Trading Day Logs Replay Limitations and Test Tools Applicability

Pavel Protsenko
Exactpro Systems
pprotsenko@lseg.com

Andrey Alexeenko
Exactpro Systems
andrey.alexeenko@exactprosystems.com

Anna Khristenok
Exactpro Systems
anna.khristenok@exactprosystems.com

Tatiana Pavlyuk
Exactpro Systems
tatiana.pavlyuk@exactprosystems.com

Anna-Maria Lukina
Exactpro Systems
anmay@exactpro.com

Iosif Itkin
Exactpro Systems
iosif.itkin@exactpro.com

***Abstract* - This paper is an experience report on replaying full trading day production log files for dynamic verification of securities exchange matching engines. Three types of test automation tools developed in-house are described along with their characteristics. The paper analyzes various approaches to reproduce processes and scenarios observed in the systems during their production usage. The applicability and limitations of these approaches are also considered. The authors point out that for most complex distributed real-time trading systems it is extremely difficult to obtain an identical behavior using production logs replay via external gateways. It might be possible to achieve this by implementing additional instrumentation inside the exchange system's core. The authors assume however that such an intrusion has limited value and should not be prioritized over other, more appropriate, test design methods for testing such systems.**

Keywords – test tools, trading systems, matching engines, exchanges, test design, data replay

I. Introduction

Quality and reliability of stock exchange trading platforms is crucial for integrity of the global financial markets. Rapid increase in the volumes of transactional data, the never ending race to zero latency and growing complexity of the market infrastructure is what turns modern exchanges into very large distributed real-time systems presenting significant testing and maintenance challenges [1]. Exchange operators want confidence knowing that changes introduced into the system will not result in regression problems. Quite frequently operators express a desire to have the ability to reproduce production activity for a given trading day in a test environment.

Recorded data replay is widely used across different industries, including telecommunications [2], web-portals [3], and industrial automation [4]. In addition to the ability of repeating the normal behavior of systems, record replay can be used as an effective way of reproducing failures that occur in the field during production exploitation of software systems. Intensive research is underway to propose instrumentation required to replicate the activity in multi-threaded environments [5, 6, 7]. A substantial effort is targeted at reducing the overhead caused by the instrumentation without losing the ability to correctly replicate the sequence of events.

As most of high frequency trading systems are trying to avoid any possible overhead, their authors often face a strong push from the stakeholders towards minimizing any additional internal instrumentation and relying on external gateways in implementing record replay test harnesses. This paper contains an industrial case study of attempts to use several types of proprietary test tools in an effort to satisfy such requests. The authors discuss a set of limitations encountered along the way.

Part of the paper discusses some basic aspects of stock exchange architecture and functionalities and ways to obtain activity logs from large scale trading systems. Part III outlines the main characteristics of three in-house test automation tools: Sailfish, Load Injector and Mini-Robots. The last part contains the analysis of the authors' observations in the course of the mentioned experiments.

II. Securities Exchange Technology Platform

The authors believe that his part may be useful for researches and industry practitioners working in test tools and electronic trading areas as an overview of exchange systems architecture and a description of test tools types used in this domain. Security exchanges are at the very foundation of the modern markets, but their architectures are not covered in research publications. The authors attempt to list key electronic exchange components and their interaction. This overview is not based on any particular system, but rather aggregates knowledge obtained through many engagements targeted at verifying modern and most advanced trading platforms.

We are providing a very simplistic view of the trading flow. The benefit is that it clearly shows the interface between the client and the exchange, which, effectively, is the entry and exit point for messages sent and received by the client. The messages reach the access point, get routed within the exchange and finally get into the matching engine which reports back to the access point and eventually the client. The natural first thought is to somehow capture the flow from the access points. Generally, there are two options: either the access point logs or the network capture. Neither of the options is ideal: quite often, the log level on access points is set to Minimal to improve performance and network capture is not packet loss free.

Also Fig. 1 helps understand the scalability approach to the trading systems. Normally, there are multiple processing layers to allow for load balancing and initial hot redundancy, as well as mirrors to the key components on, in some cases, the whole system in a stand-by mode acting as a mirror to the primary one. Exchange systems can be scaled by distributing participants between different access points. Exchange systems scalability by instruments is possible if the traded securities are independent from each other. Any intersections like strategies or shared risk limits can dramatically reduce the potential for horizontal scalability.

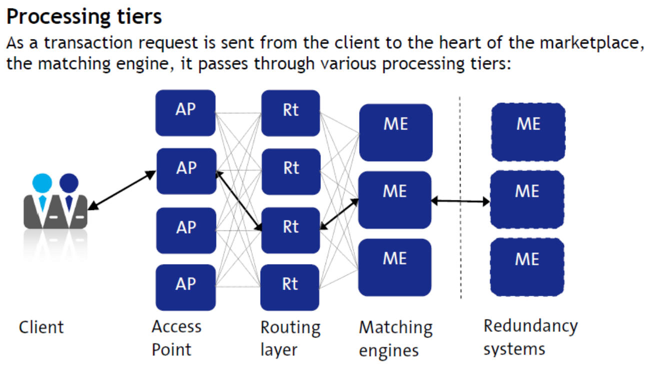

Processing tiers
As a transaction request is sent from the client to the heart of the marketplace, the matching engine, it passes through various processing tiers:
AP
AP
AP
AP
Rt
Rt
Rt
Rt
ME
ME
ME
ME
ME
ME
Client
Access Point
Routing layer
Matching engines
Redundancy systems

Fig. 1. Processing tiers and horizontal scalability approach in Exchange Systems.
Source: A Cinnober white paper on: Latency [8]

As shown on the diagram, several distributed levels are involved prior to a point in time when the message reaches the matching engine core. It means that the data needs to be collected from many servers and time synchronization between the boxes becomes a serious issue. Even when timers of extreme precision and exact inbound sequencing are established, there is no guarantee that the message that comes to one access point earlier will get into the matching core earlier.

The following diagram lists typical components present in a securities exchange system:

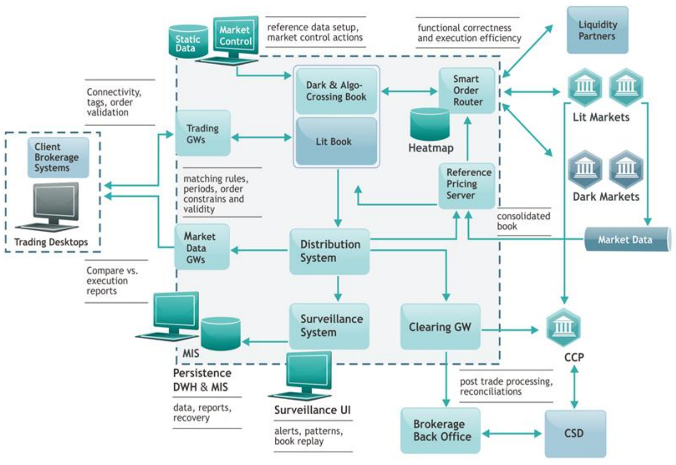


Fig. 2. Typical exchange system components

Access points are represented by Trading and Market Data Gateways. Clients' systems connect to them using API or though the trading desktop. APIs can be implemented as a network specification or can be provided as a linkable library. Data record and replay is usually a more complicated process in the latter case. Trading gateways receive information from particular participants and send back responses dedicated to particular participants. Market Data gateways on the other hand disseminate information market wide. In most cases generic purpose public market data lacks some of the information required for replay. In order to monitor trading activity of the clients or desks within the same trading firm, participants often use Drop Copy gateways. These gateways send a copy of all messages received by the monitored connections. It is possible to use Drop Copy data if it is configured to track executions for all clients, which is rarely the case due to performance limitations.

The core of exchange system is a matching engine, the component responsible for crossing inbound orders and determining execution prices. The matching engine needs to sequence all incoming orders prior to determining the crossing outcome for every inbound event. This sequencing can happen inside the matching engine or prior to it on the routing and sequencing level. There are various types of matching cores. Some are price forming that determine execution prices on the basis of the orders submitted into the market, others are price referencing that uses prices produced elsewhere. Hybrid options are also available, where the prices are formed inside the exchange but some of the external pricing information is used, e.g. currency data or interest rates. The component that provides external pricing information is labeled as Reference Pricing Server on the diagram. Data replay for price reference and hybrid markets requires capturing not only client traffic, but also all external pricing information and its synchronization.

Some exchange systems not only take prices from external markets, but pass through inbound client orders when liquidity is not available on exchange. The component responsible for the process is called a Smart Order Router (SOR). SOR checks the liquidity available on other markets and executes the order in the optimal way. The presence of such a system broadens the number and complexity of communications links. Data replay for such a system also becomes a dual-stage process where the test tools have to submit inbound messages and afterwards properly react to the outbound messages from the system.

Once matching engine has determined the crossing outcome, it is necessary to distribute this information to a multitude of other components, including: trading and market data gateways, clearing, market surveillance, data persistence and warehousing, other post-trade and back office systems, etc. In order to avoid an extra load on the matching core, a separate sub-system is usually used for data distribution. The corresponding component is labeled as the Distribution System. Sometimes the presence of a dedicated system can help with data collection for replay.

The diagram shows two components important for data record and replay: Surveillance System and Data Persistence. The latter one is required to capture the data for analysis and reporting. Usually it is implemented as an asynchronous logging process that stores transactional data in a set of files later uploaded into a relational database. This data is also useful to restore the system's state in case of an outage. Market surveillance systems are targeted at helping exchanges to maintain orderly markets by analyzing all events in the system to check if any of them can represent signs of malicious participants' behavior, such as prices or volumes manipulation, money laundering, front-running, etc. Both Data Persistence and Surveillance Systems receive information about all events that are already correctly sequenced. This can potentially make them a perfect source for data record and replay. However, most of the time these systems process only business related data fields and do not store the data related to the low level networking technicalities.

On the other hand, interface logs and network captures contain the required low level details, but not necessarily properly sequenced and timed. The authors consider that in the future convergence can happen between technical monitoring and fraud detection [9] and the next generation of the Market Surveillance systems will capture both business data fields and low level networking details. This will make them a better source for data replay activities.

III. Test Automation Tools

There are two types of testing activities required to deliver software systems: functional testing and non-functional testing. The goal of functional testing is to verify that a system satisfies expectations in terms of its functionality. Non-functional testing is an activity targeted at validating the attributes of a system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability, and portability. Latency, throughput and capacity are critical non-functional characteristics the high frequency trading systems. Functional and non-functional testing are both important for the orderly functioning of the financial markets.

The authors have participated in a number of projects that put a series of innovative exchange trading systems live. In the course of this work, a set of test automation tools was created to cover all necessary aspects of the quality assurance process. This part covers three tools:

a) Sailfish – functional testing;

b) Load Injector – load testing;

c) Mini-Robots – testing at the confluence of functional and non-functional testing.

Sailfish

Sailfish is a test automation tool whose primary target is testing of bi-directional message flows in distributed trading platforms and market data delivery systems. It is a back-end tool that is typically connected to message gateways / APIs utilized by trading or market data traffic.

The purpose of Sailfish is to minimize manual intervention required to execute test suites. In its more sophisticated deployments, Sailfish makes it possible to achieve fully autonomous scheduled test execution that does not require on-going operator monitoring.

Sailfish has a modular structure whereby a shared framework is used in conjunction with specialized plug-ins. Separate plug-ins are used for each protocol version, including industry-standard protocols, such as FIX [10], SWIFT [11], etc., and proprietary protocols.

Test Libraries developed for Sailfish include tests for a variety of business contexts (Regulated Markets, MTFs, Dark Pools, Clearing Houses, and Brokerage Systems) realized in a wide range of technical and middle-ware infrastructures.

Sailfish is a simple keyword-driven test tool. Test scripts are specified in Comma Separated Values (.CSV) format. Each line in the file contains one of the following:

- *Send valid and invalid messages into system under test;*
- *Compare received messages with a filter and create a report with comparison results;*
- *Synchronization points and test cases start/stop markets;*
- *Wait for a predefined number of milliseconds between these types of actions.*

Every keyword has its corresponding action implemented as a Java class. Parameters from each line are passed to the class for processing. Parameters can be in a form of constant values, java functions, and references to the values in previous steps.

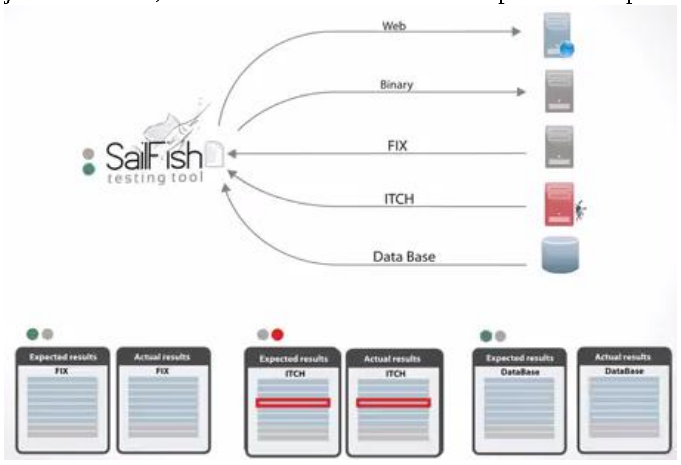


Fig. 3. A generic representation of Sailfish

To turn production data into Sailfish test scenarios, one needs to generate a set of .csv files. Outbound messages are translated into Send actions, inbound messages are translated into Receive and Compare actions. The key challenges with replaying the data are:

a) Efficient usage of the functions and references;

b) Correct specification of synchronization points and wait times.

Functions are required to provide values for the fields that can't be used as constants during replay, e.g. timestamps and unique client order identifiers. References are required when outbound messages need to refer to some inbound data, e.g. generated internal exchange identifiers. Synchronization points are responsible for splitting message flow between test cases and ensuring correct sequence of events for concurrent messages. The following table summarizes the main characteristics of Sailfish:

Capacity & Precision	Throughput 40 transactions per second with validations / 800 in performance mode. Time precision ~25-50 ms
Testing Type	Active Real-Time
Target SUT	Trading Platforms, Market Data Delivery and Post-Trade Systems
SUT Interface	Back-end (typically connected to message gateways / APIs, and DBs); GUI Testing Capabilities supported via plug-ins to other tools (e.g., Selenium)
SUT Interaction Method	Message injection and capture for testing of real-time low-latency bi-directional message flows; DB queries for data verification
Protocols	Extant plug-ins for Industry-standard (FIX and dialects, FAST, SWIFT, ITCH, HTTP, SOAP, etc.) and Proprietary (MIT, SAIL, HSVF, RTF, RV, Reuters, Fidessa OA, Quant House, etc.) protocols. New plug-ins for additional protocols developed by request (shared between Sailfish and Shsha)
Test Scripts	Human-readable CSV files; scripts generated manually by test analysts or automatically by test script generator using results of passive testing performed by other tool (e.g., Shsha)
Test Management, Execution and Reporting	Integrated (Web front-end), allows for multiple simultaneous heterogeneous connections, consecutive execution of multipl planned scripts, test results summary and detailed test reports REST API supports remote control of Sailfish instances. Optiona Big Button framework supported
Platform requirements	Written in Java. Low footprint cross-platform application MySQL or other RDBMS

Fig. 4. The main characteristics of Sailfish

Load Injector

Load Injector is a powerful load generator targeted at stressing scalable high load trading infrastructures [12]. Load Injector supports FIX (all versions), ITCH, LSE, Native, SOLA SAIL & HSVF, HTTP, SOAP, and various binary trading systems protocols. The tool's architecture allows flexible expansion into additional protocols.

Load Injector is an open-cycle load generator capable of supporting both model and measurement approaches of performance testing.

The tool relies on a set of methods to increase its efficiency:

a) managing the executed threads via a central controller
b) using pre-configured templates for generating message streams
c) coordinated processing of data obtained from a reverse data stream

Fig. 5. A generic representation of Load Injector

Due to performance requirements, Load Injector is not capable of transforming CSV or other data format into messages due to related overhead. That is why Load Injector relies on a set of raw data files that contain messages to be sent. It is necessary to perform minimal required modifications of the messages to be sent, such as timestamps, unique client order identifiers, checksums and other related fields. As in Sailfish, it is sometimes necessary to base outbound messages on inbound data. The process is implemented in an efficient manner via a centralized reversed data processing loop. Any outbound messages modifications and inbound messages processing required for successful data replay had to be implemented inside the Load Injector source code instead of the test scripts. This introduces additional difficulties in comparison to a functional testing tool. The following table summarizes the main characteristics of Load Injector:

Item	Description
Capacity & Precision	Throughput up to 75,000 msg per core per second. Total capacity hundreds of thousands messages per second. Precision is in microsecond range.
Testing Type	Active Load and Non Functional Testing
Target SUT	Trading Platforms, Market Data Delivery and Post-Trade Systems and their combinations
SUT Interface	Back-end (typically connected to message gateways / APIs; data streams generation: mcast/ucast); GUI Testing Capabilities not supported
SUT Interaction Method	Inputs and outputs are generated based on the configured load shapes, parameters and templates. Captured messages can be viewed and analyzed post-factum using the DB queries (Shsha) or/and performance calculator tool (also developed by Exactpro)
Protocols	Extant plug-ins for Industry-standard (FIX and dialects, FAST, ITCH, etc.) and Proprietary (MIT, SAIL, HSVF, RTF, RV, Quant House, etc.) protocols. New plug-ins for additional protocols developed by request
Test Scripts	Capable to stress the system with high rate of transactions including microbursts. Used for Throughput, Bandwidth, Latency tests. Can be used for support of fault tolerance (Failover) tests

Test Management, Execution and Reporting	Simulation of multiple client connections with specified load shape for each connection or group of connections (configure number of connections, messages templates, Load Shape for each connection or group of connections, messages distribution for each connection or group of connections) Simulation of market data streams with required SLAs
Platform requirements	Written in C++. Linux on 64-bit platform

Fig. 6. The main characteristics of Load Injector

Mini-Robots

Some defects are located on the border of functional and non-functional testing (for example, complex race conditions scenarios). It is difficult to catch such defects using ordinary functional test automation or load generation tools. The test tool named Mini-Robots is targeted at addressing these limitations over the course of conducting testing of trading systems. In a nutshell, this testing tool finds its place in between Sailfish, a functional testing tool, and Load Injector, a non-functional testing tool. The Mini-Robots tool has been developed with the idea to simulate real traders' behavior, i.e. reacting to specific market conditions in a common fashion, but, on the other hand, having a certain degree of freedom of how to react. Each of the robots acts independently and can execute a particular trading strategy or simply replay a stored list of orders. Mini-Robots uses realistic gateways to establish connectivity with the systems under test [13].

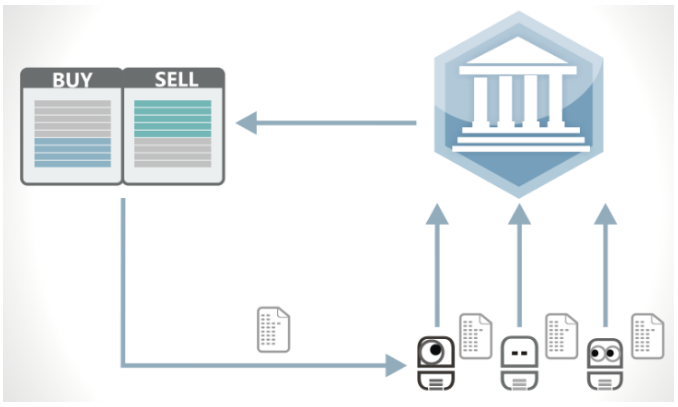


Fig. 7. A generic representation of Mini-Robots

The following table summarizes the main characteristics of Mini-Robots:

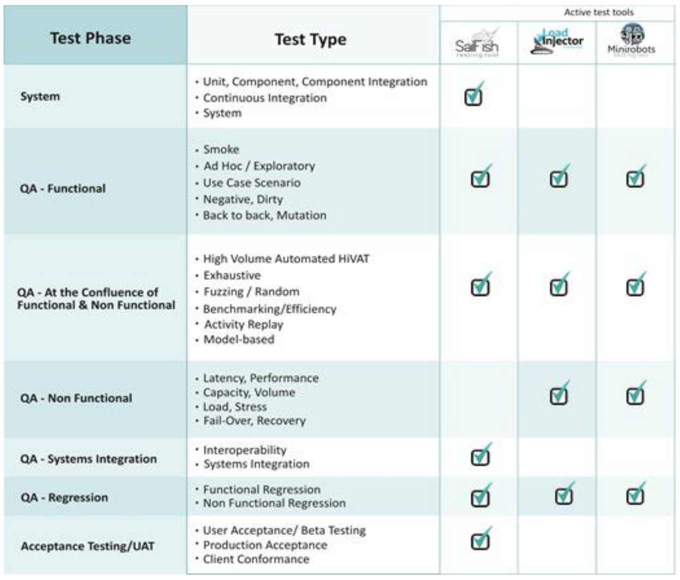
Item	Description
Capacity & Precision	Hundreds – thousands of messages depending on the algorithm complexity. Millisecond precision
Testing Type	Active Multi-Participants (applicable for testing at the confluence of functional and non-functional testing)
Target SUT	Trading Platforms and Market Data Delivery Systems
SUT Interface	Back-end (typically connected to message gateways / APIs); GUI Testing Capabilities not supported
SUT Interaction Method	Message injection and capture to emulate multiple participants' activity in electronic markets (essential when there is a need to reproduce complex scenarios that can be created by trading algorithms)
Protocols	Extant plug-ins for Industry-standard (FIX and dialects, etc.) and proprietary protocols. New plug-ins for additional protocols developed by request
Test Scripts	Multi-threaded Java code specifying different liquidity profiles
Test Management, Execution and Reporting	Integrated (Web front-end), allows for multiple simultaneous heterogeneous connections, concurrent emulation of multiple participants, detailed test reports. Optional Big Button framework supported
Platform requirements	Written in Java

Fig. 8. The main characteristics of Mini-Robots

Outbound messages based on an inbound flow are incorporated into Mini-Robots as they are similar to ordinary algorithmic trading systems, and, as such, maintain received data in a way required to send order amends/cancellation and are able to react on fill and market data signals.

Test Tools Applicability Comparison

The following diagram contains a summary of using the test tools:



Test Phase	Test Type	Active test tools		
		SailFish	Load Injector	Minirobots
System	• Unit, Component, Component Integration • Continuous Integration • System	✓		
QA - Functional	• Smoke • Ad Hoc / Exploratory • Use Case Scenario • Negative, Dirty • Back to back, Mutation	✓	✓	✓
QA - At the Confluence of Functional & Non Functional	• High Volume Automated HiVAT • Exhaustive • Fuzzing / Random • Benchmarking/Efficiency • Activity Replay • Model-based	✓	✓	✓
QA - Non Functional	• Latency, Performance • Capacity, Volume • Load, Stress • Fail-Over, Recovery		✓	✓
QA - Systems Integration	• Interoperability • Systems Integration	✓		
QA - Regression	• Functional Regression • Non Functional Regression	✓	✓	✓
Acceptance Testing/UAT	• User Acceptance/ Beta Testing • Production Acceptance • Client Conformance	✓		

Fig. 9. Test tool usage summary

IV. Replicating Trading Scenarios and Full Day Production Logs

The authors have used the three tools to test various matching engines across different asset types, including equities, listed derivatives, FX and interest rate swaps. The test tools were connected to the systems using FIX, ITCH [14] and a set of proprietary binary protocols. Another testing tool named Shsha [15] developed by Exactpro was used to turn production data into test cases for the tools. The authors and Exactpro QA teams have tried all sources of production data referred in Part II (interface logs, network captures, and surveillance/drop copy feeds) to produce test cases describing production activity.

There are three levels of complexity in matching engines behavior and corresponding challenges related to the order book replay:

a) Simple – confined order book independent for each instrument. This is usualy true for European lit cash markets

b) Reference price - instrument independent from each other that should take into account prices from some external market data feed. It is true for European dark cash markets

c) Strategies – multi-leg instruments and strategies, such as spreads, butterflies, condors, etc. Due to the presence of the strategies, instruments are no longer independent and any movement for a single instrument can result in changes across many other instruments through implied liquidity.

North American markets introduce an extra level of complexity due to the necessity of passing through to other markets orders that could not be executed within the National Best Bid Offer (NBBO) [16]. Record replay for such markets requires simulating both inbound and outbound SOR endpoints.

As expected, higher order book complexity results in extra challenges in order book replay. The ability to segregate the data by instrument can substantially reduce the volume of data that needs to be replayed in order to reproduce the order book state. Thus, replay techniques have appeared to be much more stable for confined central order books in contrast to interconnected strategy instruments with implied liquidity and reference price feeds.

The tests confirmed that it is possible to use all three tools to recreate steps for most of the observed failures. However, experiments also show that there is a decent chance of any given failure and that the data replay will not recreate the exact sequence of events at the first attempt. As expected, time sequencing of the events in distributed systems can be unreliable [17].

The first challenge is the precision of the injection process itself. Clearly, more light-weight injector written in C++ (Load Injector) is capable of obtaining better precision of the inbound flow. Load Injector works in a microsecond range. Mini-Robots have milliseconds precision, while the precision of Sailfish is at least an order of magnitude worse. Repeated experiments show that this factor is important to reproduce race-conditions and similar problems during micro-bursts. It turned out that the precision of Load Injector and Mini-Robots is sufficient to reproduce all scenarios encountered in practice.

Test tool precision has three main aspects:

a) Logical events sequencing
b) Time scale
c) *Absolute physical time*

The first aspect is important for any given scenario as the state of the order book and execution prices depend on the sequence of orders arriving on the market. A single recording can result in several replay options for the concurrent cases due to the proximity of outgoing and incoming messages. The time scale aspect is important for concurrency scenarios executed within the time frames comparable to internal processing delays, e.g. within a millisecond. Apart from the simplest market structures, there are many cases where absolute timing becomes important. The main ones are the trading cycle transitions, such as market opening/closing, and good till time orders.

The ability of replicating recorded events precisely is also severely affected by non-deterministic factors present in the modern exchange systems. To promote market fairness and reduce the space for manipulation, many exchange systems introduce random uncrossing times for auctions and circuit breakers [18], green rooms [19], etc.

Inconsistent data replay is not really a problem for issue reproduction. Test automation tools enable one to repeat the sequence a reasonable number of times to trigger the required failure mode. The situation appears to be different when one tries to apply replay to the whole trading operational day. Even 99% percent replay stability for a single scenario means a certain deviation in case of hundreds of thousands of transactions per instrument. Wrong event sequencing can have a profound effect on the order book status and behavior, a.k.a. phase transition. For example, it can easily trigger a volatility interruption that will last for several minutes [20].

In course of the work, a few approaches were used to decrease the possibility of phase transitions and bring the test replay closer to original recordings:

a) Tweaking exchange systems parameters responsible for circuit-breakers and other macro changes in behavior, e.g. increase price boundaries;

b) Using inbound market data to change the prices of outbound messages;

c) Using risk control software to filter inbound messages that can cause volatility interuption;

d) Adding extra liquidity after submitting recorded messages to bring the order book state towards original execution pattern.

The first approach appeared to be the simplest one and, surprisingly, a very effective one in providing reasonable stability of replay over the course of the day. The latter approach is the most complex. It was implemented using Mini-Robots. It was easy to implement market data changes in both Sailfish and Mini-Robots as they are designed to use inbound information from the system to structure consequent messages. However, Sailfish replay is only applicable to less scalable systems as it has reasonably low performance. Load Injector, on the other hand, has limitless inbound capacity, but requires extra effort to have market data changes used by the feedback mechanism. Despite the fact that all these methods improve the entire day stability, they result in replaying different messages into a system that is different from the one used to perform recordings.

The authors have also analyzed the effectiveness of the full production day replay as a regression testing tool. It has been determined that any given trading day contains a small percentage of functional test libraries and scenario permutations from those used by QA teams to verify the systems. From the non-functional testing point of view, it is worth mentioning that volumes observed during the better part of the trading day are far away from the peak systems' throughput. The replay of the daily trading activity results in an inefficient usage of the scalable test system to cover a limited portion of available test scenarios. Moreover, changes in protocol specifications or market rules immediately make available recordings invalid for precise data replay.

II. Conclusion

Tests executed in scope of described work confirm that one hundred percent log replication is extremely difficult via external gateways even for the simple market structures and asset types. The following is required to ensure precise event replication:

a) Synchronized inbound data feeds;

b) Control over events sequencing within a distributed trading system, the ability to re-order events across gateways and internal components;

c) The ability to replace physical timing with logical timing;

d) The ability to intervene at original time scale;

e) Control over non-deterministic nature of the trading system.

Clearly, such instrumentation introduced into low-latency system's core will dramatically modify its behavior. Additional studies of the reference systems with access to their source code and architecture are required to quantify the impact of instrumentation. A more precise data replay might be useful to reproduce the failures. However, the authors believe that attempts to replicate the full trading day bring an unsatisfactory return on investment.

References

[1] The Government Office for Science foresight: *The Future of Computer Trading in Financial Markets, 2012* Final Project Report http://www.bis.gov.uk/assets/foresight/docs/computer-trading/12-1086-future-of-computer- trading-in-financial-markets-report.pdf

[2] K. Balck, O. Grinchtein, J. Pearson, *Model-based protocol log generation for testing a telecommunication test harness using CLP*. DATE '14: Proceedings of the conference on Design, Automation & Test in Europe, European Design and Automation Association, 2014

[3] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, P. Flora, *Continuous validation of load test suites*. ICPE '14: Proceedings of the 5th ACM/SPEC international conference on Performance engineering, 2014

[4] R. Ramler, W. Putschögl, D. Winkler, *Automated testing of industrial automation software: practical receipts and lessons learned.* MoSEMInA: Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation, 2014

[5] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, P. M. Chen, *Multi-stage replay with crosscut.* VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 2014

[6] S. Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, I. Neamtiu, *Deterministic Replay based Cyclic Debugging with Dynamic*. CGO '14: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014

[7] N. Viennot, S. Nair, J. Nieh, *Transparent mutable replay for multicore debugging and patch validation*. ASPLOS '13: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, 2013

[8] A Cinnober white paper on: Latency October 2009 update http://www.cinnober.com/sites/default/files/news/Latency_revisited.pdf

[9] I. Itkin, N. Pryadkina, A. Kryukov, *Data Analysis in Highload Trading Systems*, AIST Conference: Analysis of images, social networks, and texts, 2013 http://clubqa.ru/blogs/?p=436, AIST-2013

[10] FIX Trading Community
http://www.fixtradingcommunity.org/

[11] ISO 20022 Universal financial industry message scheme, *About ISO 20022*
http://www.iso20022.org/about_iso20022.page

[12] D. Guriev, M. Gai, I. Itkin, A. Terentiev, *High Performance Load Generator for Automated Trading Systems Testing*, Tools & Methods of Program Analysis 2013

[13] A. Matveeva, N. Antonov, I. Itkin, *The Specifics of Test Tools Used in Trading Systems Production Environments*, Tools & Methods of Program Analysis 2013

[14] M. Sherman, P. Sood, K. Wong, A. Iakovlev, N. Parashar, *Building the Book: A Full-Hardware Nasdaq Itch Ticker Plant on Solarflare's AoE FPGA Board*, May 2013 http://www.cs.columbia.edu/~sedwards/classes/2013/4840/reports/Itch.pdf, 2013

[15] A. Alexeenko, P. Protsenko, A. Matveeva, I. Itkin, D. Sharov, *Compatibility Testing of Protocol Connections of Exchange and Broker Systems Clients*, Tools & Methods of Program Analysis 2013

[16] SEC Release No. 34-51808, *Regulation NMS*
http://www.sec.gov/rules/final/34-51808.pdf, 2005

[17] L. Lamport Massachusetts Computer Associates Inc., *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM,issue 21(7), pp. 558–565, July, 1978.
http://web.stanford.edu/class/cs240/readings/lamport.pdf

[18] U.S. Securities and Exchange Commission, *New Stock-by-Stock Circuit Breakers*
http://www.sec.gov/investor/alerts/circuitbreakers.htm

[19] E. Holley Senior Staff Writer at Banking Technology, *Banking Technology, Tradition FX platform cuts out HFT 'predators'* http://www.bankingtech.com/64201/tradition-fx-platform-cuts-out-hft-%E2%80%98predators%E2%80%99/, 2013

[20] LSEG Response to ESMA Consultation: *Guidelines on systems and controls in a highly automated trading environment for trading platforms, investment firms and competent authorities,* ESMA/2011/224 , October 2011
http://www.londonstockexchange.com/about-the-exchange/regulatory/lsegresponsetoesmaconsultationonsystemsandcontrols.pdf