TCP/IP XML Solution

Vlad DIACONITA PhD Candidate
Economic Information Department, Academy of Economic Studies
Bucharest, Romania

Ion LUNGU PhD
Economic Information Department, Academy of Economic Studies
Bucharest, Romania

ABSTRACT

Since its introduction, Extensible Markup Language (XML) has evolved and helped us evolve in the way which we think about structuring, describing, and exchanging information. The ways in which XML is used in the software industry are many and growing. For example, for Web services the importance of XML is crucial; all key Web service technologies are based on it. In this paper it's presented a TCP/IP XML solution for integrating an intermediary trading system with the Bucharest Stock Exchange trading system. The solution is presented from the intermediary point of view and can be the first step of a complete SOA solution.

Keywords: TCP/IP, XML, Bucharest Stock Exchange, Java

THE PURPOSE OF XML

One great thing about XML is that it is constantly changing and evolving, this means that many times it is difficult to follow. To name only a few technologies associated with xml, we can make this list: XML Namespaces, XPointer, XLink, XPath, XSLT, XQuery, RDF, SOAP, WSDL, UDDI, XAML, WSFL, WSCL, WS-I, SAX, DOM, JAXB, JAXP, JAXM, JAXR, JAX-RPC [1].

We can say that the predecessor of XML is HTML (*HyperText Markup Language)*. It's the technology that made XML necessary. This markup language packaged data and logic in a way that allowed users to view it via applications specifically designed to present it, usually web browsers .This made possible for the end users to work through data and logic remotely without too much of a problem. Of course, there was also a need for other servers, processes and applications to access and act upon data and logic stored elsewhere on a network or across the planet. This created a pursuit to find the best way of moving this data and logic from point A to point B. The main problem was that the varying sources of data were often not compatible with the platform where the data was to be served up and that's why a standardized way to structure and represent the data was needed.

The idea was to mark up a document in a manner that enabled the document to be understood now matter the working boundaries. Many systems existed to mark up documents so that other applications could easily understand them. Applying markup to a document means adding descriptive text around items contained in the document so that another application or another instance of an application can decipher the contents of the document. For example, Microsoft Word provides markup around the contents of document. As you type words into Microsoft Word, you can also provide data about the document, the so called metadata. For instance, you can specify whether a word, paragraph, or page is bolded, italicized, or underlined. You can specify the size of the text and the color. You can actually alter the data quite a bit. Word takes your instructions and applies a markup language around the data.

Like Word, XML uses markup to provide metadata around data points contained within the document to further define the data element. XML provides such an easy means of creating and presenting markup that it has not only become the most popular way to apply metadata to data, it has become the standard for

data representation [4]. In the past, one way to represent data was to place the data within a comma-, tab-, or pipe-delimited text file. For example:

10004| false |213| to ADMIN: Hello |20080827174849442

We still use these kinds of data representations. The individual pieces of data are separated by pipes, commas, tabs, or any other characters. Looking at this collection of items, it is hard to tell what the data represents. You might be able to get a better idea based on the file name, but the meaning of *213* might not be easy to deduct. On the other hand, XML relates data in a self-describing manner so that any user, technical or otherwise, can read the data. The same piece of data can be represented using XML like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
    <m: incomingEngineMessage >

<csq>10004</csq>
   <err>false</err>
   <id>213</id>
   <kmsg>to ADMIN: Hello</kmsg>
 <ktime>20080827174849442</ktime>
</m:incomingEngineMessage>
```

You can now tell, by just looking at the data in the file, what the data items mean and how they relate to one another. The data is laid out in such a simple format that is quite possible for even a non-technical person to understand the data. You can also have a computer process work with the data in an automatic fashion. XML is similar to HTML. Both markup languages are related, but HTML is used to mark up text for presentation purposes and XML is used to mark up text for data representation purposes.

THE PROBLEM

The intermediary has a RDBMS based operational system. There is an Oracle 10g database, for the data tier, an Oracle Application server for the middle tier and the clients connect to the system by using a java-enabled web browser. Before Arena Gateway was introduced it was hard for intermediary to make an automated interface between the client and the market. The client connected to the intermediary's system to place an order. The order was validated by the back-office and a line was generated in a TXT file. This file was read by an order collector which wrote the response (accepted/rejected) in another TXT file. The quotas was received in the same manner so pretty much all the interaction was done using plain ASCII files. The system had limitations, especially when working with larger sets of data, for example the intermediary didn't received a full market picture, only orders till a depths of five levels.

THE SOLUTION

Arena Gateway is an application implemented by the Bucharest Stock Exchange that acts as a message intermediary between the participant systems and the stock exchange central system. It provides request/response services; event based services as well as connectivity. Using a TCP/IP XML based messaging system it will send receive commands from the participant systems (gateway client), send them to the central system and provide responses and market data events back to the client.

The solution has two components. The first is the gateway that is deployed at the member site and will be used as a standalone application. The second is the gateway client and it is an application that should be built and that connects to the gateway, sends commands / receive responses from the central system through the gateway. The communication protocol between the gateway client and the gateway is TCP/IP XML schema based.

As for the communication protocol format, a message has the following format [2]:

|start sequence | message body | checksum end sequence |

1. The start sequence is a 9 byte sequence (0xEF 0x81 0x86 0xE2 0x86 0xA6 0xEF 0x81 0x86)
2. Message Body - the message payload; it has variable length
3. Checksum – the MD5 hash computed over the message body - 16 bytes
4. The end sequence is a 9 byte sequence (0xEF 0x81 0x85 0xE2 0x86 0xA4 0xEF 0x81 0x85)

UTF-8 charset is used to convert between bytes and character sequence representation of the message body. The message body should not contain the start or end sequences in order to prevent a misinterpretation of the real message. There are two types of messages that flow between the gateway and the client:

- outgoing messages : messages that are sent from the gateway client to the central system through the gateway
- incoming messages: messages received from the central system by the gateway client through the gateway

At application level the message body is interpreted as an XML formatted text. The XML message structure is fully described using an XML Schema file named arena-gateway-messages.xsd. Every outgoing message will be parsed and validated against the provided XML Schema. In the event of an invalidated message the gateway will send an error message to the client formatted against this specification.

Any outgoing message has a client sequence and an inner content that represents the actual command with it's parameters. An **outgoing** engine message has the following general structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<m:outgoingEngineMessage xmlns:c="http://www.bvb.ro/xml/ns/arena/gw/constraints"
xmlns:m="http://www.bvb.ro/xml/ns/arena/gw/msg" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
   <content xsi:type=…>
   …
   </content>
   <csq>10001</csq>
</m:outgoingEngineMessage>
```

OutgoingEngineMessage fields description follows:

Field name	Description
content	This is the body of the actual command that is sent to the central system. The content is detailed for every type of command in the next chapters.
csq	The client sequence can be used to identify an incoming message as being the response for this outgoing message. The central system will not modify the content of this field. Client sequence is managed by the gateway client.

Every incoming message has a header and an inner structure that embeds a command confirmation, a report, a market data event or other information. Those embedded structures are called Data Transfer Objects. An **incoming** message has the following general structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<m:incomingEngineMessage xmlns:c="http://www.bvb.ro/xml/ns/arena/gw/constraints" xmlns:m="http://www.bvb.ro/xml/ns/arena/gw/msg" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <content xsi:type=…>
      …
   </content>
   <csq>10003</csq>
   <err>false</err>
   <id>213</id>
   <kmsg>to ADMIN: Hello</kmsg>
   <ktime>20070827174849441</ktime>
</m:incomingEngineMessage>
```

The common fields of an incoming engine message are described below:

Field name	Description
Id	The type id of the message; based on this id the client knows what to expect in the content
ktime	Time (yyyyMMddHHmmssSSS); the time at witch the message was generated at the central system.
kmsg	comment, message or error description
err	error flag
csq	Client sequence, used to match an incoming message to a command. It is 0 in case of un solicited messages (like market event incoming message).
content	Contains the actual message content. It contains one or more data structures packed together. Also, it can be null. Content is detailed for every type of incoming message in the next chapters.

An incoming message can be a response to a previous command in witch case it has a client sequence equal to the client sequence of the command or can represent an event that happened on the market place.

CLIENT SIDE IMPLEMENTATION

The Gateway was meant as a modern solution for giving the intermediary the means of offering the best possible services to the end-client, the investor. For interacting with the gateway we can use a JAVA solution. At the lowest level, Java applications can read and write lexical *XML*, that is, XML represented in character form with angle-bracket markup. Reading lexical XML is called parsing; writing lexical XML is called serialization [3]. Still in this solution we use dedicated classes for interacting with the gateway and manipulating XML.

For connecting to the database we can use the JDBC drivers:

```
public void conexiune() throws SQLException
 {
```

```
String  s1 = "jdbc:oracle:thin:@10.60.0.34:1521:ORCL";

DriverManager.registerDriver(
new oracle.jdbc.driver.OracleDriver());
this.conn = DriverManager.getConnection( s1, "user_bd",parola");

this.StmtPersoane=conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
}
```

For connecting to the gateway we can use this:

```
GatewayClientApp() {

GatewayClient client = null;
String user = "XXXX";
String password = "XXXX";
int gwPort = 9000
String gwHost = "192.X.X.X";
String tradingServer = "193.X.X.X";
int tsPort = 9000;
String reporterAddress = "https:// 193.X.X.X:8443/reporter/report.rep";
try
        {
         conexiune(); //conectare la baza de date
         Statement st = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
st.executeUpdate("truncate table m_level2");
st.close();
           }
  catch ( SQLException eee ) {
System.out.println ("\n*** Java Stack Trace ***\n");
                    }
         }
```

For processing incoming messages:

```
try {
client = new GatewayClient(gwHost, gwPort, new GatewayClientListener() {

public void onMessageReceived(IncomingEngineMessage iem) {
 try
             {
 if (iem.id==802) modifica(ClassUtils.toString(iem));
if (iem.id==175){
 MboDto md=(MboDto)iem.content;
  if (md.reg!=null)
   {
java.util.List ls=md.reg;
   for(int ii=0;ii<ls.size();ii++)
      {
```

```
      OrdDto ot=null;
      ot=md.reg.get(ii);
                                        ins_ml2(ot.sym,ot.mkt,ot.sde,ot.prc,ot.siz);
         }
       }
      }

if (iem.id==800)
     {
   TickersPack tp=(TickersPack)iem.content;
  CommonTickersPack  ctp=tp.cmn;
```

For sending a command, an order for example we can use this:

```
public static OutgoingEngineMessage getAddBuyOrder(int pAcc,String pSym, String pMkt, BigDecimal
pPrc, int pSiz,String pOrd, int pGTD)
       {
 AddOrderBuyCmd buy = new AddOrderBuyCmd();
  buy.sym = pSym;
  buy.mkt = pMkt;
  buy.stt = OrdDto.ORDER_STT_STANDARD;
  buy.clr = 3;
    if (pGTD>0)
    {
    buy.trm=3;
              buy.opd=pGTD;
              }
              else
              {
                 buy.trm = OrdDto.ORDER_TRM_DAY;
              }

              buy.ref = pOrd;
              buy.acc = pAcc;
    buy.prc = pPrc;
    buy.siz = pSiz;
    buy.ver = OrdDto.ORDER_VER_NONE;
    buy.dcv = 0;
    buy.tpa = OrdDto.ORDER_TPA_ORDINARY;
    buy.tgp = new BigDecimal("0");

return new OutgoingEngineMessage(buy, GatewayClient.getNextClientSequence());
}
```

The old solution needed many pipes for connecting the intermediary's trading system to the stock exchange system. New orders and order cancelation went through a pipe, order changes went through another pipe. Incoming messages, like order confirmations and trade confirmations came through different pipe. The stock quotes came through two different pipes, one for level 1 quotes and one for level 2 quotes.

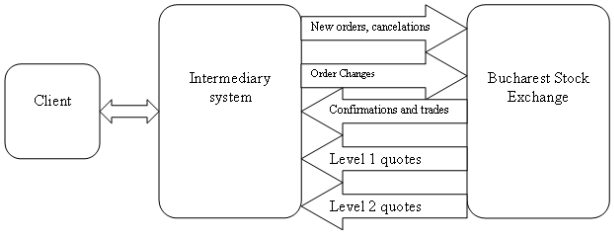

Figure 1: Integration before Arena Gateway

The internal processes had to be redesigned to suit the new integration processes.

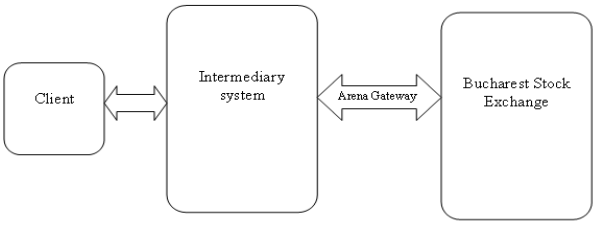

Figure 2: Integration with Arena Gateway

CONCLUSIONS

Arena Gateway, like every automation solution, regardless of platform, represents a collection of features and functions designed to execute some form of business process in support of one or more related tasks. The requirements for which such a system is built are generally well-defined and relevant at the time of construction. But, as with anything in life, they are eventually subject to change. My contribution in this project concerned remodeling the integration process between the intermediary system and the stock exchange system. This required two main tasks. The first involved developing the interface between the database and Arena Gateway, by using the Java language like shown in this article. The second task involved re-routing several internal processes without changing too much of the existing software. For now, we can say this solution delivers and delivers well; by switching from a data integration oriented solution to a process oriented one.

REFERENCES

[1] **Professional XML**, Bill Evjenet, Wrox Press 2007 (888 pages),ISBN: 9780471777779

[2] **Arena Gateway 1.1.4 Documentation**, 22.July,2008

[3] **Building Web Services with Java™: Making Sense of XML,SOAP, WSDL, and UDDI**, By Steve Graham, Simeon Simeonov, Toufic Boubez,Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama, Sams Publishing, ecember 12, 2001, ISBN : 0-672-32181-5

[4] **Understanding Web Services Xml Wsdl Soap And Uddi**, Eric Newcomer, Addison-Wesley Professional; 1 edition (May 23, 2002), ISBN 978-0201750812

[5] **Two Integration Flavors in Public Institutions**, Vlad Diaconita, Iuliana Botha, Adela Bara, Ion Lungu, Manole Velicanu, Wseas Transactions On Information Science & Applications, revised Apr. 28, 2008, ISSN: 1790-0832 (SCOPUS)