Upon a Message-Oriented Trading API

Claudiu VINȚE
Opteamsys Solutions, Bucharest, Romania
claudiu.vinte@opteamsys.com

In this paper, we introduce the premises for a trading system application-programming interface (API) based on a message-oriented middleware (MOM), and present the results of our research regarding the design and the implementation of a simulation-trading system employing a service-oriented architecture (SOA) and messaging. Our research has been conducted with the aim of creating a simulation-trading platform, within the academic environment, that will provide both the foundation for future experiments with trading systems architectures, components, APIs, and the framework for research on trading strategies, trading algorithm design, and equity markets analysis tools.
Mathematics Subject Classification: 68M14 (distributed systems).
Keywords: *trading system API, straight-through processing, distributed computing, service-oriented architecture (SOA), message-oriented middleware (MOM), Java Message Service (JMS), OpenMQ*

1. A brief introduction to trading APIs

The information technology (IT) has made exchanges far more efficient in handling heavy volume in a timely fashion and at reasonable cost. Furthermore, IT enables geographically dispersed marketplaces to be more effectively consolidated [1] [2]. The strategic advantage of an electronic platform can be summarized in form of the following beneficial effects:

- support for an efficient vertical integration (from trading to settlement);
- supply the premises for a national and regional strategy (horizontal integration);
- offers the fundaments for a decentralized market access for participants;
- continuous or extended trading hours;
- better overall services for members (availability, functionality, support);
- means for an effective centralized market surveillance from the national regulatory bodies.

In nowadays trading environment, a market center strives to offer a fast and reliable access, from anywhere and anytime, aiming to achieve a fully integrated *straight-through processing* (STP). The STP desiderate means that, once an order is placed through the order routing channel, it follows a fully integrated online procedure, passing seamlessly from its entrance on the order book, to matching on the exchange, and on through to clearance and settlement [3] [4]. On IT level, STP requires at least the vertical integration of the capital market.
In this context, the trading systems employed by the exchange members face a multitude of challenges when it comes to the ability to adapt to continuous changes and improvements implemented both *upstream* (client connectivity and interface) and *downstream* (settlement and clearing).

In this context, the *application-programming interface* (API) employed within the trading system of a broker/dealer, or a market access provider, acts as a binding agent among a variety of applications that compose the system. From that perspective, the manner in which the API is designed to facilitate the interactions between applications, it determines the characteristics and the general behavior of the entire trading system.
The architectural design and API design of a trading system are intrinsically connected [5]. There have been various approaches in trading architecture design, along with the appropriate APIs.

One of the earliest approaches was the *client-server* architecture, employing TCP/IP socket-based communication between the multiple clients and the server [6] [7]. The corresponding API consists in a collection of *call-forwards* (request calls from the clients) and *call-backs* (responses or other informative data provided back to the clients by the server). Conceptually, this communication paradigm can be identified in most of the distributed computing models, the differences residing in the underlying transfer mechanism. One of the most commonly used distributed computing model today by both Java and .NET platforms is based on the concept of *remote procedure call* (RPC). Component-based architectures such as JavaBeans are built on the top of this model. RPC attempts to mimic the behavior of a system that runs in one process. When a remote procedure is invoked, the caller is blocked until the procedure completes and returns the control to the caller. This synchronized model allows the developer to view the system as if it runs in one process. Work is performed sequentially, ensuring that tasks are completed in a predefined order. The synchronized nature of RPC tightly couples the client (the software making the call) to the server (the software servicing the call), as it is shown in Figure 1. The client cannot proceed – it is blocked – until the server responds [8].

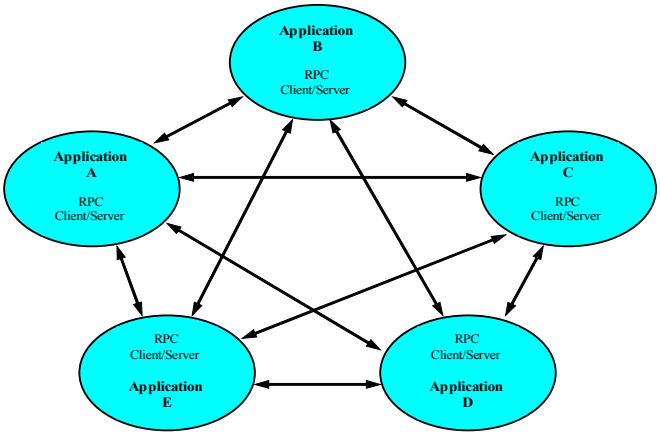

Fig. 1 – A tightly coupled RPC based architecture

On of the most successful areas of the tightly coupled RPC model has been in building 3-tier, or n-tier, applications. In this model, a presentation layer (fist tier) communicates using RPC with business logic on the middle tier (second tier), which accesses date stored on the backed (third tier). The tightly coupled nature of RPC creates highly interdependent systems, where a failure on one system has an immediate and debilitating impact on other systems. RPC works well in many scenarios, but its synchronous, tightly coupled nature is severe handicap in system-to-system processing where vertical applications are integrated together, as it is the case with a trading platform, that has to integrate client connectivity components, order processing, market execution capture, trade generation etc. In system-to-system scenarios, the lines of communications between vertical systems are many and multidirectional, as Figure 1 illustrates. When there is a new system to be added to the platform that implies a going back, and let all the other system know about it. When one part of the system goes down, everything halts. For example, when a client order is posted to the order entry system, it needs to make a synchronous call to each of the other systems. This cause the order entry system to block and wait until each system is finished processing the order. Multithreading and looser RPC mechanisms like CORBA's one-way call can be employed as options, but these solutions have their own complexities. Threads are expensive when not used wisely, and CORBA one-way calls still require application-level error handling for failure conditions. Furthermore, systems can crash, object interfaces need to be updated and, therefore, scheduled downtimes need to happen.

Summarizing, it is the synchronized, tightly coupled, interdependent nature of RPC systems that cause entire system to fail as a result of failures in subsystems. When a tightly coupled nature of RPC is not appropriate, as in system-to-system scenarios, messaging provides an alternative.

On a different approach, problems with availability of subsystems are not an issue with *message-oriented middleware* (MOM). A fundamental concept of messaging is that communication between applications is intended to be asynchronous. The API design to connect the applications together is a *one-way* message that requires no immediate response from another application. In other words, there is no blocking, or least no indefinite blocking. Once a message is sent, the messaging client can move on to other tasks; it does not have to wait for a response. This is the major difference between RPC and asynchronous messaging, and it is critical to understand the advantages offered by messaging systems.

In an asynchronous messaging trading system, each subsystem (client connectivity lines, order managing, market ordering lines, market execution capture, trade generation etc.) is decoupled from the other subsystems, as Figure 2 illustrates.

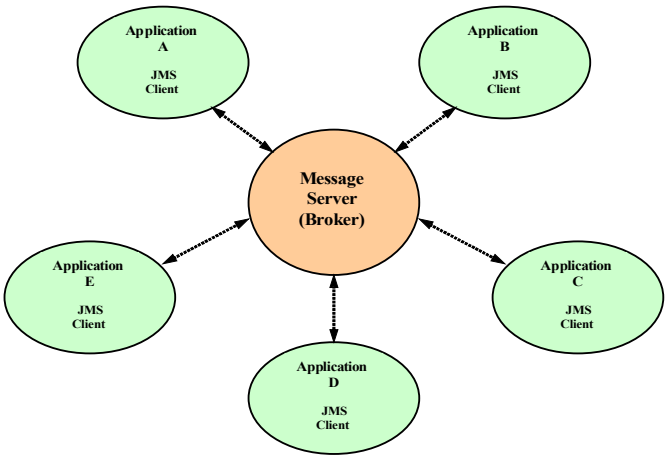

Fig. 2 – JMS provides for a loosely coupled message-oriented architecture

The applications, subsystems communicate through messaging server (message broker), so that a failure in one does not impede the operation of the others. This aspect is particularly critical in the case of a trading platform, where is imperiously necessary to offer order entry availability to the clients, and ensure that the executions results are returned to the investors as soon as they are captured from the market by the member or intermediary trading system. In a distributed computing system, partial failure is a fact. One of the subsystems may have an unpredictable failure or may need to be shut down at some time during its continuous operation. Geographic dispersion of in-house and partner trading systems can further amplify a failure situation. In recognition of this, Java Message Service (JMS) provides *guaranteed delivery*, which ensures that intended consumers will eventually receive a message, even if partial failure occurs. Guaranteed delivery uses a store-and-forward mechanism, which means that the underlying message broker will write the incoming messages out to a persistent store, it the intended consumers are not currently available, or active, from the message server perspective. When the receiving applications become available at a later time, the store-and-forward mechanism will deliver all the messages that the consumers missed while not connected to the message broker. The guaranteed delivery capability of a MOM sets it apart from an object request broker (ORB). An ORB or ORB-based middleware enables an application's objects to be distributed and shared across heterogeneous networks, but object persistence, even when this is ability is offered, it increases the complexity of the ORB and

makes for an even more accentuated dependency upon the common object libraries to be distributed and maintained across the systems [9] [10].

2. API requirements within a simulation-trading environment

In real world environment, a trading system has to combine a multitude of requirements, many of them having puling in different directions and, consequently, and equilibrium of contraries it is desired to be obtained: it has to be fast, yet flexible and adaptable; responsive, yet reliable and consistent. In order to achieve such characteristics, technically opposite in their nature, the architectural design and the API employed for the inter-application communication have to be carefully tailored to the specific needs. In our case, we have explored for an appropriate architecture and API suitable for a simulation-trading platform, within the academic environment. In this context, sheer response time of the system as whole is not an issue and, therefore, rather than focusing on the inter-application communication aspects we have opted out to research into the system functionalities and the application-programming interface that connects everything together. Therefore, the API requirements within a simulation-trading environment concern the followings:

- the simulation-trading platform should consist of the following systems at minimum:
  - trading graphical user interface (GUI);
  - order management server (OMS);
  - trade generation and portfolio management server (PMS);
  - exchange simulation engine (ESE), to act as a market place;
  - pseudo-random order generator (PROG), to enable a controlled, and desirably high, liquidity on the simulation market;
  - delayed-data feed (DDF), components to be built around web service clients, for capturing and disseminating delayed-data supplied by the Bucharest Stock Exchange (BSE) through the means of web services;
- the architecture for the trading platform has to be one of service-orientation; the component applications/subsystems need to be clearly defined functionally, and have to have the functionality exposed in the manner of a service provider;
- the communication layer has to offer support for both *point-to-point* and *publish-and-subscribe* communication models;
- the *point-to-point* communication model has to offer support for an *event-driven* architectural behavior, and for a *request/reply* type of mechanism;
- the simulation-trading platform has to be reliable when it comes to order clients, matching results, trades and portfolios handling i.e., in case of subsystems failure, there should be recovery mechanisms in place;
- the system architecture has to offer great flexibility regarding the possibility of adding new application/subsystems in the future; the API has to be design in such a way that, in the eventuality of an extension, the current functionality has not to be affected;
- the communication middleware has to provide support for data persistence and *store-and-forward* mechanism for possible assistance in recovery scenarios.

Having the above stated requirements, for a simulation-trading platform that is to be design from ground up, an API design based on a message-oriented middleware seems to be the most appropriate approach [11] [12]. We presented in [13] our earlier research upon the design of a trading system architecture based on a MOM, namely OpenMQ.
In this paper, we present the results of the subsequent level of our research, concerning the application-programming interface for the architectural trading design previously introduced.

3. Designing a service-oriented trading architecture

In the process of designing an API for a trading architecture of a service-orientation, we departed from the functionalities the systems listed above have to provide within the trading platform, and the nature of the data that is to be exchanged between them [14]. The trading API is design based on platform independent Java Message Service (JMS) interface [15] [16]. It exploits all the communication models and the mechanisms provided by JMS, for supporting the specific functionality of each system, and the way it interacts with another. For example, when an investor places a new order in the system, it is employed the asynchronous request/reply mechanism provided by JMS. The trading GUI produces and *sends* a new order message, to the destination queue **`ORDER_REQUEST_QUEUE`**, and then waits on the reply queue **`ORDER_REPLY_QUEUE`**, for a specified amount of time, to receive an acknowledgement from the order management server (OMS). The name of the reply queue is sent to the initial receiver (OMS) through the request message. We have to point out here that the asynchronous request/reply offered through the JMS interface does not block the requester processing flow indefinitely, as is the case with a synchronous, RPC-based request/reply, but for a certain amount of time, specified by the application programmer through the JMS interface. Once the client order was successfully received and processed by OMS, it is then flowed to the simulation exchange (ESE) by being sent to the destination queue **`CLIENT_ORDER_QUEUE`**. This *point-to-point* order sending, from OMS to ESE, is achieved using the *fire-and-forget* mechanism, which means that the OMS sends the client order to the specified destination, and then continues its processing flow, without waiting for any reply from ESE. This mechanism completely decouples ESE from OMS. However, the client order status can be captured back by OMS, in a similar asynchronous fashion, by receiving the messages sent from ESE to the destination queue **`MARKET_ORDER_QUEUE`**. The below Figure 3 illustrates these flows, in a normal trading operation scenario.

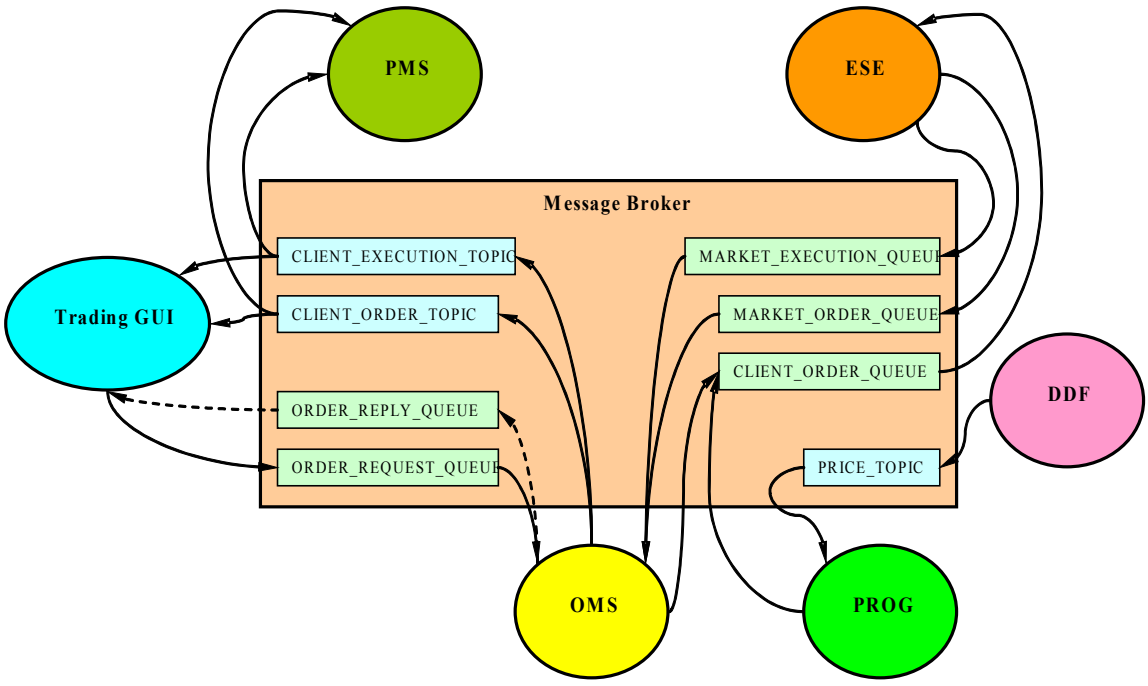

Fig. 3 – Client order and market execution flows in normal operation scenario

The dashed lines depict the reply message flow. If the client order is matched on the simulation market, then ESE generates an execution message, which is sent asynchronously

to the destination **MARKET_EXECUTION_QUEUE**.
OMS listens to both destinations **MARKET_ORDER_QUEUE** and **MARKET_EXECUTION_QUEUE**, which are fed asynchronously by ESE, and then publishes the order status updates and the market generated executions to the **CLIENT_ORDER_TOPIC** and **CLIENT_EXECUTION_TOPIC**, respectively. The applications that subscribe to these topics, trading GUIs and portfolio management server (PMS), will have to filter the published messages in order to process only the messages intended to them. In particular, PMS will subscribe for all client order updates and market executions, in order to generate the corresponding trades and maintain client portfolios. It is worth mentioning here, that PMS subscriptions to afore mentioned topics are realized in a *durable* way. That allows PMS to receive all the messages published to those topics, regardless of it maintaining continuously an active connection to the message broker.
On a different flow, the delayed-data feed (DDF) publishes, for the applications interested in it, real market data updates captured from Bucharest Stock Exchange (BSE), via a collection of web service clients. Figure 3 shows the pseudo-random order generator (PROG) as subscriber to the **PRICE_TOPIC**. Based on the price updates received from this topic, PROG is designed to generate new orders and send them to ESE, for enabling a controlled, and desirably high, liquidity on the simulation market.
The trading GUI is offered as a web browser accessible Java applet and, consequently, its communication with the messaging platform is achieved through a Java servlet responsible with the HTTP tunneling. The intention and the format of this paper do not afford us the necessary space to go into all the details of this message-oriented trading API. For further references, can be consulted our website: www.iem.ase.ro. There are numerous message flows, which cover various levels of communication between applications, for supporting multiple layers of system business logic. For example, the initialization phase of the trading GUI, may involve the acquirement of the list of tradable financial instruments for the given trading day. Trading GUI achieves the list from OMS, through the asynchronous request/reply mechanism supplied by JMS. GUI sends a request to the destination **INSTRUMENT_LIST_REQUEST_QUEUE** and then waits for OMS reply on destination **INSTRUMENT_LIST_REPLY_QUEUE**, as Figure 4 shows.

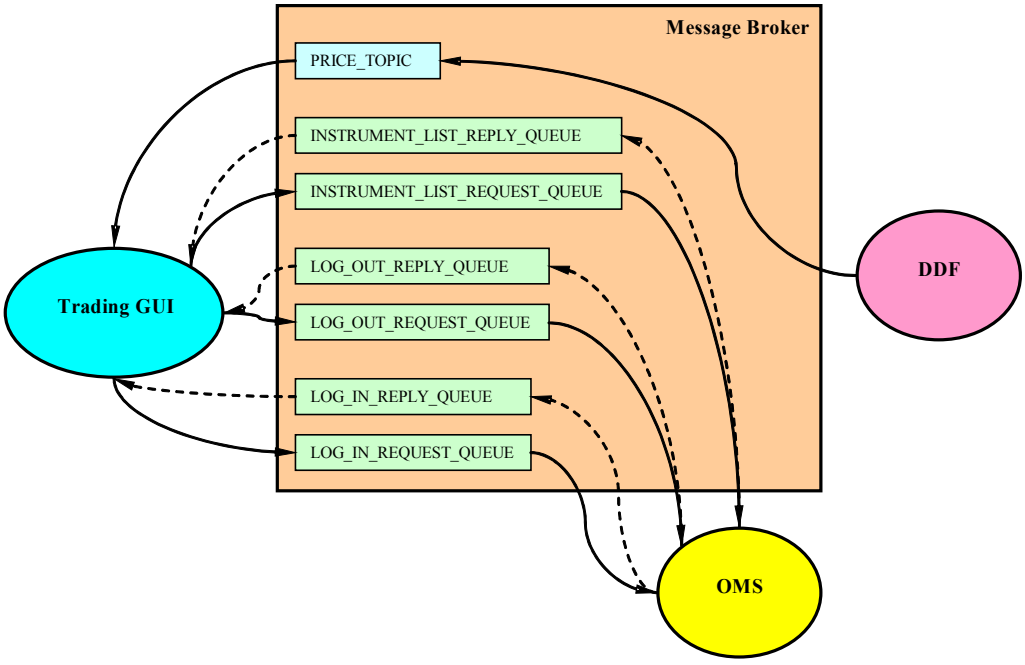

Fig. 4 – Logging in and out, along with GUI initialization flows

Employing the same JMS communication model, the logging in and logging out procedures are achieved from the trading GUI with respect to the corresponding involvement of OMS.
The trading GUI may also subscribe to the **PRICE_TOPIC**, if the user wants to consult the price data available from the market, along with daily volume, number of transactions etc. for a specified symbol.
All the messages flowed through the trading system are marked to be persistent and, therefore, to be stored by the message broker until the intended destination acknowledge their consumption. Once a persistent message is received and acknowledged by the messaging platform, its delivery to the intended destination is guaranteed. The message server stores out on disk every message marked as persistent, providing a guaranteed delivery of the message to the destination even in the case of server failure.
In line with JMS requirements, the messages are to be autonomous and self-contained entities. Each message of the trading API contains only the relevant data for a potential consumer to process it. A message does not carry imbedded instructions regarding the way in which it has to be process by the consumer. However, a JMS message may have defined certain application-specific properties, in the form of a list of pairs:
<property name><property value>.
The application-specific properties may be used for message filtering, *event-driven* processing, or for letting the consumer know about the nature of the message and, therefore, the possible ways of processing it. Departing from **Message** interface, JMS defines five more types of messages that can be handled by the JMS message server, namely: **TextMessage**, **ObjectMessage**, **BytesMessage**, **StreamMessage**, and **MapMessage** [17].
The **Message** interfaces are defined according to the kind of payload they are designed to carry. In some cases, **Message** types were included in JMS to support legacy payloads that are common and useful, which is the case with the **TextMessage**, **BytesMessage**, and **StreamMessage** message types. In other cases, the **Message** types were defined to facilitate emerging needs; for example **ObjectMessage** can transport serializable Java objects.
Being a new design from the ground up, our current implementation of the proposed trading API makes use of the **ObjectMessage** interface, since the whole trading system is based on the Java platform. However, it may be easily converted to a more open approach, which would employ the **MapMessage** interface. This interface allows for defining the payload as list of pairs *<key><value>*. This approach would open the road for a *self-defined* API: each data field in the system has a uniquely assigned identification label (key). Hence, each message payload may be composed of a subset of the generically defined collection of available date fields in the system. Such an implementation would require additional layers in the applications, for accomplishing the necessary *marshaling* and *de-marshaling* activities, in order to convert the data encapsulated in a Java object to the **MapMessage** type of payload, and vice versa.

4. Messaging as an agile and reliable approach for a trading API

Benefiting from the guaranteed message delivery supported by the JMS server, along with the *store-and-forward mechanism* offered to the durable subscribers, the trading API that we propose does not need to address aspects related to data persistence in each subsystem of the trading platform. In fact, the order management server (OMS) and the portfolio management server (PMS) are the only subsystems that are designed to interact with a database. It is the business logic of the trading platform, which commands for a persistent storage of the order books, market executions and client portfolios. In addition to that, the trading API has to provide reliable procedures for system recovery in case of a partial failure. For example, by subscribing in a durable manner to **CLIENT_ORDER_TOPIC** and **CLIENT_EXECUTION_TOPIC**, PMS ensures that if it goes down unexpectedly, all the messages published to those topics,

while it has lost the connection to the message server, will be delivered to it once the connection is reestablished. Similarly, the message broker will store the client order messages, placed by OMS onto the **`CLIENT_ORDER_QUEUE`**, until the exchange simulation engine (ESE) consumes them. However, there are restart/recovery procedures that need to be addressed programmatically, and the trading API has to support them. For example, a trading GUI may normally connect to and disconnect from the trading system for multiple times during a trading session. The investor would need to have recovered and shown in the GUI the entire trading activity that he or she has done during the current trading day. In order to achieve this desiderate transparently to the user, the trading GUI has to actively request from OMS the list of the orders that the investor has placed into the order book during the current trading session, and the list of the market executions associated to the possibly matched orders. These flows make use of the asynchronous request/reply mechanism, as Figure 5 illustrates.

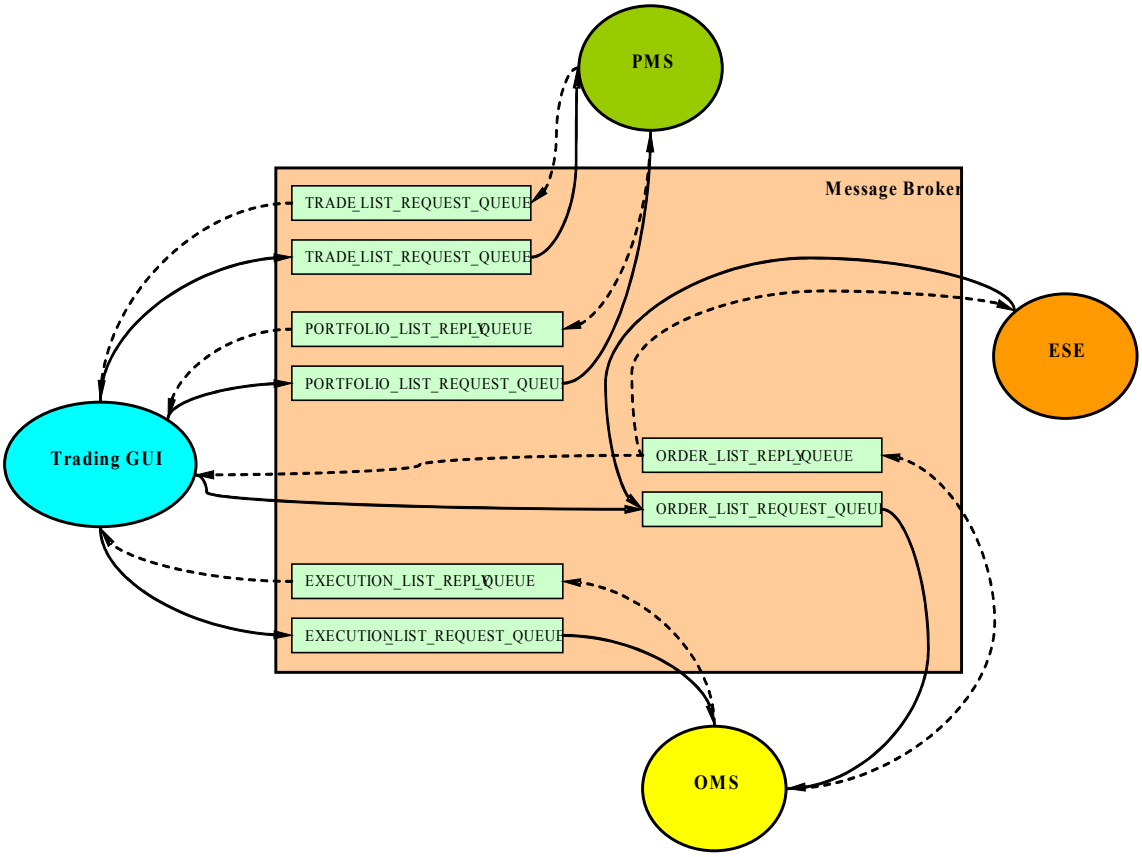

Fig. 5 – Flows concerning trading GUI and ESE restart/recovery scenarios

In addition to procedures described above, there may be requests from the investor for consulting his or her history of completed transactions, and the current situation of the portfolio of owned financial instruments.

In case of an ESE failure, being a simulation-trading environment, the recovery procedure implies an active request to the OMS for all the client orders sent to the market during the current trading session, and which are not totally executed. The exchange simulation engine is designed to be very responsive and, in order to achieve that, it keeps all the data in memory, and does not waste time in persisting any data on disk.

Summarizing, messaging is a very effective means of building the abstraction layer within SOA, needed to fully abstract a business service (functionality) from its underlying

implementation. Through business messaging, the business service (say, the order booking) does not need to be concerned about where the corresponding implementation service is, what language it is written in, what platform it is deployed in, or even the name of the implementation service. Messaging also provides the scalability needed within a SOA environment, and also provides a robust level of monitoring and control of requests coming into and out of an enterprise service bus (ESB). For example, in our implementation of the trading API, it was not important how many OMS instances might be brought up and kept running at the same time. Scalability, in the context of messaging systems, is achieved by introducing multiple message receivers that can process different messages concurrently. As messages stack up waiting to be processed, the number of messages in the queue, or what is otherwise known as the *queue depth*, starts to increase. As the queue depth increases (as client order requests may accumulate in the `ORDER_REQUEST_QUEUE`, for example) system response time increases and throughput decreases. One way to increase the scalability of a system is to add multiple message listeners to the queue to process more requests concurrently. This can be easily done dynamically, if the API is designed to use message queues that handle homogenous type of messages. Consequently, in the design of our trading API we carefully ensured that each specified destination handles a particular type of payload.
The use of messaging, as part of the overall service-oriented trading solution, allows for greater architectural flexibility and agility. These qualities are achieved through the use of abstraction and decoupling. With messaging, subsystems, components, and even services can be abstracted to the point where they can be replaced with little or no knowledge by the client components. Architectural agility is the ability to respond quickly to constantly changing environment. By using messaging to abstract and decouple components, the trading API that we have proposed in this paper, can quickly respond to changes in software, hardware and even business logic. Our intention was to design a trading API, which can be adapted with ease to the academic needs for future researches on trading strategies, design of trading algorithms, and equity markets analysis tools.

5. Conclusions

As part of our undergoing research, directed to the overall design and implementation of a simulation-trading platform within an academic, the trading API proposed in this paper intrinsically determines the characteristics of the system as a whole.
With the presented API, the architecture of the trading system that we intend to build within the ASETS project (an abbreviation from the Romanian version of the Trading System of The Bucharest Academy of Economic Studies), is currently contoured. In a simulation-trading environment, human agents compete on resources created by computer algorithms, within a scenario-driven market place. The components that create these scenarios have to *sense* the trading patterns of the human investors, and act accordingly. Designing a trading API based on a message-oriented middleware provides the optimum balance, with regards to the overall system response, availability, reliability, and flexibility in accepting future changes and extensions.
The ability to swap out one system for another, change a technology platform, or even change a vendor solution without affecting the client applications can be achieved through abstraction using messaging. Through messaging, the message producer, or client component (from the perspective of the message server), does not need to know which programming language or platform the receiving component is written in, where the component or service is located, what the component or service implementation name is, or even the protocol used to access that component or service (as we have seen with the HTTP tunneling, for web accessible trading GUI). It is by means of these levels of abstraction that enable for replacing the components and subsystems more easily, thereby increasing architectural agility.

References:

[1] McIntyre Hal (editor) - *How the U.S. Securities Industry Works - Updated and Expanded in 2004* - The Summit Group Press, New York, 2004
[2] Schwartz A. Robert, Francioni Reto - *Equity Markets in Action (The Fundamentals of* Liquidity, Market Structure & Trading) - John Wiley & Sons, Inc., 2004
[3] McIntyre Hal (editor) – *Straight Through Processing* - The Summit Group Publishing, Inc., New York, 2004
[4] Harris Larry – *Trading and Exchanges* – Oxford University Press, Oxford, 2003
[5] Vințe Claudiu - *The Informatics of the Equity Markets - A Collaborative Approach* – in Informatica Economică vol. 13, no. 2/2009, INFOREC, Bucharest, 2009
[6] Stevens W. Richard - *UNIX Network Programming* – Vol. 1, Networking APIs: Sockets and XTI, Second Edition, Prentice Hall, 1998
[7] Tanenbaum S. Andrew - *Computer Networks* - Fourth Edition, Vrije Universiteit Amsterdam, The Netherlands, Pearson Education Inc., Prentice Hall PTR, New Jersey, 2003
[8] Tanenbaum S. Andrew, Maarten van Steen - *Distributed Systems - Principles and Paradigm* - Vrije Universiteit Amsterdam, The Netherlands, Prentice Hall, New Jersey, 2002
[9] Vințe Claudiu - *Aspecte ale Proiectării unui Order Request Broker (ORB) - Partea I* - in Informatica Economică, Vol. V, No. 2 (18)/2001, INFOREC, Bucharest, 2001
[10] Vințe Claudiu - *Aspecte ale Proiectării unui Order Request Broker (ORB) - Partea a II-a* - in Informatica Economică, Vol. V, No. 3 (19)/2001, INFOREC, Bucharest, 2001
[11] Kerievsky Joshua - *Refactoring to Patterns* - Addison-Wesley, Boston, 2005
[12] Mattson G. Timothy, Sanders A. Beverly, Massingill L. Berna - *Patterns for Parallel Programming* - Addison-Wesley, Boston, 2005
[13] Vințe Claudiu - *Upon a Trading System Architecture based on OpenMQ Middleware* – in Open Source Scientific Journal, Vol.1, no.1, 2009 - http://www.opensourcejournal.ro/
[14] Erl Thomas (with additional contributors) - *SOA Design Patterns* – Prentice Hall by SOA Systems Inc., New Jersey, 2009
[15] Sun Microsystems, Inc. – *Java Message Service* - http://java.sun.com/products/jms/
[16] Sun Microsystems, Inc. - *Open Message Queue: Open Source Java Message Service (JMS)* - https://mq.dev.java.net/
[17] Richards Mark, Monson-Haefel Richard, Chappell A. David - *Java Message Service (Second Edition)* – O'Reilly Media Inc., Sebastopol, California, 2009

Claudiu VINȚE has over twelve years experience in the design and implementation of software for equity trading systems and automatic trade processing. He is currently CEO and co-founder of Opteamsys Solutions, a software provider in the field of securities trading technology and equity markets analysis tools. Previously he was for over six years with Goldman Sachs in Tokyo, Japan, as Senior Analyst Developer in the Trading Technology Department. Claudiu graduated in 1994 The Faculty of Cybernetics, Statistics and Economic Informatics, Department of Economic Informatics, within The Bucharest Academy of Economic Studies. He holds a PhD in Economics from The Bucharest Academy of Economic Studies. Claudiu has also given lectures and coordinated the course and seminars upon *The Informatics of the Equity Markets*, within the Master's program organized by the Department of Economic Informatics. His domains of interest and research include combinatorial algorithms, middleware components, and web technologies for equity markets analysis.