Upon a Trading System Architecture based on OpenMQ Middleware

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

Abstract: *In this paper we shall discuss how an open source product, namely OpenMQ middleware, can be employed as a reliable intercommunication platform for a trading system, and what impact it has on the architecture of such a system. Our design and implementation concern an academic simulation-trading environment, intended to provide both a platform for future experiments with trading systems architectures, components and applications, and a framework for research on trading strategies, trading algorithm designs and equity markets analysis tools.*

Keywords: *trading system architecture, distributed computing, open source sollutions, Message Oriented Middleware (MOM), OpenMQ, Java Message Service (JMS).*

1. Some preliminary considerations upon trading system designs

An electronic trading system resides normally within a brokerage house, and has to offer essentially the following three indispensable components/functionalities (see Figure 1):

1. a client side trading application, commonly implemented by the means of a Graphical User Interface (GUI), through which the investor may be able to place buy/sell orders and receive markets executions along with market prince data and portfolio updates;
2. an Order Management Server (OMS), which is the hearth of the entire system, meant to deal with the clients' orders, send them to the stock exchange, and process the received execution data from the market, and then flow it back to the clients [1];
3. a battery of dedicated communication lines that connect the brokerage house to a stock exchange, or multiple stock exchanges and various market sections within them.

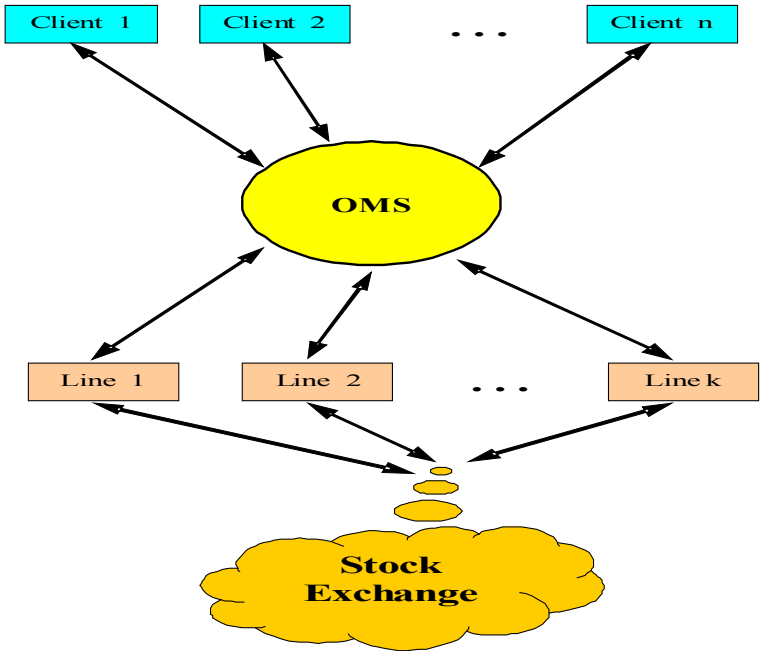

Fig. 1. The generic, simplified architecture of a trading system

The necessity of these major components, and the way they are to interact with each other, determined, organically, the architectural approach for an electronic trading system [2] [3].

Obviously, a straightforward design is to provide a client-server architecture, very similar to the one depicted in Figure 1, in which case the client applications are directly, and tightly connected to the OMS through TCP sockets, handling messages defined by a proprietary API [4][5]. Although it may deliver a very good response time for the overall system, the shortcomings of this approach come precisely from its simplicity: the tightly coupled components are all impacted, anytime a change has to be done for accommodating a new requirement, either on client side, or on serve side - the common communication API being also affected. The lack of flexibility in developing and maintaining such a system motivated, as in many other domains that rely on distributed computing, the search for more efficient approaches, concerning the communication between the various components of a distributed the system, particularly on a heterogeneous environment, when it comes to the hardware and operating systems . The most cost effective way to integrate heterogeneous components is not to recreate them as homogeneous elements, but to provide a layer that allows them to communicate despite their differences. This layer, called *middleware*, allows software applications that have been developed independently and that run on different network platforms to interact with one another [6]. Conceptually, the middleware resides between the application layer and the platform layer (the operating system and the underlying network services). Now, a middleware implementation may fall into one of the following categories:

- Remote Procedure Call, or RPC-based middleware; the approach implements a linking mechanism that locates remote procedures and makes them transparently available to the caller, as if they were accessed through local calls; historically, this type of middleware handled procedure-based programs, but now it may include object-based elements;
- Object Request Broker, or ORB-based middleware; this implementations enables an application's objects to be distributed and shared across heterogeneous networks;
- Message Oriented Middleware, or MOM-based middleware; through this approach, distributed applications can communicate, and exchange data by sending and receiving messages.

Although all these models support various levels of interaction between applications over a network, the main difference between them resides precisely in degree of interdependency among the components of a system that have to be in place, in order for the system to behave unitary. With other words, if the RPC-based middleware tend to create tightly coupled components, whereas ORB-based, and particularly MOM-based systems allow for a looser coupling of components. We presented in an earlier paper our implementation of an ORB-based middleware, namely XQuark, and the trading system architecture built based on it [7][8].

In Figure 2, we synthesize this design, evidencing that the components of the system:

- are not anymore directly interconnected, allowing for a dynamic scaling of the system - depending on the actual load, and processing capacity of each component, more OMS, and market lines may be added to the system for coping with needs;
- the communication layer is separated from the business logic of the applications, and isolated in the network daemon XQuark;
- the applications are to subscribe to set of distributed and shared objects they are interest in, and the XQuark ORB provides the delivery mechanism of these objects to the consumer applications, from the application which produce (publish) them.

This design provides for a flexible architecture, but there is one major disadvantage: the distributed and shared objects handled by the XQuark middleware are not persisted. If a XQ node goes down unexpectedly, due to a hardware or software failure, the components connected to the system through that node will miss the flow of objects made available to then by the publishing application within the system. Even if system may consist, at any given moment, of multiple instances of the critical consumers and service providers, the fact that an application may be deprived of the messages intended to it, without any means in place for recovering the missed messages, considerably diminishes the reliability of the middleware [9].

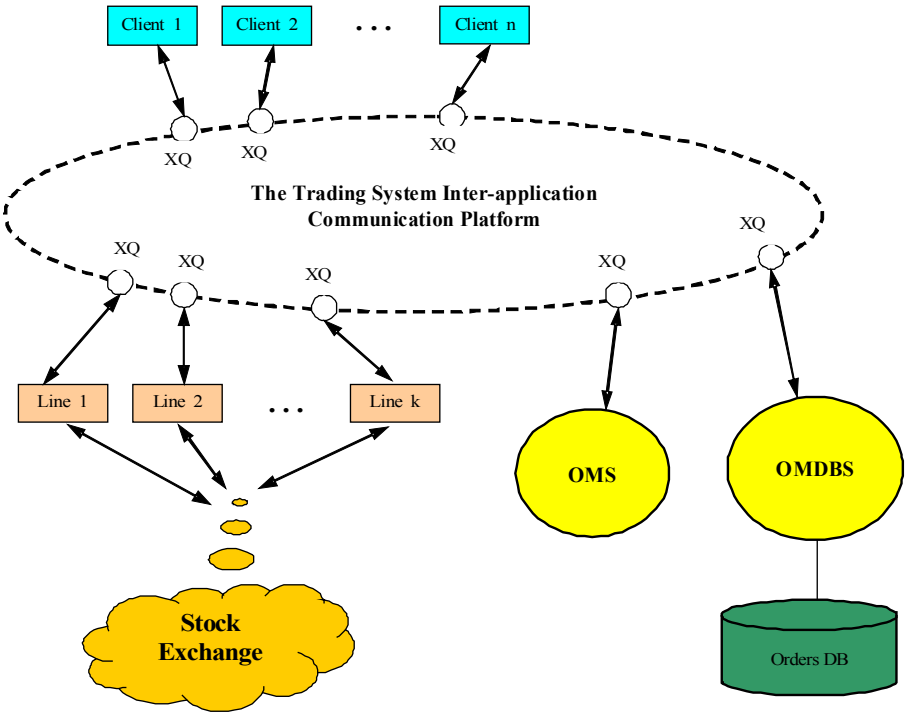

Fig. 2. The architecture of a trading system based on XQuark (XQ) ORB platform

2. Why employing an open source messaging solution? JMS as a MOM standard

Compared to an ORB-based middleware, MOM-based systems allow communication to be accomplished through asynchronous exchange of messages. A Message Oriented Middleware makes use of a messaging provider (broker) to mediate the messaging operations. In this parading, the elements of a MOM-based system are client applications, messages, and MOM messaging provider. Under the broad umbrella of client applications, could be in fact identified applications that play functionally the role of a client, and others that have the functional role of a server. All the system applications are perceived as clients of the MOM messaging provider. Using a MOM system, a client makes an API call to send a message to a destination managed by the provider. The call invokes provider services to route and deliver the message to the consumer. Once it has sent the message, the producer can continue processing flow, relying on the fact the the messaging provider retains the message until a receiving client retrieves it. In this manner, the MOM-based model, in connection with the messaging provider, makes it possible to create a system of loosely coupled components. Such a system can continue to function reliably, without downtime, even when individual

components or connections fail. The client applications are consequently effectively relieved of every communication issue, except that of sending, receiving and processing messages. Through an administrative tool coupled with the messaging provider the user can monitor and tune the performance.

With a MOM-based system there also disadvantages, and the main one results from the loose coupling itself. With a RPC-base system, the calling function does not return until the called function has finished its task. In an asynchronous system, the calling client can continue to load work upon the recipient until the resources needed to handle this work are depleted and the called component fails. Normally, the conditions in which such a situation may occur can be minimized, or avoided by monitoring performance, and adjusting message flow, i.e. the type of work that is not required with a RPC-based system. Therefore, it is important to take into account the advantages and the liabilities of each kind of system. Sometimes RPC-based components can be combined with a MOM-based system in order to obtained the desired functionality, as we shall show below.

A more serious problem with MOM-based system may arise from the fact that they are commonly implemented as proprietary products, such as TIBCO Rendezvous. The migration from one proprietary MOM to another can be as painful as it was the adoption of a MOM in the first place. To resolve this problem, a standard messaging interface is required. Then application developed to run on one system could also run on the other. Such an interface should be simple to learn but provide enough features to support sophisticated messaging applications. Introduced in 1998, the Java Message Service (JMS) aimed to cover precisely this aspect. JMS specification was originally developed to allow Java applications to access existing MOM systems. Since its introduction, many existing MOM vendors have adopted it, and it has been implemented as an asynchronous messaging system in its own right [10].

JMS specification captured, from its conception, the essential elements of the existing messaging systems, namely:

- the concept of a messaging provider that routes and deliver messages;
- distinct messaging patterns, or domains such point-to-point messaging and publish/subscribe messaging;
- facilities for synchronous and asynchronous message receipt;
- support for reliable message delivery;
- common message formats such as text, byte and stream.

All the above-mentioned elements have equally constituted the reasons why we turned to Open Message Queue (OpenMQ), as the open source MOM implementation of JMS, for designing a trading system architecture based on it [11].

3. Business logic integration through OpenMQ messaging infrastructure

Departing from the above presented considerations, the proposed architecture for a simulation trading system based on OpenMQ middleware is illustrated in Figure 3. In this architecture, the Trading GUI is implemented as a Java application meant to run within a web browser, and to access the simulation market place through a HTTP server. This approach is meant for providing a readily available trading environment to the students within an academic campus. It is important to point out from the very beginning that the diagram does not show in detail the actual message flows among the components of the system, but merely the logical connections between these components. It is also important to specify that the

messaging domains employed by the components differ, depending of the functional role played by each component within the system.

The messages produced by the Trading GUI, and which represent processing requests, are to be consumed sequentially by the Order Management Server (OMS), or by the Portfolio Management Server (PMS), and thus, in this direction the communication pattern, or messaging domain, is point-point-point messaging. The *sender* of the message (Trading GUI) produces messages that are stored in a queue by the messaging provider (OpenMQ Broker). Then the *receivers*, OMS or PMS in our case, consume these messages from the queue, in the order in which they were sent, normally FIFO (first in, first out).

The same messaging domain (point-to-point messaging) governs the communication between the OMS and the Stock Exchange Simulation Engine, in both directions this time. This is paramount, since a real stock exchange processes investors' orders sequentially; the actual time at which an order reaches the stock exchange is in fact a matching criterion for the matching algorithm. A simulation-matching engine has to perform in the same fashion, thus the point-to-point domain is natural here.

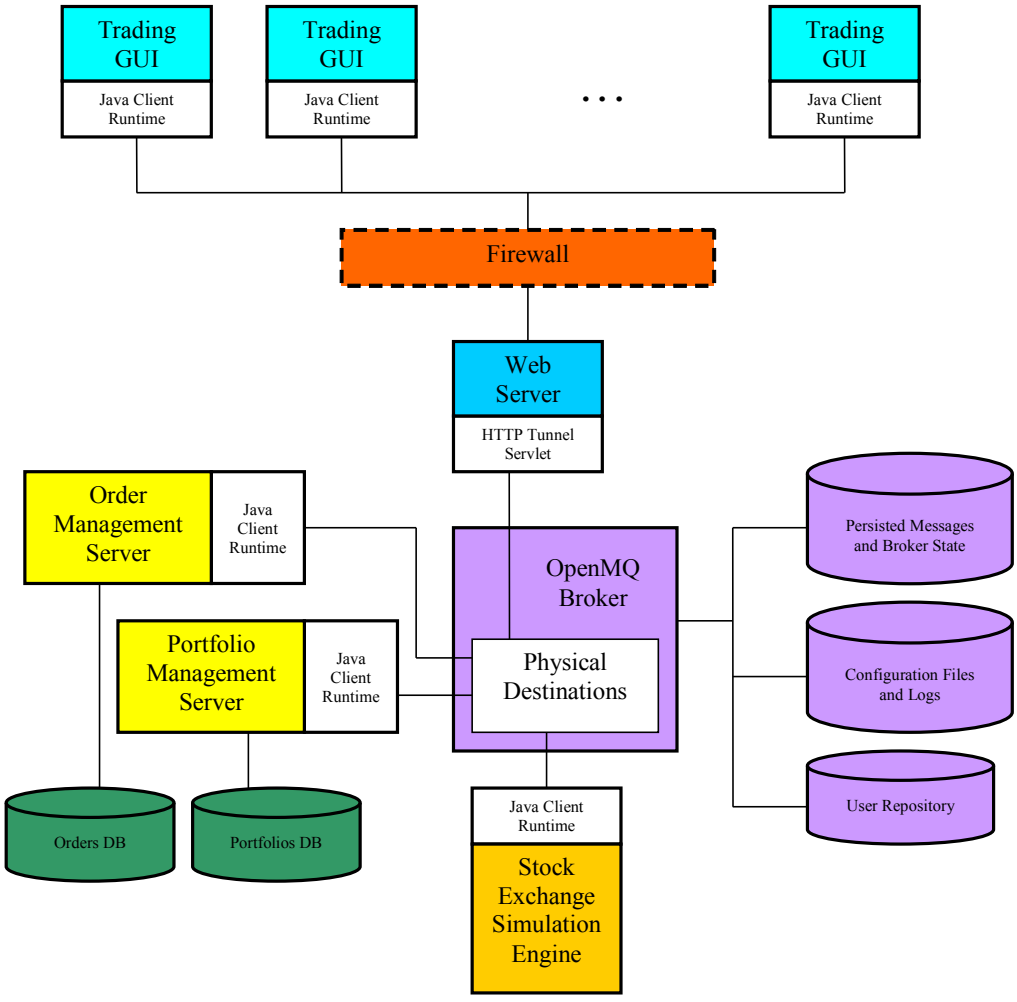

Fig. 3. The proposed architecture of the trading system based on OpenMQ infrastructure

When it comes to the messages produced by OMS and PMS, and which are meant to reach back to the Trading GUIs, there it is employed the publish/subscribe messaging. In the publish/subscribe domain, the message producers, in our case OMS and PMS, are called *publishers*, and the message consumers are called *subscribers*. They exchange messages by

the means of a destination called a *topic*: publishers produce messages to a topic; subscribers *subscribe* to a topic and *consume* messages from that topic [12]. The Trading GUI, acting as a *subscriber* for the messages received from the OMS or PMS (via messaging provider, and tunneled through by the *servlet* running within HTTP server) has to have also implemented a selector, in order to filter out messages that are logically intended to be consumed by it. That is achieved by having attached a client ID to each *call-forward* message, coming from a Trading GUI. Then OMS and PMS reuse that client ID, by inserting it into the *call- back* message that is published for being consumed by the Trading GUIs. Each Trading GUI filters out the messages with a different client ID than its own. The client IDs are managed by the OpenMQ broker.

Client runtime support is provided in libraries that a linked when building OpenMQ clients. When a a client code makes an API call to send a message, code in these libraries is invoked that packages the message bits appropriately for the protocol that will be used to relay the message to the physical destination on the broker.

Services that provide JMS support and allow clients to connect to the broker (jms, ssljms, http, https) have a service type NORMAL, and are layered on the top of TCP, TLS, HTTP or HTTPS protocols.

Services that allow administrators to connect to the broker (admin, ssladmin) have a service type ADMIN, and are layered on the top of TCP and TLS protocols.

The business logic encapsulated in the components of the system can be largely preserved and integrated within the architecture based on OpenMQ middleware. By identifying the input data (the messages that are to be consumed), and the output data (the messages that are produces) for each component of the trading system, independently of how the components interact one with another, there were made also improvements by reducing the overall amount of data that had to be passed between the components of the system.

4. The future of open source software and the academic environment

The open source concept of building reliable software solutions, and the community dedicated to support this model, have changed dramatically the process of software development, over the past decade particularly. Many open source project originated from university campuses and, perhaps, now this adventurous and, turned out to be, fruitful trail has to go back to its roots. Having access to the actual code of the software and thus, being able to get actively involved in its development, changed greatly the perception of the generic software, from being a tradable commodity, to becoming more an intellectual achievement that can be shared by everyone.

From the commercial usage point of view, the lack of a contractual form of technical support may be sometimes daunting. One the other hand, the open source community has proved for many times that it may actually react more promptly than a commercial entity, in addressing issues related to certain open source software, when reported by a large number of users.

The academic environment instead, may have only benefits from using open source software, and participating effectively in its development. There is no better way of conducting research projects in informatics, than departing from reading source code written by dedicated enthusiasts.

References

[1] H. McIntyre (editor), *How the U.S. Securities Industry Works - Updated and Expanded in 2004)*, The Summit Group Press, New York, 2004.
[2] R. A. Schwartz, R. Francioni, *Equity Markets in Action (The Fundamentals of Liquidity, Market Structure & Trading)*, John Wiley & Sons Inc., 2004.
[3] H. McIntyre (editor), *Straight Through Processing,* The Summit Group Publishing Inc., New York, 2004.
[4] R. W. Stevens, "UNIX Network Programming," *Networking APIs: Sockets and XTI, Second Edition*, Vol. 1, Prentice Hall, 1998.
[5] A. S. Tanenbaum, *Computer Networks* - Fourth Edition, Vrije Universiteit Amsterdam, The Netherlands, Pearson Education Inc., Prentice Hall PTR, New Jersey, 2003.
[6] A. S. Tanenbaum, S. van Maarten, *Distributed Systems - Principles and Paradigm*, Vrije Universiteit Amsterdam, The Netherlands, Prentice Hall, New Jersey, 2002.
[7] C. Vințe, "Aspecte ale Proiectării unui Order Request Broker (ORB) - Partea I," *Informatica Economică*, Vol. V, No. 2 (18)/2001, INFOREC, Bucharest, 2001.
[8] C. Vințe, "Aspecte ale Proiectării unui Order Request Broker (ORB) - Partea a II-a," *Informatica Economică*, Vol. V, No. 3 (19)/2001, INFOREC, Bucharest, 2001.
[9] C. Vințe, "Sisteme distribuite de asistare a tranzacțiilor bursiere – Doctorate Thesis," *Library of The Bucharest Academy of Economic Studies*, Bucharest, 2006.
[10] Sun Microsystems, Inc. – *Java Message Service* - http://java.sun.com/products/jms/
[11] Sun Microsystems, Inc. - *Open Message Queue: Open Source Java Message Service (JMS)* - https://mq.dev.java.net/
[12] Sun Microsystems, Inc. – Sun *Java System Message Queue 4.1* – Technical Overview, Part No: 819-7759, September 2007

Author

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 founder of Opteamsys Solutions, a software provider in the field of securities trading technology and equity markets analysis tools. The firm is committed to the employment of reliable solutions supplied by the open source community in all aspects of the software development process: from the operating system platform (Linux), through the IDE (Eclipse), to the data base server (MySQL), HTTP server (Apache Tomcat), and the middleware layer (OpenMQ/JMS) for inter-application communication. Previously he was for six years with Goldman Sachs in Tokyo, Japan, as Senior Analyst Developer in the Trading Technology Department. In March 2006, Goldman Sachs acknowledged the importance of the integer allocation algorithm created by Claudiu, and filed in his name a patent application for *Methods and apparatus for optimizing the distribution of trading executions* with the US Patent Office (USPTO Application 20060224495). Claudiu graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 1994, Department of Economic Informatics, within The Bucharest Academy of Economic Studies. He holds a PhD in Economics from The Bucharest Academy of Economic Studies. His domains of interest and research include combinatorial algorithms, middleware components, and web technologies for equity markets analysis.