byte code runs slower than compiled machine code, however, the JIT compilation offers far better performance. In addition, in certain cases JIT code can perform even faster than native code [4]. This is possible because the JVM can make global optimizations at runtime, like in-lining of library methods and rearranging execution instructions, which is not possible with statically compiled code. Also, the compilation can be optimized for the targeted CPU and the operating system where the application runs. By performing the compilation at runtime, the system is able to collect statistics about how the program is actually running and based on these statistics it can perform the appropriate optimizations.

However, there is a limited amount of code which can be JIT compiled. The virtual machine keeps all the compiled code in memory, in a code cache. When the cache fills the compilation has to stop and no other portions of the byte code can be further optimized. For this reason, the JVM implementations have in place specific rules about what code is translated to native code and how the compiled code is marked obsolete and removed from the code cache. A common approach, also used in the reference implementation of the Java Virtual Machine, the Oracle HotSpot JVM, is to monitor which sequences of the byte code are frequently executed (the hot spots) and translate only those sequences to machine code for direct execution on the hardware. This approach is based on the observation that programs spend most of the time executing a minority of their code, so it is not efficient to compile portions of the code which are not going to be executed too many times. Instead, only hot sequences will be translated to machine code, thus reducing the time needed for compilation.

Another limitation of the Java platform which prevents Java applications from easily achieving ultra-low latencies is the Garbage Collector. Java is a managed environment in which objects cannot be explicitly deallocated. It is the runtime environment which performs all memory reclamation when objects are no longer in use. The component responsible with all memory management is called Garbage Collector. All JVM implementations have a garbage collector which deletes objects as they become unused. Since there is no specific collection mechanism imposed by the Java platform, a large variety of algorithms have been developed, including reference counting, mark-sweep, mark-compact, copying, and non-copying implicit collection. Most of these techniques halt the processing of application logic when garbage collection is needed, generating pauses in the program execution. These pauses highly depend on a number of factors, like the total amount of memory allocated by the program, the number of objects that need to be reclaimed and the available free memory [3]. In practice, for applications requiring large amounts of memory, the pauses generated by the garbage collector can take from a few milliseconds to a few seconds, which is unacceptable for high performance systems.

In order to avoid these interruptions, new garbage collections techniques have been developed that allow the collection process to be interleaved with the main program execution. Other techniques, known as generational collection, attempt to improve efficiency and memory locality by working on smaller areas of memory called *generations*. With this technique, most of the new objects are created in a memory zone called young generation. This is the area where short-lived objects will reside and on which the garbage collector is usually more active, given the observation that recently allocated objects are likely to become garbage within a short period of time. When the young generation fills, the garbage collector performs a minor collection, in which only this memory zone is cleaned. Minor collections can be optimized assuming that most objects in the young generation are unused and can be deleted. As objects survive minor collection cycles they are promoted to another memory zone called old generation. When this zone becomes full, the garbage collector performs a major collection, in which the entire memory is cleaned. Major collections usually last much longer than minor collections because a significantly larger number of objects are involved.

The garbage collection process imposes a penalty on the application performance. Consequently, it is important that the garbage collector is efficient and interferes with program execution as little as possible.

4 Tuning Java Applications for Ultra-High Performance

There are multiple steps to perform in order for a Java application to be able to achieve ultra-low latency performance. These include, but are not limited to: network and hardware configuration, operating system and auxiliary software configuration, proper design of application architecture and preparation of the runtime environment. This paper does not cover hardware or OS configuration, and concentrates on the optimizations taken from a software architecture and development perspective.

Given the nature of the Java platform, developers must pay special attention to both the application itself, and to its runtime environment, specifically the Java Virtual Machine. Incorrect configuration of the JVM can cause the application to perform badly or may prevent the runtime platform from taking advantage of the performance optimizations available in modern JVM implementations.

The most common configurations of the Java Virtual Machine are those related to the maximum amount of heap space the application

DOI: 10.12948/issn14531305/17.3.2013.06

can use, and to the Garbage Collector activity. The two are tightly interconnected because the total available memory is the most important factor affecting the performance of the garbage collection process.

There are two primary measures of the garbage collector impact on the running application: *execution time*, which is the total time not spent in garbage collection, and *pauses*, which are the times when application execution is interrupted for the collection to occur. Other common measures are *footprint*, which is the total amount of memory used by the application process, and *promptness*, which refers to the amount of time between the moment when the object becomes unused and the one when the memory is freed [5].

In general, most configurations try to accommodate all of these four aspects. However, in a high-performance application, like a trading system, execution time and pauses are more critical measures than the others. Footprint will likely not make a difference, since trading systems are generally memory-intensive and have large working sets, while promptness has little overall impact.

A common solution to reduce the impact of the garbage collector is to design the system so that garbage is collected only one time a day, during off business hours or during a daily maintenance window. This can be achieved by giving the application a large heap, enough to hold all of the objects allocated during the program execution, throughout the day. By having a heap which never fills there will be no need for garbage collection. This is the desirable approach, since it completely eliminates the overhead caused by garbage collection and also the need to tune the Garbage Collector configuration. However, the application itself must be developed so that it uses mostly long-lived objects, and that it creates as few short-lived objects as possible, in order to prevent even the young generation of the heap from filling. It also requires the data structures to be simple, and its objects to have a simple life cycle which allow object reuse.

Although not impossible, this is difficult to implement, so a trade-off must be found. Depending on execution time that needs to be achieved, the young generation area of the heap may be sized so that it permits a small number of minor collections, while the old generation is sized so that it does not get filled. Since minor collections perform relatively fast, very short GC pauses can be achieved. To reduce the pauses even more, alternative collection techniques can be used, like concurrent collection. The concurrent collector is designed specifically for applications that prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage collector while the application is running [5].

Choosing a correct size for the entire heap or for each of its memory areas is not a straightforward process, because the amount of available memory affects the performance of the garbage collector. A decision about the memory size must be taken only after extensive performance testing and monitoring of the GC activity. The general recommendations made by JVM implementers usually do not apply to ultra-low latency applications, therefore measurements are the only valuable information upon which a decision may be made.

Other configurations that affect the performance of the application are those related to the JIT compiler. As mentioned above, the JVM can compile portions of byte code to native machine code and execute them directly on hardware, dramatically improving execution time. Since the compiled machine code is kept in a memory cache, it is possible that this cache get filled, preventing further JIT compilation. This usually happens when a complex application performs repetitive operations during startup, before commencing the actual execution. In such a case, the code cache needs to be sized to accommodate the compilation of the actual business logic of the application, or the JIT compiler needs to be given a hint on when byte code should be translated to native code, so that it bypasses the startup processing.

Turning a Java application into an ultra-low application is not easy. This often involves using low-level solutions, optimization techniques targeted to specific platforms or even native solutions implemented directly at the operating system level. All these increase the complexity of the application code and reduce its portability and maintainability for the benefit of having latencies of less than a millisecond. While not all trading venues require this level of performance, there are systems which need to take advantage of all possible solutions.

Some of the implementation techniques and architectural optimizations commonly used to get the best performance for a Java application are detailed in the following sections.

- *Simplicity*
  Simplicity and understandability are key factors in designing an ultra-high performance system. They allow developers and software architects to easily identify performance bottlenecks and to reason about what parts for the flow need to be optimized. Overly complex architectures will make it hard to test and improve the system and will impact the performance. This is why the trading application logic must be kept simple and must not contain any functionality that can be performed by components or processes.
- *Use of memory*
  Garbage collection can become problematic when developing ultra-low latency systems in a managed environment like Java. The more objects are allocated, the more work the Garbage Collector has to do. In order to keep the collection cycles at a minimum, a common practice is for the applications to create as few garbage objects as possible. This can be achieved through reutilization of objects. It is, therefore, a good practice for applications to pre-allocate a set of objects which are reused throughout the execution of the program. These objects will exist as long as the application runs so they will not impact the garbage collection process. The pre-allocation is also beneficial for caching and improving memory access, as data is likely to be laid out contiguously in main memory.
- *Use of parallelism*
  Traditionally, trading systems have been single-threaded, due to the requirement to process all requests in the order they are received. However, as the number of order requests continued to grow, the single-thread design started to show its limitations. In order to keep up with high capacity requirements, systems had to start processing data in parallel. A common approach, which is still used nowadays, is to keep the main processing logic single-threaded, but to perform the preparations and other auxiliary tasks in parallel. This allows for a better utilization of the existing processing power, which leads to higher throughput and, in some cases, to overall better latencies.
  It is advisable to simplify the execution model and to parallelize as much work as possible, keeping on the main execution thread only those tasks which require strict sequential processing, like matching orders. The rest could all be done in parallel: distributing trade confirmations, pushing market data to feed handlers, or even aggregating data for reporting services. It is not uncommon in trading environments to have data *eventual consistent*, meaning that at the peaks of activity, reports might contain data which is not consistent with what exists in the trading system's memory. Some of the trading systems even save the exchange state asynchronously on storage devices in their attempt to achieve ultra-low latency.
- *Use of lock-free techniques*
  Although parallel processing brings many benefits, it also introduces complexity and limits the ability of the developers and software architects to optimize the application. Moreover, the use of traditional solutions to

DOI: 10.12948/issn14531305/17.3.2013.06

solve multi-threading issues - locks, for synchronizing concurrent access to data, and queues, for communication between components - is not always suitable in the trading environment, where large volumes of data are coming in very short periods of time. Large volumes and very short timeframes means a lot of locks will be performed within short periods of time and data contention will be frequent.

The problem with locks is that they require arbitration when contended [15]. The arbitration is achieved by a context switch to the operating system kernel, to allow the kernel to suspend other threads waiting on the lock. During a context switch the control of execution is transferred to the operating system, which might choose to perform additional tasks. Moreover, the execution context can lose previously cached data and instructions. Since it is very likely to have data contention in a high performance application, using locks will inherently lead to jitter and high latencies.

For this reason, a high performance system should use lock-free solutions and alternative concurrent data structures. Lockless architectures have the great benefit of being able to avoid data contention which dramatically improves performance. Alternative data structures do not use locks for synchronizing access to resources shared by multiple threads. Instead other mechanisms are used, like *compare-and-swap (CAS)* operations, and busy spins, which prevent kernel arbitration. They offer steady performance up to the point of load saturation, and allow the system to scale when necessary.

- *Use of low-level techniques*

  In some edge cases, when a trading system needs ultra-high performance, beyond what the Java Virtual Machine can offer, it can resort to the use of low-level solutions which are at the boundary between the JVM and the operating system.

  Thread affinity refers to binding a thread to a physical CPU or core, so that the execution of the thread will be performed only by of the designated CPU or core [13]. This has the advantage that the cache of the CPU will not be polluted with data from other threads, thus reducing the need of the CPU to fetch data from main memory. Setting the thread affinity is not possible with the standard Java API, and it must be performed through native operating system calls.

  Optimizing code for CPU cache is another low-level technique, which is based on the observation that data in the CPU cache is not stored as objects but rather as blocks of data with a specific size, called *cache lines*. This means that more than one object can be cached at a time, which might be beneficial in some case, for example when using arrays, or might create issues in other cases (the *false sharing* phenomenon). False sharing occurs when two threads own two different objects which reside in the CPU cache and one thread modifies its object, forcing the other thread to reload the whole cache, even if the object it was trying to access was up to date [14]. Moreover, if the two threads try to write to their own object at the same time, the CPU will consider it as a write to the same variable - therefore a contended write. In order to minimize these situations, a high performance application should make sure that concurrent written variables do not get fetched in the same cache line, for example by utilizing variable padding techniques.

5 Efficient Trading Platform Architectures

The specifics of the trading industry and specifically the requirements in regards to the flow of the order execution process impose the existence of certain technical components within the trading system. Each of these components is responsible for a specific processing stage from the order request lifecycle.

Typical components of a trading system are presented in the diagram below.

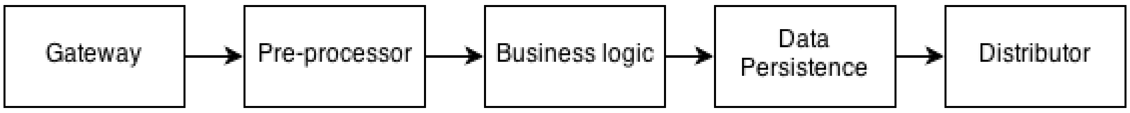

Fig. 3. Core technical components of an exchange trading system

The role of each of these components within the processing workflow is detailed below.

- The *gateway* is the entry point in the trading system. It might be represented by a FIX session handler, a native connection listener, or an API handler accepting external connections. The gateway usually performs some light validations on the incoming messages, like format or syntax verifications and simply forwards the requests on to the next component. Gateways are also responsible with delivering messages generated by the trading system to the external connections they are managing.
- The *pre-processor* component might consist of one or more sub-components responsible with different tasks, according to the specifics of the trading system. Typical such sub-components are:
  - Journaling module, which persists the input message to a durable event log, allowing the application to recreate its state by replaying the entries from the log;
  - Replicator, which sends the input messages to a replica of the trading system, usually residing on another physical machine;
  - Un-marshaling module, which converts the input messages to objects that are used by the trading application logic;
  - Validator, which verifies the integrity of the received messages, as well as the access level and permissions of the trading participant who initiated the message.
- The *business logic processor* is the central component of a trading system responsible with the matching of orders and with other tasks critical to the proper functioning of the market place, like scheduling and handling market events, opening and closing of trading sessions, monitoring price variations and handling administrative tasks. Most of the business logic processors work with data from memory, in order to perform very fast and they usually interact only with internal components.
- The *data persistence* component is responsible with storing the state of the exchange to a database for reporting purposes. It is different from a journaling module, which only stores the input messages to a journal or event log, without any information about the effects of those messages. Instead, the persistence component will save the data as it exists after a message has been processed. For most of the systems, this will be performed synchronously, offering a consistent view of the data, as it exists in the trading application memory. However, for some systems that require ultra-high performance, the persistence may be done asynchronously, in order to avoid the overhead incurred by the relatively slow performance of the storage devices. In this case, the reporting database will be *eventual consistent* with the actual trading system state.

- The *distributor* component delivers the results of the trading activity to other components of trading venue infrastructure, to post-trading systems and to external clients. Consequently, there might be multiple distributors in a trading system: output routers, which send messages back to gateways, for delivery to FIX sessions or native connections, and data handlers, which delivers messages to trade management modules, market surveillance and monitoring applications, data feed consumers and other external systems.

From a functional point of view, order requests must be handled by each component sequentially, even if from the execution point of view, systems might parallelize some of the tasks. It is also not a requirement for trading systems to have separate functional modules corresponding to each of these components. Most traditional trading system architectures would combine more components into a single monolith, all-purpose processing module, as presented in Figure 4.

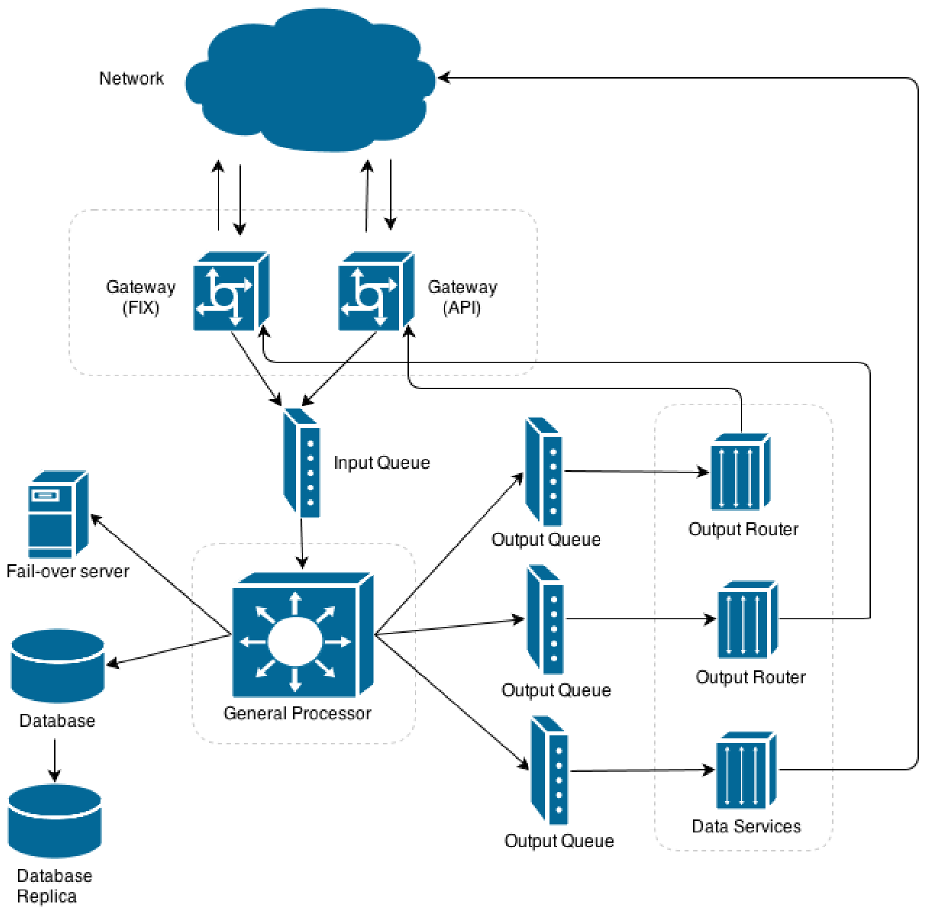

Fig. 4. The high-level architecture of a traditional trading system

As seen in the diagram, a traditional trading system usually contains two gateways, one for FIX connections and one for native connections, a general processor and multiple parallel distributor components. The general processor is usually single-threaded, consuming all messages sequentially and being responsible with all the processing, from unmarshalling and validation, to order matching and persistence. The communication between the components is done through queues.

This architecture is still common among trading venues which don't have strict requirement for latency or throughput. The use of blocking queues and the single-threaded processing limits the ability of the system to scale up or to optimize its performance.

An improved system architecture, which offers low latencies and very high throughput, is presented in Figure 5.

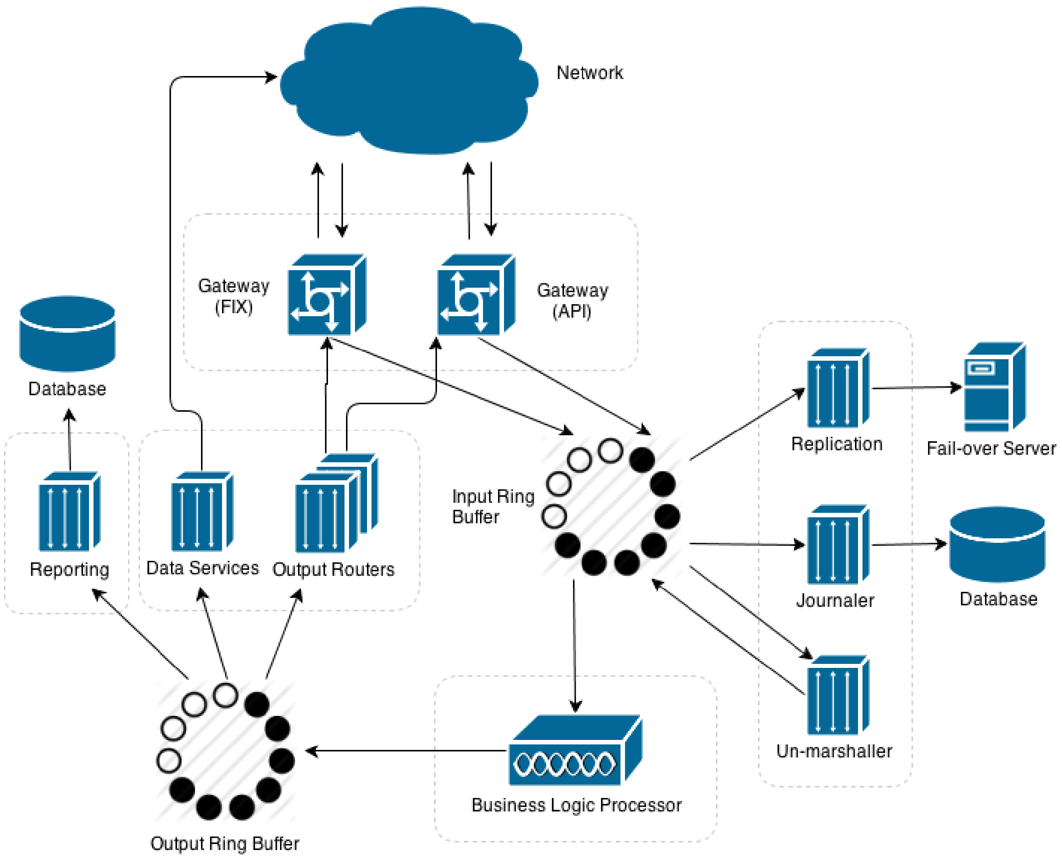

Fig. 5. The high-level architecture of a trading system based on circular buffers

This system tries to parallelize as many tasks as possible, so most of the components will run in separate threads. The communication between components is done through two circular data structures, called *ring buffers*, one for input messages (the input ring buffer) and one for execution results (the output ring buffer). Ring buffers are essentially linear data structures, usually arrays, which overwrite the oldest entries of the data structure as it fills up, hence the circular nature. Ring buffers have in place well-defined mechanisms for components to put data into the ring structure and to read it. There is also a protection which does not allow the ring to overlap and overwrite entries which have not been processed by all consumers. For this reason, the size of the ring buffers is usually very large, in order to accommodate the situation when consumers are slow and cannot keep up with the rates at which messages are put into the ring structure.
The processing stages might differ from one implementation to another, but a typical flow is presented below:

1. Data is fed into the system through the two gateways and put into the input ring buffer
2. Different components of the system, like the replicator, the journaling or the un-marshalling modules, which run in parallel, take data from the ring buffer and process it in the same order as it arrived in the system. Since these components are independent from each other, their processing can be performed concurrently without any conflicts. The un-marshalling component is the only one which needs to put data back into the ring buffer after it finishes translating it into an internal representation format.
3. The business logic processor waits until the un-marshalling module puts the transformed data into the circular buffer. Since it needs the data to be represented as internal objects, rather than a stream of characters, it cannot continue until at least one message has been un-

marshalled. This is the only place where a component needs to wait for another one, but the coordination is done without the use of locks. Once the business logic processor finishes its tasks, it puts the results into the output ring buffer.

4. Different output components running in parallel read results from the output ring buffer and deliver them to their destinations. The output routers send results to the gateways, data services component sends data to external systems and feed handlers while the reporting component stores data for reporting purposes.

With ring buffers, each component processes all messages in the same order they were received in the system. By coordinating the activity of some components, like the un-marshalling module and the business logic processor, the system can guarantee that all messages are handled in sequential order.

This architecture also has the advantage that the behavior of the system can be easily adjusted by defining new dependencies between different modules. For example, the un-marshalling process could be configured to wait for the journaling module to finish saving the messages to the event log; or the output routers could start delivering data only after the reporting component finished saving it to the database.

High performance is achieved mainly because most of the processing tasks are executed in parallel. Also, by avoiding the use of locks and queues, stable performance can be achieved even at bursts of activity.

Another efficient architecture, offering ultra-low latencies and high throughputs is depicted in Figure 6.

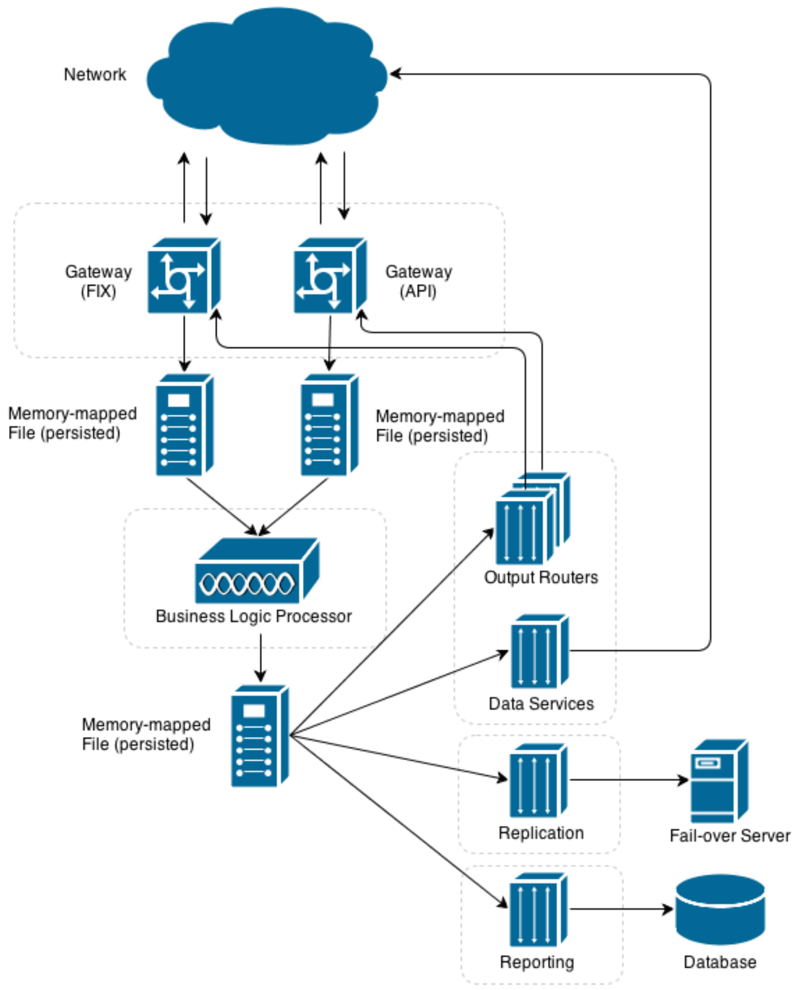

Fig. 6. The high-level architecture of a trading system based on memory-mapped files

This architecture is based on a different concept: *memory-mapped files*. A memory-mapped file is a segment of virtual memory which is directly correlated with some portion of a physical file present on the disk. This byte-for-byte correlation between the file and the memory space permits the application to treat the mapped portion as if it were primary memory, while the operating system transparently deals with loading the requested data and writing it into the underlying file. The main benefit of this approach is enhanced performance; accessing memory-mapped files is faster than using direct read and write operations on a file [12]. This is mainly because writing to a memory-mapped file only consists in changing the program's local memory, while regular read and write operations on files involve expensive system calls. Moreover, in most operating systems, the memory region is mapped in kernel space, so no copies of data need to be created in user space [17].

Another advantage of memory-mapped files is the ability to safely share them between multiple threads or even multiple processes. Two or more threads or applications can simultaneously map a single physical file into memory and access this memory concurrently. This is how the components in the above diagram communicate with each other. For example, each of the two gateways writes all input messages to a memory-mapped file. The Business Logic Processor maps both of these files in memory and waits for messages to arrive. Once a message is written in the local memory of any of the gateways, and implicitly stored in the mapped file, the memory region of business logic processor correlated with that file will be updated as well. Thus, the business logic processor component simply needs to read the new data from memory and start processing it. Coordination is still required, but it is performed through lock-free mechanisms to avoid high latencies.

There is, however, a disadvantage in working with memory-mapped files: for large files, only a portion of the file is loaded in memory and if the application requests a block of data which is not present in memory, a page fault will be raised and, as a result, the portion of the file containing the requested data will be brought to memory. Since page faults are expensive, they may degrade the performance of the entire application if they occur at high rates. It is, however, possible to prevent page faults from happening, if the consumer thread, reading messages from the memory-mapped file is at least as fast as the producer. By processing messages which have been recently written to the mapped file, it is very likely that the new data is still loaded in memory, so when the consumer requests it, no page fault will occur.

The typical process flow of a trading system based on memory-mapped files, as seen in the above diagram is presented below:

1. Requests received by the gateways are written to memory-mapped files. These files also serve as event logs which can be used for replay in case the system needs to restart.
2. The Business Logic Processor maps both of the input files in memory and reads new messages as they arrive. In such architecture, this component is usually very efficient and very lightweight, in order to keep up with the gateways. In practice, the gateways will always be slower due to their interaction with the network, so the business logic processor will be able to read the input messages while they are still mapped in memory, keeping the page faults to a minimum. The results of the processing are written to another memory-mapped file.
3. Different components running in parallel will map the output file in memory and process the results written by the Business Logic Processor.

As seen above, no explicit persisting of the data is required throughout the process flow, except for storing data for reporting purposes. The operating system will automatically update the underlying memory-mapped files on the disk once the mapped memory region is changed throughout the execution of the program. This yields ultra-low latencies, since all I/O operations are performed at operating system level. Moreover, by processing most of the tasks in parallel, and by having fast consumers that help reduce the number of page faults, the system is able to achieve ultra-high performance.

6 Conclusions and Further Research

Java technology has been successfully used by many large-scale high performance applications throughout many industries. However, software developers have faced some intrinsic limitations of the Java platform which prevent applications from meeting the rigors imposed by the financial market place. Chief among these limitations are the pauses caused by the garbage collector and the slowness of program execution. Achieving ultra-low latencies and high throughput using Java is not unfeasible and, with proper configuration of the runtime environment, and a careful application design, it is possible to overcome these limitations. This paper has presented a summary of the most important measures that can be taken on the Java Virtual Machine configuration and the Java application itself in order to optimize for ultra-high performance. In the last part of the paper, two efficient architectures have been presented. They make use of parallelism and implement alternative data structures for passing data between components. Initial tests have shown that these architectures perform better than the traditional systems under all conditions tested. Our ongoing research aims to explore the performance characteristics of the proposed system architectures and to identify key points for further increasing their performance.

References

[1] A. R. Schwartz, R. Francioni, *Equity Market in Action (The Fundamentals of Liquidity, Market Structure & Trading)*, John Wiley & Sons, Inc., 2004

[2] I. H. Kazi, H. H. Chen, B. Stanley, D.J. Lilja, “Techniques for Obtaining High Performance in Java Programs”, *ACM Computing Surveys*, 2000

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley, “Myths and Realities: The Performance Impact of Garbage Collection”. *Proceedings of the ACM Conference on Measurement & Modeling Computer Systems*, pages 25–36, ACM, New York, 2004.

[4] Oracle, “The Java HotSpot Performance Engine Architecture”, white paper, http://www.oracle.com/technetwork/java/whitepaper-135217.html

[5] Oracle, “Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning”, technical paper, http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

[6] M. Mühlberger, “Alternative Trading Systems: A Catalyst of Change in Securities Trading”, *Deutsche Bank Research*, 2005

[7] L. Harris, *Trading and Exchanges*, Oxford University Press, Oxford, 2003

[8] Cinnober Financial Technology AB, “The Benefits of Using Java as a High-Performance Language for Mission Critical Financial Applications”, white paper, 2012

[9] Sun Microsystems, Inc., “Building a Low Latency Infrastructure for Electronic Trading”, white paper, 2009

[10] CDW LLC, “High-Performance Computing: Capital Markets”, white paper, 2012

[11] Cinnober Financial Technology AB, “Latency”, white paper, 2009

[12] P. Lawrey, “When Using Direct Memory Can Be Faster”, technical article, 2012, http://vanillajava.blogspot.ro/2012/11/when-using-direct-memory-can-be-faster.html

[13] C. Terboven, D. an Mey, D. Schmidl, H. Jin, T. Reichstein, "Data and Thread Affinity in OpenMP Programs", *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, pages 377-384, ACM, New York, 2008

[14] T.E. Jeremiassen, S.J. Eggers, “Reducing False Sharing On Shared Memory Multiprocessors Through Compile Time Data Transformations”, ACM, Vol. 30, No. 8, 1995

[15] M. Thompson, D. Farley, M. Barker, P. Gee, A. Stewart, “High Performance Alternative To Bounded Queues For Ex-

changing Data Between Concurrent Threads", technical paper, LMAX Exchange, 2011

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

[17] G. Back, W. Hsieh, "Drawing the Red Line in Java", *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 116-121, IEEE, 1999

Ionuţ-Alexandru LIXANDRU graduated from the Bucharest Academy of Economic Studies in 2008. He is a Ph.D. candidate in the field of Economic Informatics at the Bucharest Academy of Economic Studies. Alexandru is currently working at the Bucharest Stock Exchange, as a Software Developer within the Trading System Development department. Previously he has been for 5 years with TechTeam Global within the Global Business Applications department. His main areas of interest are system integrations, web technologies, and low-latency trading systems.

DOI: 10.12948/issn14531305/17.3.2013.06