LEVERAGING SERVER SIDE MESSAGING THROUGH WEBSOCKET-BASED FRONT END

Claudiu VINŢE
Bucharest University of Economic Studies
claudiu.vinte@ie.ase.ro
Alexandru LIXANDRU
Bucharest University of Economic Studies
alex.lixandru@gmail.com

Abstract. *This paper briefly presents our proposal for a communication mechanism between a web-based front end application and a server side collection of distributed services, glued together through a message oriented middleware (MoM). Our ongoing research has been focused upon disseminating the server side functionality, provided by our academic trading system ASETS, to a broader range of OS platforms for the front end client applications. This particular need for reachability has grown along with the increased popularity of the OS platforms designed for smart mobile devices. As long as the mobile OS platform offers the prerequisites of accommodating a web browser that supports HTML5 web sockets. Being a SOA implementation, ASETS trading platform consists, on the server side, in a coherent collection of services interconnected through a proprietary message oriented API based on JMS. One the specific particularities of a trading system is the relatively intense flow of data, generated asynchronously on the server side, that has to be pushed to the front end, without the existence of an explicit request from the client. In this context, the web sockets come to offer a viable solution for exposing the JMS publisher/subscriber communication model to a web-based front end client.*

Keywords: JMS, Messaging, MoM, Trading Architectures, Web Sockets
JEL classification: G23 - Financial Instruments; Institutional Investors; **G24** - Investment Banking; Brokerage.
Mathematics Subject Classification: 68M14 (distributed systems).

1. Introduction

A normal scenario for accessing a web-based resource implies that the web browser sends a HTTP request to the web server that hosts the resource. The web server responds by sending back an acknowledgment to this request. It is increasingly frequent the situation in which, for instance for stock prices updates, trading execution confirmations, news reports, medical device readings, and so on, that the response could contain obsolete data by the time the browser renders the page. In order to get the most up-to-date information, the user would have to constantly refresh that page manually, but this is neither efficient, nor a user friendly solution.

The traditional work around for this problem has been to provide the web-based applications with polling mechanisms, and to simulate of server side push technologies. The most notable of the latter it is Comet, which delays the completion of an HTTP response to deliver messages to the client. Comet-based push is generally implemented in JavaScript and uses connection strategies such as long-polling or streaming. [A].

The polling technique was the first attempt from the browser side for obtaining near rea-time information: the browser sends HTTP requests at regular intervals, expecting an immediate

response from the server. This is an acceptable solution if the exact interval of message delivery is known, and consequently the client request can be synchronized to be sent only when information is available on the server. However, real-time data is often not generated at regular time intervals and, inevitably, there are made unnecessary requests. The result is that, in low-message-rate situations, many connections are opened and closed needlessly.
With long-polling, the browser sends a request to the server and the server keeps the connection open for a set period, or timeout period. If new data is generated on the server-side, then a notification is sent within that period as response to the client. If a notification is not available for the client within the set time period, the server sends a response to terminate the open request. It is important to point out that, in cases that involve high message volume generated frequently, the long-polling technique does not provide any substantial performance improvements over traditional polling, but it can be even worse.
Connecting the above briefly described scenarios with server side architecture designed as service oriented one, and implemented upon a message oriented middleware (MoM), then the manner in which the server updates and fresh data are generated is fundamentally disconnected from any client's perspective of synchronization.
On the server side, fresh and useful data form the client application viewpoint, can be generated any time and a polling approach would not be able to properly mitigate the heterogeneous message types that a complex distributed system would need to handle.

2. HTML5 web sockets as support for server pushing

Web sockets provide a novel approach to web clients for communicating bi-directionally with servers without the overhead of HTTP protocol. The initial versions of the proved to have some security issues, and although they were embraced by most of the web browser developers, there were browsers like Opera, in which they were not enabled by default. Nevertheless, the newest versions of the protocol have overcome those issues, and currently most of the web browsers are supporting them.
The web sockets, apart from have implemented its own protocol, it also provides an API, namely the WebSockets API, which offers the web front end applications the ability to open and close connections, and to handle messages over a full duplex bi-directional communication channel between server and client [B].
This creates the premises for building faster, more scalable and more robust high performance real-time applications on the web. According to an analysis published by Kaazing Corporation, using web sockets as communication mean between web applications can reduce the size of HTTP header traffic by 500:1 to 1000:1, and reduce network latency by 3:1 [A]. When it comes to web applications that require fast, almost real-time updates and notifications, this translates into a low latency distributed software solution.
The WebSocket protocol specification emphasizes that one of the design decisions when this protocol was conceived was to ensure that both HTTP based clients and WebSocket based ones can operate on the same port [6], [7]. This is done by relying a similar three-way-handshake pattern that the HTTP protocol uses (built on TCP/IP), where the client sends out a request to connect, and if the server agrees, it will send out a response accepting the connection. In this manner, for client and server there is going to be only an upgrade from an HTTP based protocol to a WebSocket based protocol.
Our ongoing research project is targeted to expand the reach of ASETS trading platform toward the users of mobile devices that offer a web browser that supports HTML5 web sockets communication channel. ASETS trading platform features service oriented architecture, consisting in distributed software components, being implemented entirely in Java. It has a proprietary API, designed and built upon JMS messaging interface.

We have already explored and implemented a solution for exposing the message oriented functionality of the server side to a front end application dedicated to Android platform, building a solution based on GCM (Google Cloud Messaging) [1], [6].
The current research is targeting a web-based front end which can on any OS platform that offers a web browser provide support for web sockets.

3. The proposed front end WebSocket-based solution for the server side messaging

Since the JMS libraries, required for accessing the messaging interface that facilitates the communication with the message provider, are not available for every OS platform, programing language or script combination, we needed to explore a different approach in order to provide a seamless communication mechanism between a trading client and the ASETS trading platform [2], [3]. The server side of ASETS trading platform consists in a collation of services that provide the flowing functionality (Figure 1):

- investor order management (OMS - *Order Management Server*);
- investor portfolio (PMS - *Portfolio Management Server*);
- the order matching algorithm (ESE - *Exchange Simulation Engine*);
- a liquidity provider for the simulation environment (PROG - *Pseudo Random Order Generator*);
- on-line feeds for the trading simulation platform with real world pricing data, and instrument data, captured from Bucharest Stock Exchange (DDF - *Delayed Data Feed*).

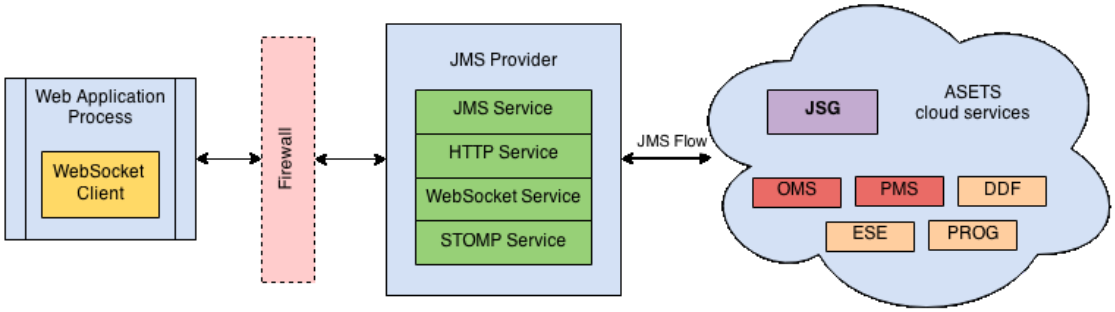

Figure 1. The overall architecture of the messaging solution via web sockets

The proprietary ASETS API, designed and implemented upon JMS, cannot be access directly from a WebSocket-based front end, since the messages handled on the server side have as payload Java objects [4], [5].
The front end solution had to be designed for being implemented in JavaScript, employing web sockets and JSON formatted STOMP 1.2 protocol messages [C], [D].
The solution that we present herein assumes the following approach on the server side:

1. enable the message broker (Open Message Queue, version 5.0.1) for supporting the WebSocket clients of type *mqjsonstomp* - any WebSocket client that sends JSON formatted STOMP 1.2 protocol to broker;
2. the creation of a new ASETS service, designed to handle the JSON formatted messages that the ASETS platform would have to interchange with the WebSocket-based front end.

The JSON (JavaScript Object Notation) formatted STOMP message do reach the messaging broker directly, having the *mqjsonstomp* setup in place, but these messages are essentially JMS messages of type TextMessage, which have as payload string of characters [2]. These messages cannot be places on the current ASETS JMS destinations (queues on the message

broker), because their payload is different than the payload created by a Java client, and will not be interpreted by the ASETS services, which expect to consume messages of type ObjectMessage or, in other words, Java objects as business payload.
The new JSG (JavaScript Gateway) service, as it is depicted in Figure 1, is a Java implemented component that has the following roles:

1. listen to newly dedicated destinations for JMS incoming messages of type TextMessage;
2. construct from the JSON formatted message the corresponding Java object(s), and place them on regular ASETS destinations;
3. listen to ASETS destinations (both queues and topics) for JMS messages of type ObjectMessage;
4. build JSON formatted messages form the Java object payloads, and place them on the dedicated destinations for JMS outgoing messages of type TextMessage.

In Figure 2 is presented the message flow that corresponds, on the server side, to the JMS communication model *request-reply*.

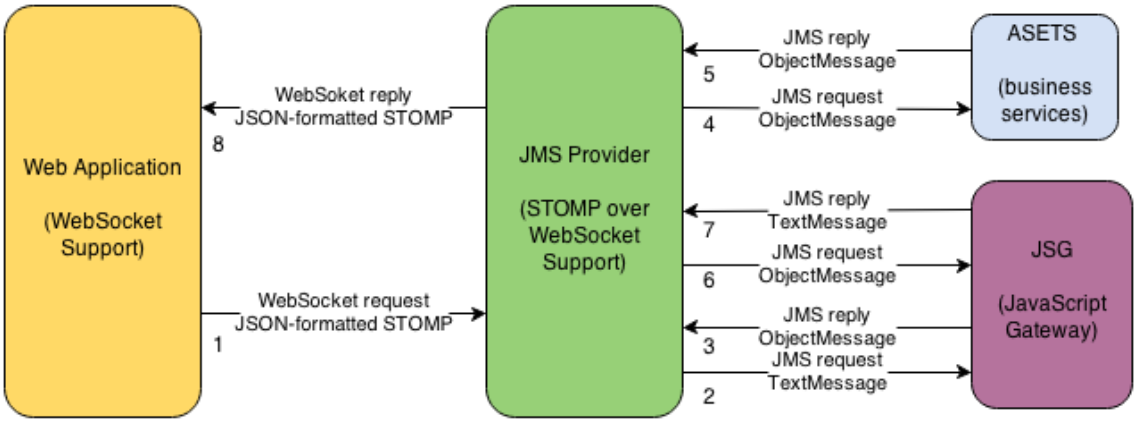

Figure 2. The message flow for request-reply communication model

Figure 3 describes the message flow that corresponds, on the server side, to the JMS communication model *publisher-subscriber*.

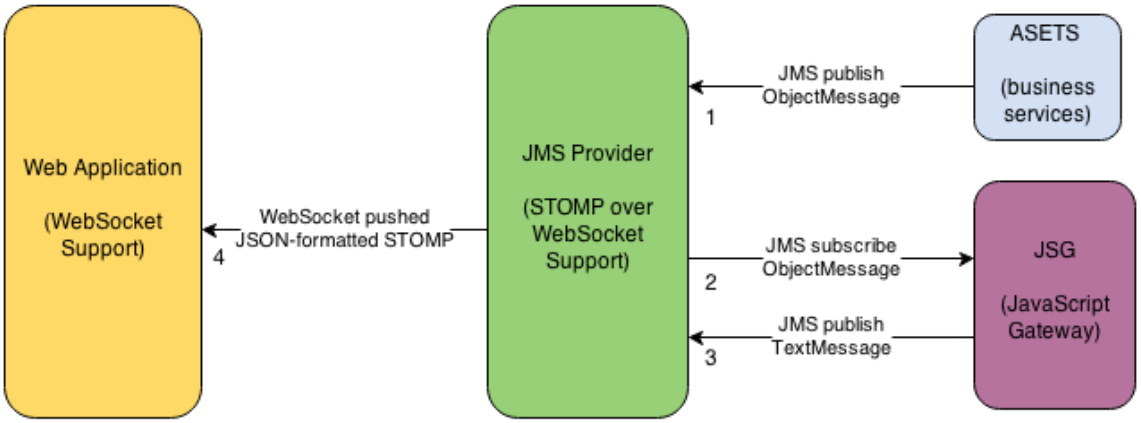

Figure 3. The message flow for publisher-subscriber communication model with message pushing through web sockets

Essentially, all the normal Java destinations (queues and topic) that handle homogeneous messages of type ObjectMessage had to be replicated. For each destination dealing with

messages having ObjectMessage payload, there was necessary to create a corresponding destination (queue or topic) that would handle TextMessage payload. The JSG creates a thread for each corresponding flow of messages, each flow being treated independently. Figure 4 shows a part of the ASETS trading platform destinations and new destinations that had to be created (prefixed with "JS_" for a uniform and consistent naming policy).

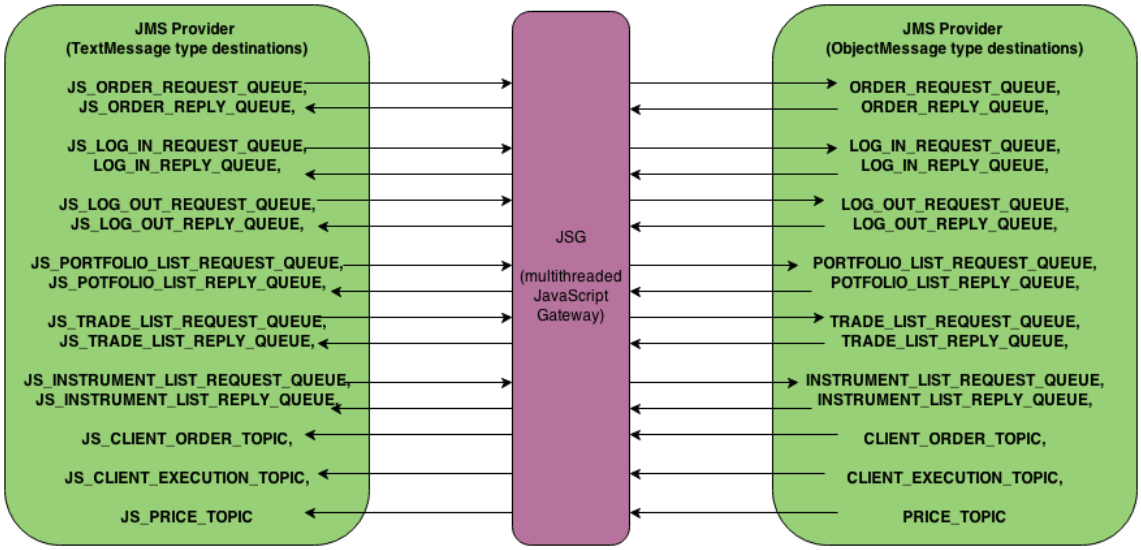

Figure 4. The JSG – JavaScript Gateway interacting with some of the ASETS destinations

4. Conclusions and further research

Our ongoing research is focused upon disseminating the server side functionality, provided by our academic trading system ASETS, to a broader range of OS platforms for the front end client applications, in a collaborative manner [8]. This paper briefly presents our proposal for a communication mechanism between a WebSocket-based front end application and a server side collection of distributed services, glued together through a message oriented middleware (MoM). As long as the OS platform offers the prerequisites of accommodating a web browser that supports HTML5 web sockets our solution delivers a rich front end trading application, as it is illustrated by the sample screen shown in Figure 5.

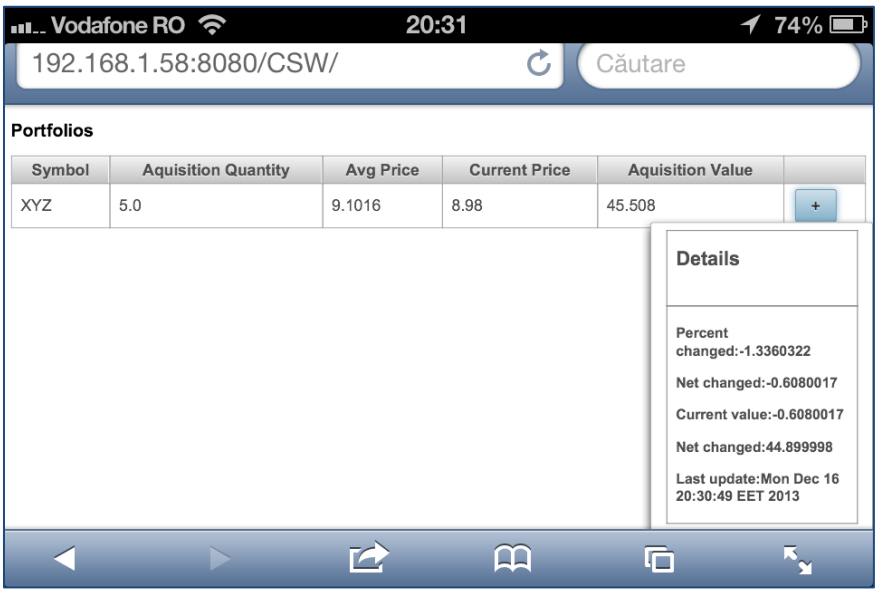

Figure 5. The portfolio panel with data details accessible at row level

The front end client is specifically designed based on the philosophy of revealing the content in successive layers, for making the graphical interface friendly with small displays.
One the specific particularities of a trading system is the relatively intense flow of data, generated asynchronously on the server side, that has to be pushed to the front end, without the existence of an explicit request from the client. In this context, the web sockets come to offer a viable solution for exposing the JMS publisher/subscriber communication model to a web-based front end client. In the context, our next direction of research within this project is to tackle the security aspects that the usage of web sockets [9], [10].

References

[1] Claudiu Vinţe, "A GCM Solution for Leveraging Server-side JMS Functionality to Android-based Trading Application", Informatica Economica Journal Vol. 17, Issue 3/2013, pp 49-59, ISSN 1453-1305, Available: http://revistaie.ase.ro/content/67/05%20-%20Vinte.pdf
[2] Claudiu Vinţe, "ASETS – An Academic Trading Simulation Platform", Informatica Economica Journal Vol. 14, Issue 2/2010, pp 97-107, ISSN 1453-1305, Available: http://revistaie.ase.ro/content/54/10%20Vinte.pdf
[3] Mark Richards, Richard Monson-Haefel, David A. Chappell, *Java Message Service (Second Edition)*, O'Reilly Media Inc., Sebastopol, California, 2009
[4] Claudiu Vinţe, "Upon a Message-Oriented Trading API", Informatica Economica Journal Vol. 14, Issue 1/2010, pp 208-216, ISSN 1453-1305, Available: http://revistaie.ase.ro/content/53/22%20Vinte.pdf
[5] Sivano Mffeis, *Professional JMS Programming*, Wrox Press 2001, pp. 515-548, Available: http://www.maffeis.com/articles/softwired/profjms_ch11.pdf
[6] Einar Vollset, Dave Ingham , Paul Ezhilchelvan, "MS on mobile ad-hoc networks", Personal Wireless Communications, PWC Conference 2003, pp. 40-52, Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.63.4140
[7] Andrew S. Tanenbaum, Maarten van Steen, *Distributed Systems - Principles and Paradigm*, Vrije Universiteit Amsterdam, The Netherlands, Prentice Hall, New Jersey, 2002, pp. 99-119, 414-488, 648-677
[8] Ivan Ion, Ciurea Cristian, Doinea Mihai, "Collaborative Virtual Organizations in Knowledge-based Economy", Informatica Economica Journal Vol. 16 Issue 1/2012, p143-154, Available: http://revistaie.ase.ro/content/61/13%20-%20Ivan.pdf
[9] Toma Cristian, Popa Marius, Boja Cătălin, "Mobile Application Security Frameworks" , Annals of the Tiberiu Popoviciu Seminar – Supplement Romanian Workshop on Mobile Business, vol. 6, 2008, Mediamira Science Publisher, Cluj-Napoca, pp. 79–93, ISSN 1584-4536
[10] Boja Cătălin, Doinea Mihai, "Security of Mobile Clients in Event-Driven Distributed Architectures", Proceedings of the 6th International Conference on Security for Information Technology and Communications (SECITC'13), 2013
A. Peter Lubbers & Frank Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web", Internet: http://www.websocket.org/quantum.html, Kaazing Corporation, 2013
B. The Web Socket API, Internet: http://www.w3.org/TR/websockets/, September 20, 2012
C. Open Message Queue 5.0.1--WebSocket Integration, Internet: https://mq.java.net/5.0.1/ws.html, March, 2014
D. STOMP Protocol Specification, Version 1.2, Internet: http://stomp.github.io/stomp-specification-1.2.html, October 22, 2012