Low-Latency Trading in a Cloud Environment

Andrew Addison, Charles Andrews, Newas Azad, Daniel Bardsley, John Bauman, Jeffrey Diaz,

Tatiana Didik, Komoliddin Fazliddin, Maria Gromova, Ari Krish, Ryan Prins, Larry Ryan, Nicole Villette

andrew.addison@bjss.com, charles.andrews@bjss.com, newas.azad@bjss.com, daniel.bardsley@bjss.com, john.bauman@bjss.com, jeffrey.diaz@bjss.com, tatiana.didik@bjss.com, komoliddin.fazliddin@bjss.com, maria.gromova@bjss.com, ari.krish@bjss.com, ryan.prins@bjss.com, larry.ryan@bjss.com, nicole.villette@bjss.com

BJSS Inc., 14 Wall Street, Suite 2069, New York, NY 10005, USA

Investors seek competitive advantage through order creation, based on timely market information, and submission into an ultra-low latency matching engine. Historically, these environments were built with proprietary technology deployed on dedicated infrastructure to minimize latency and control jitter. However, as outlined in this paper, we believe that innovations in cloud infrastructure allow configuration and implementation of an acceptably low-latency trading platform in a cloud environment. We have demonstrated this by implementing a simple foreign exchange (FX) trading system and deployed it to a cloud environment, recording network latency and overall system latency. Here, we assess the capability of public cloud infrastructure to perform low-latency trade execution under various configurations and scenarios. Additionally, we provide analysis of clock skew between systems in a cloud environment to assess whether the deployment can satisfy stringent clock precision required by MiFID2 regulations. We conclude that sufficiently low latency and controlled jitter can be achieved in a public cloud environment to support security trading in the public cloud; however, currently available clock synchronization methods are not yet sufficient to meet regulatory requirements. This paper benefits public cloud vendors desiring to support low-latency trading, as well as, institutions seeking to take advantage of cloud platforms to build distributed, resilient, scalable trading systems without sacrificing the competitive advantage offered by low-latency trading. The results of this study can be extended into use cases requiring controlled time critical latencies, such as select Internet of Things applications, messaging platforms or massively multi-player online worlds.

Keywords: low latency, high performance computing, cloud, amazon web services, Microsoft Azure, Oracle Cloud Infrastructure

I. Introduction

High frequency/low latency trading systems [1, 2] symbolize today's intensive focus on eliminating technical barriers to ever faster communication, processing and decision making. To stay ahead in the high-speed race, trading systems need to be designed to maximize throughput, minimize latency, and accommodate rapid development of additional functionality. It is unlikely that modern electronic markets' relentless drive towards faster decision making will abate in the near future [3, 4]. Traditionally, to achieve low latency, high-frequency trading has required powerful server hardware in a data center, scaled to accommodate worst-case network traffic scenarios on the busiest trading days. These trading systems must be resilient in the face of network or power failures, requiring expensive redundant hardware as well as offsite data retention. Cloud-based software solutions are architected to be distributed, resilient, and easily scalable, meeting many of the needs of a high-frequency trading system. However, industry trends toward deploying applications into cloud environments, while effective and beneficial for meeting capacity, availability, and resiliency requirements, introduce additional challenges to achieving low latency. The greatest barrier to building a high-frequency trading system in a cloud environment has been the limited ability to provision hardware which is co-located within a data center, essential for meeting the low latency requirements of such systems. Other challenges include decreased performance due to virtualization of cloud instances and the difficulty of synchronizing clocks across instances to meet regulatory requirements. That said, a system running in the cloud with reduced latency is an ideal objective for a financial institution to keep up with competition and ensure that its users have the best possible chance to make informed decisions while having real-time data available.

Recent developments by cloud providers such as Azure and Oracle Cloud Instructure (OCI), including the introduction of availability sets, access to bare-metal instances, and high performance networking are enabling support for a high-frequency trading use case. Such developments increase the authors belief that it is now possible to implement a high-frequency trading system fully hosted in a public cloud environment. Such a system would provide traders with all the advantages of a fast, efficient trading platform, as well as the robust, easily maintainable infrastructure provided by a distributed cloud architecture.

The authors note that traditional low-latency trading systems built on dedicated servers with co-located trading platforms provides the lowest latency. However, they don't offer the advantages of public clouds, including cost-effective dynamic scaling.

In this paper, we examine a simple trading system deployed to a cloud environment and tuned for low latency, to establish the feasibility and performance of a low-latency trading system hosted on cloud-based servers. We executed performance tests on an automated deployment hosted on Amazon Web Services (AWS), Microsoft Azure and Oracle Cloud Infrastructure (OCI). This paper will demonstrate the effectiveness of our system to deliver round trip trades on cloud-based servers at production message rates with competitive latency. Additionally, we perform a measurement and analysis of clock skew between two servers when running performance tests, in order to determine whether it is currently possible to implement a MiFID II [5] compliant trading system in a cloud environment.

Our research aims to prove that we can provide a low-latency trading platform hosted on cloud-based servers which can achieve performance goals with an average latency of 500µs (microsecond) and 99% less than a ms (millisecond) at a transaction rate of 10,000 orders/second/single currency pair. We achieved an average latency of 240µs with 99% of latencies less then 345µs. These metrics were selected for analysis because markets are measured by speed and fairness of order handling. Speed in terms of quick order request response time, i.e. low latency, and fairness in terms of deterministic response time, i.e. minimal jitter.

II. BACKGROUND

Latency is defined as being any delay or lapse of time between a request and a response. Jitter is defined as variation in the latency. As it pertains to trading, latency directly influences the amount of time it takes for a trader to interact with the market. The timely and deterministic reception of pertinent market information and the ability to act upon its receipt are often greatly impacted by latency issues [6]. Obtaining low latency and jitter is critical as data must travel round trip to make a single transaction. Investors are constantly looking for ways to improve their chances of success. Hence, it is important for traders to have access to the lowest latency and jitter possible on trading platforms. Every microsecond counts when making trading decisions and each microsecond lost to latency can result in lost opportunities for the trader.

This research is important to demonstrate the effectiveness of our system to deliver round trip trades with reduced latency. We are committed to building a trading platform in the cloud to meet the needs of traders seeking ultra-low deterministic latencies. The goal is to provide a true and real-time trading experience as we demonstrate in studies that we can reduce latency in the cloud to a minimum.

III. LITERATURE REVIEW

Many companies have published their opinions on the importance of low latency systems especially when it comes to High Frequency Trading (HFT) which has gained a strong foothold in financial markets. This has been driven by several factors, including advances in information technology that have been conducive to its growth. Unlike traditional traders who hold their positions long term, high-frequency traders hold their positions for shorter durations, which can be as little as a few seconds. IBM and Mellanox [4] outline hardware requirements to achieve desired latency and throughput.

Tackling latency and reducing costs has always been a challenge when developing applications. Moallemi et al [4], explain why trading with low latency is valuable to investors as they need to be able to update orders in a timely fashion in response to new information. The cost of latency in trade execution is illustrated in an example showing how the median latency cost more than tripled while the median implied latency decreased when examining NYSE common stocks from 1995 to 2005. The study that was performed mathematically quantifies the cost of latency when traders are deciding between limit and market orders[1]. As latency impacts all market participants, their analysis suggests that the ability to trade with low latency results in quantifiably lower transaction costs. The method outlined in this study provides clear qualitative insight into the importance of latency. To quantitatively assess the cost of latency, the study contrasts the results in both the presence and absence of latency. Overall their results suggest that the difference in payoff between trading with a human time scale (500 milliseconds) and an automated trading platform with ultra-low latency (1 millisecond) is approximately of the same order of magnitude as other trading costs faced by institutional investors. This observation certainly underlines the significance of latency for such investors.

Achieving lowest latency without moderating jitter does not address all trader needs. As stated in [7], algorithmic trade flow requires deterministic response time in order to work consistently and profitably.

The Tabb Group [8] is an international research and consulting firm exclusively focused on capital markets. Research areas include evaluating industry adoption of new technologies, such as cloud, as well as, evaluating industry solutions, such as low-latency trading systems. On cloud adoption, The Tabb Group states that 48% of industry participants are using public cloud for Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS) and/or "multi-

1. Limit order is a trade request entered at a trader specified price. Market order is a trade that enters the market at the best price (lowest offer price to sell and highest bid price to buy).

cloud" [9]. Their report on cloud adoption [10] states that "adoption within capital markets shows an industry that has moved beyond pondering cloud's suitability, to one that is actively architecting the enterprise deployment of hybrid and/or multi-cloud environments.". Adopters include BlackRock, Credit Suisse, Goldman Sachs, HSBC, JPMorgan, MUFG, UBS, DTCC and FINRA.

view and approach are to build low-latency trading systems with industry standard technology. We believe that the advantages are manifold: availability of industry standard development tools and languages, fast/confident deployment, portability, vendor choice, seamless upgrade, cost and availability. Our experience corroborates H. Subrmoni, et al. [17] team's finding.

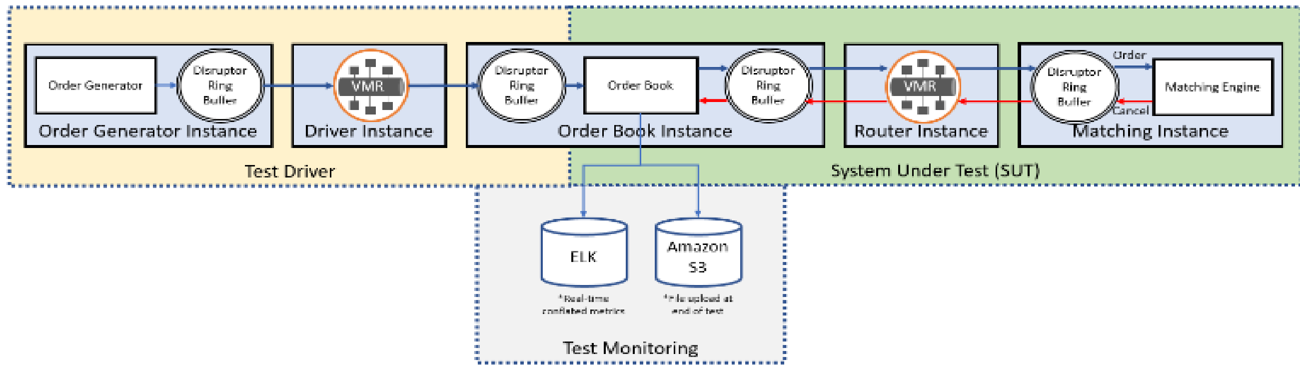

Figure 1: System Architecture

Securities Technology Analyst Center (STAC) [11] coordinates a community called the STAC Benchmark Council chartered to create specifications for performance testing capital market solutions [12]. They have created a number of testing specification across multiple domains. Their STAC-A2 (derivatives risk), STAC-A3 (strategy backtesting), and STAC-M3 (enterprise tick analytics) specifications have been tested in public cloud environments. But the trade execution specification, STAC-E, remains in draft form with no published benchmark results. We are not aware of any benchmark specifications, such as TPC-C [13], for comparing and evaluating trade execution systems. Due to the lack of an industrywide sponsored testing specification, trade execution system test specifications are unique/test, which is what we've done here. However, we've endeavored to clearly explain our system.

There is a plethora of approaches to reduce latency while increasing message rate in trade flows which includes integration of hardware-based solutions. Zoican and Vochin [14] propose a solution that embeds Graphic Processing Units (GPUs) to reduce latency. A. Boutros et al. [15] embed FPGA's into their solution and focus on building a development tool chain to effectively manage the code pushed to the FPGA. Haohuan FU et al. [16] deployed an FPGA-based trading solution for the China Financial Futures Exchange (CFFEX). They claim that their FPGA-based solution reduced latency from 100+ microseconds to 2 microseconds. H. Subramoni et al. [17] built a feed handler on an industry standard server by using best-in-class commodity hardware and an innovative software design and implementation that optimizes performance. The authors'

One industry standard technology used for reducing latency is the Disruptor pattern. The Disruptor ring buffer was developed by LMAX as an alternative messaging pattern to the use of bounded queues. According to studies performed by LMAX, after profiling various implementations, it became evident that the queuing of events between stages in the pipeline was dominating the costs. It was found that queues also introduced latency and high levels of jitter. The LMAX disruptor was designed to attempt to maximize the efficiency of memory allocation, and operate in a cache-friendly manner so that it will perform optimally on modern hardware. At the heart of the disruptor mechanism sits a pre-allocated bounded data structure in the form of a ring-buffer. Data is added to the ring buffer through one or more producers and processed by one or more consumers. By making these performance improvements, they suggest that the mean latency is much lower than an equivalent queue-based approach and have seen over 25 million messages per second and latencies lower than 50 nanoseconds [18].

IV. Test Environment

Our test system implements an order book and FX matching engine in an "immediate-or-cancel" trading scenario (see Figure 1). The order book is supplied with orders by an upstream generator, which publishes order messages to a message router, to which the order book is subscribed. The order book forwards these messages to the matching engine via a second message router.

The **Order Generator** constantly creates new open Order objects (status = "O") for its configured currency pair, then

creates a new Order Message and publishes it via its Producer disruptor to the order topic on the Solace Order VMR.

The **Order Book** subscribes to the order topic on the Solace Order VMR via a Consumer disruptor. Upon receiving a new Order Message, the Order Book decodes the message and stores it in the Order Book. The Order Book then sends another Order Message with a new timestamp via its Producer disruptor to the order topic on the Solace Matching VMR. The Order Book also subscribes to the cancel topic on the Solace Matching VMR via a Consumer disruptor. Upon receiving a cancel Order Message, the Order Book removes the Order with the matching order ID from the Order Book.

A standard **Matching Engine** traditionally matches buy and sell orders within a given market**.** The implemented test engine simply receives an order and immediately responds with a cancel message. This simulates the most common action on an FX exchange. The Matching Engine subscribes to the order topic on the Solace Matching VMR via a Consumer disruptor. Upon receiving a new order message, the Matching Engine creates a copy of the order message but sets its status to "C" (for "Cancel"), sets a new timestamp, then sends this cancel order message to the cancel topic on the Solace Matching VMR via a Producer disruptor.

The test system is deployed to provisioned instances on a cloud platform within an affinity group to guarantee that allocated resources are in the same data center and physically collocated. All instances are provisioned with identical specifications, as made available by the cloud vendors. Enhanced networking is configured as available by the cloud vendors. Each instance is provisioned running Red Hat Enterprise Linux 7.5. Clocks in the data centers used by the cloud providers were confirmed to be synchronized to a GPS clock. We install Chrony locally on the instances, which typically synchronizes system clocks at an accuracy within tens of microseconds on a LAN.

Each message routers hosts a Docker container running the Solace Virtual Message Router [19]. The Solace VMR is a version of the SolOS messaging middleware platform which runs on commodity hardware, rather than the company's purpose-built appliances. The Solace VMR allows message routing capabilities to be deployed to a cloud environment, where they can be scaled horizontally to provide improved throughput and redundancy. The test application sends messages to the router in direct message mode, that is, subscribers to a message topic, does not send an acknowledgment after receipt of a message, and the Solace VMR does not persist the messages. This was a design decision to prioritize achieving and measuring the lowest possible latency rather than guaranteed delivery. The application uses the Solace Java API (also known as JCSMP) to send and consume messages from the Solace VMR. Solace also provides a Java Real-Time Optimized (RTO) API, which is a Java Native Interface (JNI) wrapper for Solace's C API. As the message objects are statically allocated in our application, by reusing the objects in a ring buffer, the invocation of the garbage collector is eliminated in our application. This reduces the benefit of using Solaces' Java RTO API. Instead, to achieve lowest latency, the application establishes the session by setting Solace's MESSAGE_CALLBACK_ON_REACTOR property, which configures Solace to directly deliver messages to the application's listening thread from an I/O thread, bypassing a consumer notification thread.

The test system is deployed to cloud instances using Terraform [20] to automate configuration and provisioning of instances, including executing an Ansible playbook locally on each host. The Ansible playbook retrieves the test application from a remote repository and applies various modifications to the host OS and kernel to tune the system to optimize latency.

The test application is written in Java and runs on the JVM with the configuration shown in Table 1. JVM Configuration.

The test application uses the LMAX implementation of the Disruptor pattern [18] to provide high-performance inter-thread messaging in a thread-safe environment. Solace message objects are read into the statically allocated Disruptor ring buffer, reusing the message object to read, modify, and send a response message.

Table 1. JVM Configuration

JVM Parameters
-XX:UseG1GC
-Xms10G
-Xmx10G
-XX:+ExplicitGCInvokesConcurrent
-XX:+ParallelRefProcEnabled
-XX:MaxGCPauseMillis=5
-XX:InitiatingHeapOccupancyPercent=0
-XX:+UnlockExperimentalVMOptions
-XX:G1NewSizePercent=5
-XX:G1MaxNewSizePercent=30

The test application allocates all objects statically to avoid GC (garbage collection) events. We've observed that GC events introduce increased latency and therefore spent considerable effort to avoid them. Many specific OS-level and kernel-level tunings are applied in an effort to lower system latency and reduce activity interfering with the test application. For example, control groups are used to isolate a cpuset of cores for the test application process to use, and pin many OS and kernel services which cannot be disabled onto a different cpuset of cores, including IRQ interrupts. All tunings in the RHEL network_latency tuning profile are applied using the tuned-adm tool included with RHEL.

V.PERFORMANCE METRICS

The test system collects latency statistics in memory for the duration of the performance test and writes them to disk upon conclusion of the test. Metricbeat [21] is installed on the application instances and is used to send system metrics to an Elasticsearch cluster, including CPU, memory, and network I/O usage. Filebeat [22] is used to collect the latency statistics files written to disk storage and published to Elasticsearch.

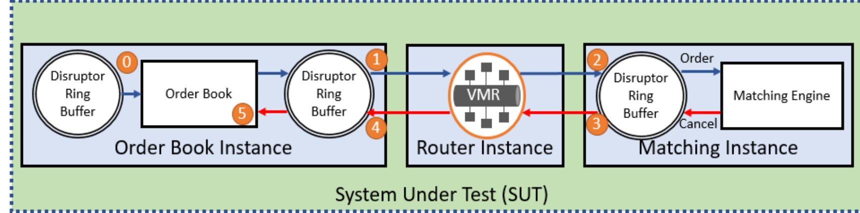

Figure 2: Latency Logging Points

Latency is recorded by invoking Java's *system.nanoTime()* timestamp method. For each order, latency is recorded at 5 points along the flow of data through the test system as shown in **Error! Reference source not found.**. Timing calculation starts when an order is read from the input ring buffer after injection by the Test Driver. The time delta is then calculated at each of the following five points:

1. The order book publishes an order to the Router VMR.
2. The Matching Engine receives the Order from the Router VMR.
3. The Matching Engine publishes the Cancelled Order to Router VMR.
4. The Order Book receives the Cancelled Order from the Router VMR.
5. The Order Book marks the Order as completed.

Total latency is the sum of the latencies measured at these 5 points along the flow of data.

For short-tests, less than one hour, latency statistics are kept in memory and flushed to Elasticsearch at end of test. For long-tests, latencies are conflated at one second intervals and sent to Elasticsearch in real-time. Metrics are visualized using a Kibana dashboard.

In parallel with the Trading Test, a client process on the Matching Engine Instance bounces a message off a server process in the Order Book Instance to directly measure the roundtrip latency (bypassing the Router VMR) and estimates the clock skew between the two systems. Clock skew is calculated from the roundtrip latency, presuming that each leg of the roundtrip should have roughly equal latency. The client sends a message to the server including a timestamp $T1_{client}$ and receives a response including a timestamp on receipt by the server (T_{server}). The client then records its own timestamp of the receipt of response ($T2_{client}$) and computes the roundtrip latency L_{RT}, then compares the expected server timestamp (half of the roundtrip latency plus the initial timestamp) to the actual server timestamp. The difference is equal to the difference between the clocks on the two systems.

Equation 1: Roundtrip latency

$$L_{RT} = T2_{client} - T1_{client}$$

Equation 2: Clock Skew

$$Clock\ Skew = T_{server} - (T1_{client} + L_{RT}/2)$$

This data was collected across a number of tests and analyzed to estimate the Cloud Skew, the difference between two clocks at a point in time, and Clock Drift, change in Clock Skew over time.

VI. RESULTS

We performed a number of tests to assess system behaviour over varying conditions, including:

- Message rate
- Message size
- Test duration from 5 minutes to 1-week
- Instance types, including virtual and bare-metal
- Public cloud vendors (AWS, Azure and OCI)

We also tested various cloud, network and system configurations to determine the optimal configuration in order to minimize latency and jitter. We conducted our tests by changing one parameter at a time and measuring the impact. This required executing a number of tests, which we managed by building a fully integrated dev/test/deploy/run performance test/measure pipeline.

Results included are from tests conducted in June and July of 2018. We noted that during the course of this study, cloud providers were actively working to improve their platforms, which resulted in progressively improving test results, demonstrating the import of this subject area. This was particularly notable in the improvement of virtualization technology and support of bare-metal servers.

Our base configuration was a virtual instance running 16 physical processor cores with hyperthreading disabled at the OS level (that is, a hypervisor was still running but was not

scheduling threads to run on sibling cores). Our application runs on a JVM configured with 10GB of memory with a ring buffer size of 1,024 message objects. Baseline tests were executed for 30 minutes with a message payload of 100 bytes sent at a rate of 1,000 messages a second.

We excluded Physical CPU Core Count, Logical CPU Core Count, Memory Size and Network Bandwidth from this report in order to protect cloud provider anonymity. With this information, a savvy reader could determine the cloud provider from the instance shape. The authors believe that comparing cloud providers distracts the reader from the report's purpose and is not required to justify our finding.

All test environments satisfied the following configuration:

- 24 to 72 Physical CPU Cores
- 2 Logical CPU Cores / Physical CPU Core
- At least 128GB of Physical Memory
- At least 14Gbps of Network Bandwidth

A typical minimum round-trip latency measured was 231µs, with a median latency of 318µs. We identified network jitter typically characterized by a 50-75% increase in latency from minimum to 50th percentile latency (best case: 12% increase) and a 150-300% increase from minimum to 99th percentile latency (best case: 91% increase). Maximum latency recorded was typically on the order of 50-100ms.

Table 2. 30-minute test performance statistics

Date	2018-07-10
Instance Type	Virtual
Message Rate	1,000 msg/s
Average Latency	377µs
Minimum Latency	231µs
50th percentile	318µs
75th percentile	413µs
95th percentile	577µs
99th percentile	669µs
99.9th percentile	5.017ms

We identified that the Solace VMR is currently the primary source of jitter in the environment, by comparing its latency against messages directly sent between the Order Book and Matching Engine, bypassing the VMR. We reported our results to Solace. Solace replicated the problem and verified that the VMR version of their product produced this jitter.

Various bare-metal server configurations, containing 32+ physical cores, were tested. We conducted this comparison test on a cloud provider that offered virtualized and bare-metal instances. Again, like virtual instances, we disabled hyperthreading at the OS level. The use of bare-metal servers typically resulted in improved stability and lower minimum latency. Compared to the virtualized instance, the latency was reduced by ~26% for the 95th percentile, however sometimes the 95th % were greater (example shown in Table 3). We hypothesize that these outliers are caused by network jitter and not the bare-metal server. Further testing would need to verify this hypothesis.

Table 3. 25-minute test - Comparison of virtual and bare-metal instance

Date	2018-07-17	2018-07-17
Instance Type	Virtual	Bare-Metal
Message Rate	1,000 msg/s	1,000 msg/s
Avg Latency	497µs	399µs
Min Latency	282µs	232µs
50th %	486µs	347µs
95th %	590µs	434µs
99th %	657µs	1,012µs
Max Latency	133ms	191ms

In our tests, very little of the system roundtrip latency is attributed to application processing, with the majority of the latency attributable to network latency. We experienced latency spikes of up to 1.5µs across various system components, for a total average latency of under 5µs due to application processing. It is evident that jitter experienced in overall system latency primarily originates from the Solace VMR.

Cloud Provider Comparison @ 10,000 msgs/s

We tested our configuration on instances offered by three prominent cloud vendors for 30 minutes. For these tests we increased the message rate to 10,000 msgs/s. The overview below demonstrates that multiple cloud offerings support our configuration with varying best-case performance results.

Table 4. Comparison of instances offered by different cloud vendors – best performance achieved

	Cloud 1	Cloud 2	Cloud 3
Date	2018-07-17	2018-07-17	2018-07-17
Instance Type	Virtual	Bare-Metal	Virtual
Msg Rate	10,000 msg/s	10,000 msg/s	10,000 msg/s
Avg Latency	251µs	399µs	633µs
Std Deviation	185µs	1,198µs	426µs
Min Latency	173µs	232µs	298µs
50th %	242µs	347µs	568µs
95th %	314µs	434µs	966µs
99th %	357µs	1,012µs	1,519µs
Max Latency	45ms	118ms	52ms

Although we started our testing with similar instance shapes, ultimately, we had to use different shapes in each cloud provider. Because we are only sending orders through for a single product and we pin cores, most cores are unused. Therefore, a high core count does not guarantee better performance. In addition, system memory size was not significant to test because application was configured to allocate ony 10GB of memory.

Cloud Provider 1 reported the best results responding to 99% of all orders within 357µs at a rate of 10,000 msgs/s. Table 5 proves that Cloud Provider 1 can sustain this latency up to 7-hours. Cloud Provider 2 high variance of 1ms is due to outliers above the 95%. Cloud Provider 3 does not have an outlier problem. In fact, its max latency is only 16% higher than Cloud Provider 1's max latency, but their average latency is 2.5 times higher than Cloud Provider 1. Variance is only 2.3 times higher.

Although there is a high variance in results between these three cloud providers, the testing shows that you can achieve deterministic low-latency on a public cloud.

Single-Day Test Results

We ran performance tests for the same three cloud providers over a 24-hour period to determine whether time of day impacts total latency (see Table 5). For Cloud Provider 1 the driver sent messages at a rate of 10,000 msgs/s. This test stopped after 7-hours due to insufficient disk space, but we included it in this report because it validated latencies reported in Table 3. Figure 3 shows the total latency for Cloud Provider 1. Note that the y-axis is a log function for all graphs.

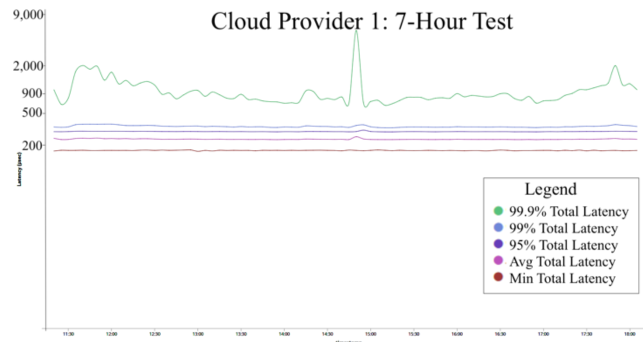

Figure 3: Cloud Provider 1: 7-Hour Test

The 50th, 95th and 99th percentiles were not captured for Cloud Provider 2 because latency measurements were conflated. Figure 4 shows the total latency.

The tests on Cloud Provider 2 ran for 24-hours at 1,000 msgs/s. Graph 4 shows the max latency, not the 99%.

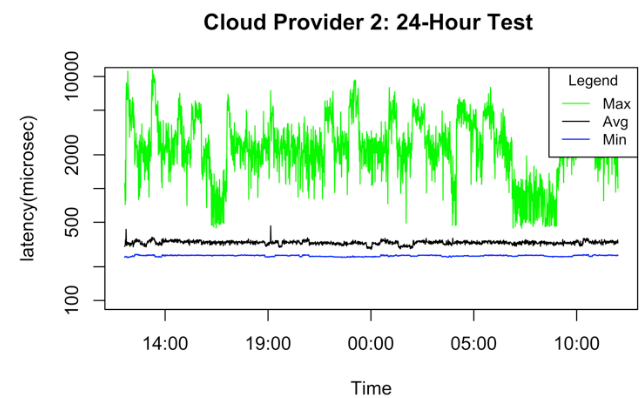

Figure 4: Cloud Provider 2: 24-Hour Test

The tests on Cloud Provider 3 ran for 24-hours at 1,000 msgs/s. Although the latencies are higher, the 99% is still less then 1 ms.

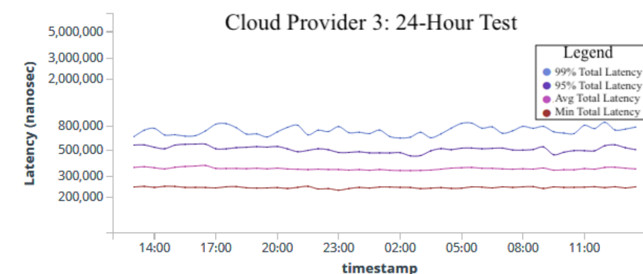

Figure 5: Cloud Provider 3: 24-Hour Test

The 7-hour test for Cloud Provider 1 reported results consistent with the shorter test reported in Table 4. Cloud Providers 2 and 3 reported higher latencies then Cloud Provider 1, but results are consistent with Table 4, except

the 99th % for Cloud Provider 3 is ½ the shorter test while the max is double.

It is important to note these tests were run only for a single day using a single data center and therefore should not be used to draw definitive conclusions. While we did not note a significant slowdown or problems over the single day, we did not collect enough evidence to make a statistically significant conclusion.

Table 5. 24-hour Test Results

	Cloud 1	**Cloud 2**	**Cloud 3**
Date	2018-07-20	2018-06-21	2018-07-11
Instance Type	Virtual	Bare-Metal	Virtual
Msg Rate	10,000 msg/s	1,000 msg/s	1,000 msg/s
Avg Latency	240µs	328µs	350µs
Min Latency	168µs	233µs	230µs
50th %	230µs	-	306µs
95th %	298µs	-	519µs
99th %	345µs	-	727µs
Max Latency	33ms	117ms	110ms

7-Day Test Results

We ran two performance tests over a 7-day period (1-week) on Cloud Provider 3 to determine whether day of week impacts latency. These tests ran concurrently; one test on Virtual Instances and one test on Bare-Metal Instances. We were unable to perform the 7-Day Test on Cloud Provider 1 and 2 due to cost constraints.

Table 6 reports minimal, average and maximum latency for these tests. We did not capture percentiles because results were conflated at source in order to report test progress in real-time through an Elasticsearch Cluster.

We observed that the test running on bare-metal instances reported stable mean response times throughout the 7-day test. However, after running for 2-days 4-hours, the Max Latency on the bare-metal instances and Mean and Max latency on virtual instances reported increased jitter. Our analysis of the virtual instance test attributes this increased jitter to the Order Book Server, which may be caused by "Noisy neighbors". The increased jitter in the bare-metal test is due to the Matching Engine Server. Because these are bare-metal servers, it cannot be due to a "Noisy neighbor". Understanding the jitter source for test tests requires further investigation into this cloud provider.

Table 6. 7-Day Test Results

Date	2018-06-25 -> 2018-07-02	2018-06-25 -> 2018-07-02
Instance Type	Virtual	Bare-Metal
Message Rate	1,000 msg/s	1,000 msg/s
Avg Latency	517µs	324µs
Min Latency	333µs	234µs
Max Latency	96ms	129ms

It is important to note that this test was only run once on virtual servers and once on bare-metal servers in the same cloud provider and therefore should not be used to draw definitive conclusions. Although we measured increased jitter on virtual instances, the bare metal instances reported consistent latencies throughout the test supporting the hypothesis that public cloud can support stable low response times.

Clock Skew Results

Our analysis of clock skew demonstrated that while network latency between the Order Book and Matching Engine instances is acceptably stable, the clocks are not sufficiently synchronized to meet MiFID II regulations, which states that clocks must achieve microsecond precision and a divergence of not greater than 100 microseconds. We ran a test of clock skew between two servers for 24-hour, which measured that 84% of the clock skews were within 100 microseconds while 99% where within 225 microseconds with a max of 285 microseconds.

However, Stanford University and Google Inc have proposed a new clock synchronization protocol [23] which they claim achieves clock synchronization to within 10's of nanoseconds across servers in the same Data Center on standard Network Interface Cards (NIC) without specialized Precision Time Protocol (PTP) Hardware [24].

VI. FUTURE WORK

Based on these test results, several opportunities for enhancement of this study are presented.

Future work can be segmented into seven broad areas:

- Complete Trading System
- Non-persistent Solace optimization
- Persistent Queueing
- Brokerless Queueing System
- Low-latency Stacks
- Stress Testing

- Public Coud Support

Complete Trading System:

This paper reported the latency of sending orders through a performance tuned environment that consisted of a simple Order Book and Matching Engine. We did not implement some elements of a complete trade environment such as FIX translation and matching, because we felt these elements are not necessary to test performance characteristics on a Public Cloud. Future work should include the missing features.

Further study should also test concurrent trading of multiple products, segregated by product pair, in order to better simulate the load diversity in a production foreign exchange trading system. As the number of product pairs increases, multiple trading system instances (Order Book, Solace VMR and Matching Engine) must be deployed to sustain latency profile. Further tuning will be required to minimize interference between product pairs.

Finally, migrate and test a live trading system to a public cloud, which includes deployment across multiple availability zones (geographical data center locations) to provide resiliency in the case of failure. This deployment should test 2 cases: (1) replication of the Matching Engine instance to a second availability zone, and (2) replication of both the Matching VMR and Matching Engine instances to a second availability zone.

Optimize non-persistent Solace:

For completeness, the test system should implement Solace's Java RTO API, using an IPC version of the C library which enables inter-process communication, further reducing notification time. It is presumed that designing the test system to take advantage of IPC will not appreciably reduce latency when considered against the variance of the latency observed across the Solace VMR in our performance tests, so this enhancement has been reserved for future work.

Persistent Queueing:

This study exclusively used the direct messaging mode of the Solace VMR. In a production trading system, delivery of messages is paramount, and future work should implement persistence of messages on a topic and acknowledgment of receipt by subscribers.

Brokerless Queuing:

A brokerless message system, such as ZeroMQ [25], nanomsg or nng [26], will eliminate two of the four network hops which would cut network latency in half. This approach would queue messages in the Order Book and Matching Engine Instances, as opposed to the separate instance.

Low-latency Stacks:

Remote Direct Memory Access (RDMA) [27] in combination with Converged Ethernet [28] or InfiniBand [29] reduces inter-system transfer time by enabling cut through routing and network controller direct access to application memory. This can reduce single message transfer time to several microseconds.

Stress testing:

In order to simulate network performance under real-world trading scenarios such as peak activity, we recommend testing the configuration under higher message volume. We suggest increasing message volume to a million msg/s across multiple products and measuring the impact on latency.

Public Cloud Support:

We also believe that Public Cloud Vendors can improve support for Low-Latency Trading Systems:

- Optimize server health agents to reduce introduction of jitter.
- Investigate network architecture in order to isolate network traffic.
- Implement low-latency network protocols, such as InfiniBand[2] or Converged Ethernet[3].

VI. SUMMARY

Low-latency trading in financial markets focuses on processing trades and retrieving market data at the fastest speed possible. Low latency is desirable as financial firms use speed as a competitive advantage. Trading in the cloud presents a new challenge of achieving low latency without direct access to the hardware. In this paper, we focused on test strategies performed in the cloud across multiple cloud providers to demonstrate the feasibility of implementing a low latency trading platform in a cloud environment. We deployed a simple trading system to a cloud platform and tested latency under various configurations and conditions. While achieving acceptable minimum latency, we identified undesirable network jitter, which we attributed to the use of the Solace Virtual Message Router; nevertheless, the stability of network latency was acceptable for some production trading platform. Additionally, an analysis of clock skew demonstrated that available clock synchronization methods are not yet capable of meeting MiFID II [5, 30] compliance in the cloud environment. Overall, we have demonstrated the ability to achieve our goal of sub-500 microsecond roundtrip

2. Azure supports InfiniBand today.

[3] OCI supports Converged Ethernet today.

latency and therefore conclude that it is currently feasible to build a production low-latency, high-frequency trading system in the cloud. Our ongoing focus will be to continue to reduce latency as it occurs and provide reliable access to the application.

VII. ACKNOWLEDGMENTS

We would like to thank Solace, Microsoft, and Oracle Corporation for their support and collaboration, and the use of their resources in concert with this effort. We understand that this research is of great importance to industry leaders in cloud computing and we appreciate their partnership.

VIII. REFERENCES

1. Loveless, J., *Barbarians at the Gateways.* Communications of the ACM, 2013. **56**(10): p. 42-49.
2. Aldridge, I., *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems*. 2nd Edition ed. 2013: Wiley.
3. Pagnotta, E.S. and T. Philippon, *Competing on speed.* Econometrica, 2011. **86**(3): p. 1067-1115.
4. Moallemi, C.C. and M. Saglam, *The Cost of Latency.* SSRN eLibrary, 2010.
5. Prorokowski, L.J., *MiFID II compliance – are we ready?* Journal of Financial Regulation

Compliance, 2015. **23**(2): p. 196-206.
6. FXCM. *How Does Latency Impact Trading?* 2018 [cited 2018; Available from: https://www.fxcm.com/insights/how-does-latency-impact-trading/.
7. Misra, H. *The Next Frontier in Exchange Trading System, Part 2*. 2018; Available from: https://tabbforum.com/opinions/the-next-frontier-in-exchange-trading-systems-part-2-of-4.
8. Group, T. *Tabb Group Home Page*. 2018; Available from: https://www.tabbgroup.com/about2.
9. 2018 FinTech Festival Cloud Survey Results, presented at 2018 Fintech Festival, November 1, 2018.
10. Summerville, M. *Capital Markets: Heads in the Cloud*. 2018; Available from: https://research.tabbgroup.com/report/v15-053-capital-markets-heads-cloud.
11. *STAC Research Website*. 2018; Available from: https://stacresearch.com.
12. Council, S.B., *Capital Markets Workload Categorization.* 2018.
13. *Transaction Processing Performance Council* 2018; Available from: http://www.tpc.org.
14. Zoican, S. and M. Vochin. *Computing system and network architectures in high frequency trading financial applications*. in *Communications (COMM), 2016 International Conference on*. 2016. IEEE.
15. Boutros, A., et al. *Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis*. in *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. 2017. IEEE.
16. Fu, H., et al. *Accelerating Financial Market Server through Hybrid List Design*. in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017. ACM.
17. Subramoni, H., et al. *Streaming, low-latency communication in on-line trading systems*. in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. 2010. IEEE.
18. Thompson, M., et al., *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads.* Technical paper. LMAX, May, 2011: p. 206.
19. Solace. *Solace Virtual Message Router Product Datasheet*. 2017; Available from: https://solace.com/wp-content/uploads/2017/02/Solace-VMR-Datasheet.pdf.
20. HashiCorp. *Terraform*. [cited 2018; Available from: https://www.terraform.io/.
21. Elastic. *Metricbeat*. [cited 2018; Available from: https://www.elastic.co/products/beats/metricbeat.
22. Elastic. *Filebeat*. [cited 2018; Available from: https://www.elastic.co/products/beats/filebeat.
23. Geng, Y., et al. *Exploiting a natural network effect for scalable, fine-grained clock synchronization*. in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018.
24. Eidson, J. and K. Lee. *IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems*. in *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*. 2002. Ieee.
25. *ZeroMQ*. 2018; Available from: http://zeromq.org/.
26. *Nanomsg*. 2018; Available from: https://nanomsg.org/index.html
27. *Remote Direct Memory Access (RDMA) Consortium*. 2018; Available from: http://www.rdmaconsortium.org.
28. Guide, D.R.D., *2018 Edition.* 2018.
29. Association, I.T. *Infiniband Trade Association Home Page*. 2018; Available from: https://www.infinibandta.org
30. Riche, T., *Time Synchronization: Time is at the Heart of MIFID Regulation*. 2018: Tabb Group Market Note.