Markup Meets Middleware

Wolfgang Emmerich[1,3]
[1] Zühlke Engineering
Mergenthaler Allee 1-3
65730 Eschborn
Germany
we@acm.org

Walter Schwarz[2]
[2] Deutsche Genossenschaftsbank
Am Platz der Republik
60325 Frankfurt
Germany
walter_schwarz@dgbank.de

Anthony Finkelstein[3]
[3] Dept. of Computer Science
University College London
Gower Street, London
WC1E 6BT, UK
a.finkelstein@cs.ucl.ac.uk

Abstract

We describe a distributed system architecture that supports the integration of different front-office trading systems with middle and back-office systems, each of which have been procured from different vendors. The architecture uses a judicious combination of object-oriented middleware and markup languages. In this combination an object request broker implements reliable trade data transport. Markup languages, particularly XML, are used to address data integration problems. We show that the strengths of middleware and markup languages are complementary and discuss the benefits of deploying middleware and markup languages in a synergistic manner.

1 Introduction

An increasing number of distributed systems are not built from scratch but rather integrate legacy systems or commercial off-the-shelf (COTS) components. These components may not have been built to be integrated and are commonly heterogeneous. The heterogeneity may exhibit itself in the use of different programming languages, availability on different hardware and operating system platforms and the use of different representations for the exchange of data. We describe an example of such a heterogeneous and distributed environment in the financial domain.

We have been involved in building a new distributed system architecture for a financial trading system. In this setting, traders utilize various front-office components to input trade data as they complete transactions on the stock exchange or directly with other traders. The front-office components execute on different hardware platforms in offices in New York, Tokyo, Hong Kong, London and Frankfurt. Front-office components for the different financial products have been procured from specialized vendors. Once completed, every transaction has to be processed by middle and back-office components in the headquarters of the bank. These components perform the settlement of the transaction, analyze the risk that the bank has undertaken and monitor the performance of individual traders. Some back office components have been written in Cobol and execute on mainframes and others are purpose built using C++ on Unix machines.

Distribution middleware, such as message queues, object-oriented middleware [3] and transaction monitors can be employed to achieve reliable transfer between distributed system components.

Middleware, however, is not very good at resolving data heterogeneity. Object-oriented middleware uses common data representations for data conversions between different data formats for atomic data types (e.g. EBCDIC characters into Unicode). The middleware does not go far enough in resolving data heterogeneity. The integration of trading components demands *semantic conversions* between different data formats. For example, trades that do not involve risks should not be sent to the risk management component.

The latest generation of markup languages, most notably the eXtensible Markup Language (XML) [1] support the definition of data structures through document type definitions (DTDs). These DTDs are, in fact, grammars for special purpose markup languages. Although they were initially meant to represent structured documents on the world-wide-web, they are increasingly used as data representation mechanisms for complex structured data that needs to be communicated between distributed system components.

The main contribution of this paper is the discussion of an example of a successful combination of distribution middleware and markup languages that facilitates system inte-

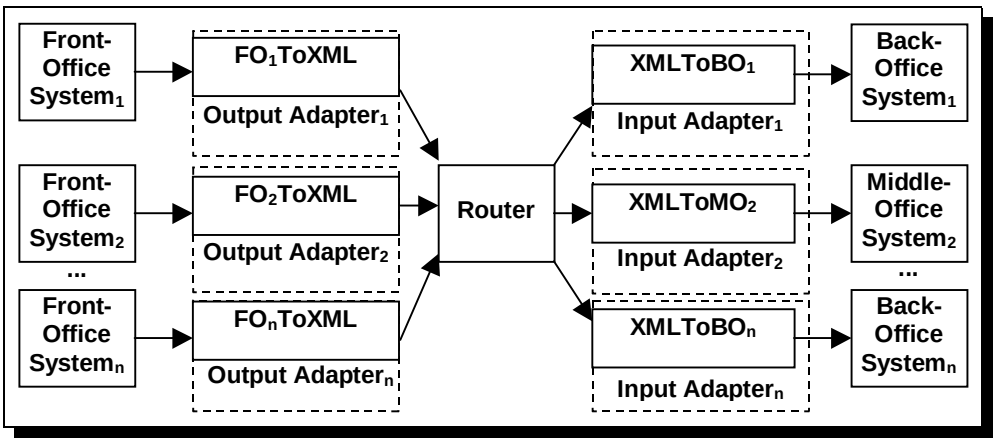

Figure 1. Overview of Trading Architecture

gration. The following Section 2 presents an overview of the trading architecture that we discuss as an example. We then indicate in Section 3 how markup languages, and in particular XML, are used in this trading architecture to resolve semantic differences between different trade data representations. We discuss in Section 4 how we use object-oriented middleware in order to control the reliable trade data transport between front, middle and back office components. We conclude by indicating research directions towards a tighter integration of markup and middleware.

2. An Overview of the Trading Architecture

The distributed trading system architecture has to meet two main requirements. It has to:

- reliably transfer trading data between the distributed system components and
- resolve the heterogeneity of the data that is produced or expected by different trading system components.

Figure 1 shows an overview of the trading architecture and the data flow between the different architectural components. The different front, middle and back office components cannot be modified, but rather have to be integrated using their legacy interfaces. Input and output adapters achieve this integration. They wrap [7] these legacy components and hide the complexities of interfacing with these components.

An essential requirement is that the trading data that originates in a front office component has to reach those middle and back office components that have to further process the trade. Trade data are usually not sent to all middle and back office components. Trades that do not involve any risk, for example do not have to be sent to the risk management component. Hence the architecture has to manage the routing of trades from front office to middle and back office components. This routing is performed by the router component.

The trading architecture meets the two main requirements shown above. The data flow is achieved by a reliable notification mechanism that uses the CORBA Notification Service. The input and output adapters that wrap existing components perform translations to and from a common trade data representation in XML. We now discuss these two aspects in more detail.

3. Semantic Translations using XML

Conversion components could be built and integrated using, for example an object-oriented middleware, such as an implementation of the CORBA standard [11]. That would, however, require modelling the complete trade data format in the OMG Interface Definition Language (IDL). The data structures of trading data are large and complex. When complex and large data structures are to be transmitted between conversion components using middleware there is a substantial development overhead because in addition to the mapping of the source data structure to the target data structure, these data structures need to be expressed in the interface definition language of the middleware. There is also a run-time performance penalty to be paid because the data structures need to be marshalled and unmarshalled. To make things worse, the data structures for trading data are far from stable. Traders develop new products (derivatives) on a regular basis. Incorporating these new products into the interface definition of the object middleware would demand interface changes on a fortnightly or monthly basis. The current CORBA standard and its implementations lack the capabilities to manage such change.

It is, we argue, impractical to express large and complex trading data structures using the interface definition

languages of middleware. High performance, low development and maintenance costs can only be achieved if the middleware does not have to interpret complex data. The trading architecture therefore uses a trade data representation in XML and trade data is transported by the object middleware in an uninterpreted way. The use of XML is motivated by the availability of standards for financial trading data and by the evolving tool support. Moreover, vendors of front office components are starting to provide XML-based interfaces to their components, which will further simplify future integration.

The architecture defines a common trade data representation. The representation has been developed starting from international financial standards, most notably the Network Trade Model (NTM) [5] that will be incorporated into the Financial Products Markup Language (FpML) [4]. The NTM trade data representations have then been adjusted so that bank-specific products can be represented. Figure 2 shows an excerpt of the document type definitions for the model. The fragment shown is used to define the data structures that need to be exchanged for bond transactions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Bond(IssueID,SettlementDate,BuyOrSell,
          PriceOrYield, Principal, Accrual)>
<!ELEMENT IssueID (#PCDATA)>
<!ELEMENT SettlementDate (#PCDATA)>
<!ATTLIST SettlementDate datatype CDATA #FIXED "Date">
<!ELEMENT BuyOrSell EMPTY>
<!ATTLIST BuyOrSell value (Buy|Sell) #REQUIRED>
<!ELEMENT Principal (#PCDATA)>
<!ATTLIST Principal datatype CDATA #FIXED "Float">
<!ELEMENT PriceOrYield EMPTY>
<!ATTLIST PriceOrYield value (Price|Yield) #REQUIRED>
<!ELEMENT Accrual (Cashflow)>
<!ELEMENT Cashflow(CashflowID,CashflowPayment,
          (FixedCashflow|FloatingCashflow|
          ComplexFloatingCashflow)?,Notes?,
          Extensions?)>
<!ATTLIST Cashflow
          CFLType (FixedInterest|FloatingInterest|
          ComplexFloatingInterest|Principal|
          Fee|Premium|Commission|Tax|Other) #REQUIRED>
```

Figure 2. DTD for a Bond Trade

Figure 3 shows the data representation for a very simple bond trade, which is an instance of the DTD in Figure 2.

The architecture has to implement mappings between the proprietary formats that front, middle and back office components produce or expect and the standardized XML based format as shown above. These mappings are implemented in a mapping service, which is called from within the output and input adapters shown in Figure 1.

Some of the front office components have a message-based interface and emit messages in proprietary formats. These messages do not have XML markup tags, mostly because the front office components were built before XML was defined. Consequently they cannot be parsed by an XML parser. Instead, the architecture uses a specialized

```
<?xml version="1.0"?>
<!DOCTYPE BondTrade SYSTEM "NTM.dtd">
<Bond>
 <IssueID> Bundesrepublik Deutschland</IssueID>
 <SettlementDate> 26.05.1999 </SettlementDate>
 <BuyOrSell value="Buy"/>
 <PriceOrYield value="Yield"/>
 <Principal>Wolfgang Emmerich</Principal>
 <Accrual>
  <Cashflow CFLType="FixedInterest">
   <FixedCashflow>
    <Rate>5.25%</Rate>
    <Period>6m</Period>
   </FixedCashflow>
  </Cashflow>
 </Accrual>
</Bond>
```

Figure 3. A Bond Trade in XML

mapping tool [6]. This mapping tool supports the definition of different message formats and rule based mappings between them.

The architecture could use the eXtensible Stylesheet Language (XSL) [2] to translate marked-up trade information into the representations that the back-office components expect.

XSL includes a rule-based language that can specify how source tree elements are translated into target elements. It supports projection (Omitting tree elements), traversing trees in a particular order and the like. The XSL programming support and processors that are currently available, however, are not yet stable and sophisticated enough to warrant mission critical use. Instead, we use the mapping tool that generates the translation for the output adapters also for mapping XML representations to middle and back office representations.

XML has originally been defined as the next generation Web Markup Language. Hence, it was initially assumed that XML data is distributed using the HTTP protocol. The HTTP protocol is however very inflexible as it supports only point-to-point connections and only put and get operations between them. Also there are no reliability guarantees for delivery of XML data over HTTP, which renders the protocol unusable for reliable system architectures, such as the one for the financial trading system. We now review how data transport can be achieved with Middleware rather than the HTTP protocol.

4. Middleware for Reliable Data Transport

There are many different middleware approaches, such as message queues, object request brokers and transaction monitors. Most of them can be employed to achieve reliable transfer between distributed system components. Message queues buffer messages for temporarily unavailable system components. Object-oriented middleware, such as OMG/CORBA implementations, Java/RMI or Microsoft's

COM, transmit structured data within operation parameters and notify requesters if failures occur. Transaction monitors use the two-phase commit protocol to achieve consensus between distributed components on the success of a transaction.

This diversity of available middleware approaches and the even bigger number of vendors offering middleware products leads to a selection problem. Our approach to selecting middleware for this trading architecture was requirements-driven.

During the early stages of the trading architecture development process, non-functional requirements were identified. These included scalability requirements, availability requirements, reliability requirements, security requirements and maintainability requirements. Scalability required that the architecture has to be able to process up to 100,000 transactions per day. Security required that trade information must not be eavesdropped and that trade information should only be passed between system components within the bank. Maintainability required the introduction of new front, middle or back-office components to be reduced to a couple of months rather than several years that it currently takes. In order to determine interfacing requirements legacy mining activities were performed so as to discover the legacy interfaces that were available for the integration of input and output adapters.

We compared message-oriented middleware, transaction-oriented middleware and object-oriented middleware first analytically and then using prototypes in order to find a suitable middleware approach. Several interesting results arose from that comparison. The degree of standardization is far higher for CORBA products than for message-oriented and transaction-oriented middleware. Message-oriented and transaction-oriented middleware are more difficult to use than object-oriented middleware, mainly because of the need to "hand-code" marshalling. As a result a middleware that implements the CORBA standard was selected.

The trading architecture has to achieve a selective and reliable multicast of trade data that is represented in XML. Only a limited amount of trade data information is needed for making the selection. These data have to be represented redundantly both in the XML trade representation and in a CORBA data type, that we refer to as `Routable`. Even with the selection of CORBA as the middleware and the aim to reuse as many of the CORBAservices as possible, a number of design options remained open. These are

- use of the CORBA Event Service as basis of the Router implementation;
- use of the CORBA Messaging Service for reliable and asynchronous trade data delivery; and
- use of the CORBA Notification Service as basis of the Router implementation.

The CORBA Event Service is specified in Chapter 4 of [9] and is available for most CORBA implementations. The CORBA Event Service supports asynchronous one-way multicast of event data from one supplier to multiple receivers. Moreover, it achieves a de-coupling of event producers from event consumers. The Event service is relevant to the trading architecture, as the trade data that needs to be multicast from one front office component to multiple middle- and back-office components can be regarded as typed events. Furthermore, the architecture aims at decoupling trade data senders and receivers and that could be achieved with the Event service, too.

The Event service supports both push- and pull-type communication. The communication pattern in the trading architecture is push rather than pull. The Event service supports both typed and non-typed event communications. In the trading architecture event communication will be typed (using the `Routable` data structure) and the event types will express those parts of the trading data structures that are of concern for the routing of event data. The Event service is, however, not suitable for the trading architecture as it does not support the specification of quality of service attributes, such as reliability of data delivery. Moreover, it does not support event filtering, which is necessary to charge the service with routing of trading data.

The CORBA Messaging service is specified in [8] and supports guaranteed delivery of asynchronous object requests in CORBA. It will be incorporated into the CORBA 3.0 standard and is not yet available in any product.

Call back objects in the messaging service support asynchronous object requests. Messaging capable IDL compilers will generate these call back objects for asynchronous IDL operations. CORBA implementations are expected to invoke call back objects transparently for the application programmer when the server object finishes the request. The Messaging and Event Services have in common that they support asynchronous delivery of request parameters. They are different in that firstly, the Messaging Service supports peer-to-peer communication, while the Event Service supports multicasts, secondly the Event Service supports unidirectional communication, while the Messaging Service supports bi-directional communication, and finally the Messaging Service supports guaranteed delivery which the Event Service does not.

The Messaging service, however, is unsuitable. The time between the creation of a trade at a front-office and the back-office might well exceed several hours. It could sometimes even exceed a night. The messaging service would need to keep callback objects for all those trades in order

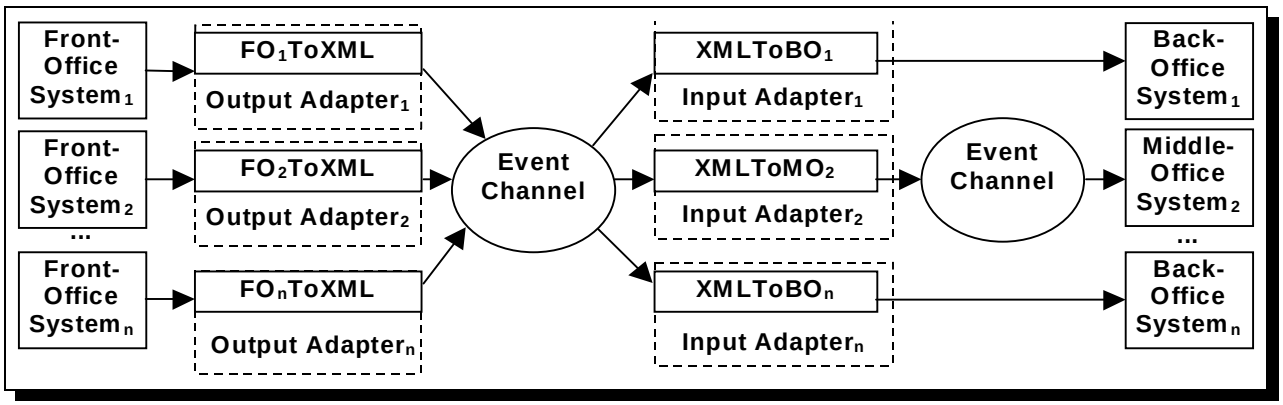

Figure 4. Use of CORBA Notification Service

to wait for acknowledgement of the receipt of the trade objects in all middle and back-office components. We would expect that there will be a substantial overhead involved in managing these callback objects in a fault-tolerant and reliable way. Moreover, there are no stable implementations of the messaging service as yet and implementing the Messaging service is beyond what can reasonably be achieved in our setting as it requires modifications of the core of an object request broker, such as the IDL compiler.

The CORBA Notification service was adopted by the OMG Telecommunication Task Force [10] and overcomes the shortcomings of the Event Service. There are various implementation of the notification service available. The Notification Service is based on the Event Service, and adds capabilities to determine reliability of event communication, event prioritisation, event expiry and event filtering. This makes the service very suitable for the implementation of trade data transport. In particular, it will be possible to treat all Output Adapters as event suppliers, all Input Adapters as event consumers and the Router as an Event Channel.

As shown in Figure 4, trade data are processed and converted by output adapters into the standardized XML representation and then passed into an event channel for distribution. The event channel knows the input adapters and applies filtering to each event so as to make sure that every event is sent to that subset of input adapters that have to receive the event. It is also shown that additional event channels may be used to further de-couple the conversion process performed by the input adapter from a receiving middle or back office component. The input adapters may also contact receiving back and middle office components without involving an event channel if the interface to receiving component already contains a queuing mechanism.

Figure 5 shows as a UML Sequence Diagram how an output adapter uses the interfaces of the Notification service. To initialize itself, it obtains a `TypedSupplierAdmin` object for `Routable` event types from a `TypedEventChannel` and it then establishes the qualities of service attribute for that channel, asking the channel to retain its connections upon failure and to guarantee delivery of event data. Whenever event data needs to be forwarded through the Notification service, the output adapter converts the data into the standard XML representation and then invokes `push_structured_events` from the `TypedProxyPushConsumer` object. This will guarantee delivery of the event to all `TypedPushConsumersObjects` that are currently registered with the event channel.

Thus, by determining persistent event and connection reliability, an implementation of the trading architecture can delegate guaranteed delivery to a Notification service implementation. By using the filtering mechanism supported by the Notification service, each input adapter can ensure that only relevant events are passed on to the middle and back office component. The Notification service supports the administration of these filters with a constraint language.

5. Lessons Learned

We learned numerous lessons during this project, the most important of which we detail below.

1. The combination of markup languages and middleware is largely successful. The use of middleware enabled us to isolate functional concerns in the mapping components. Further work will be needed by the OMG and the W3C to achieve a tighter integration. In particular it would be desirable to be able to see XML data structures through an IDL interface and vice versa. This would have allowed us to avoid encoding data redundantly in the Routable data structure.

2. Our first attempt to use XSL for the mapping of se-

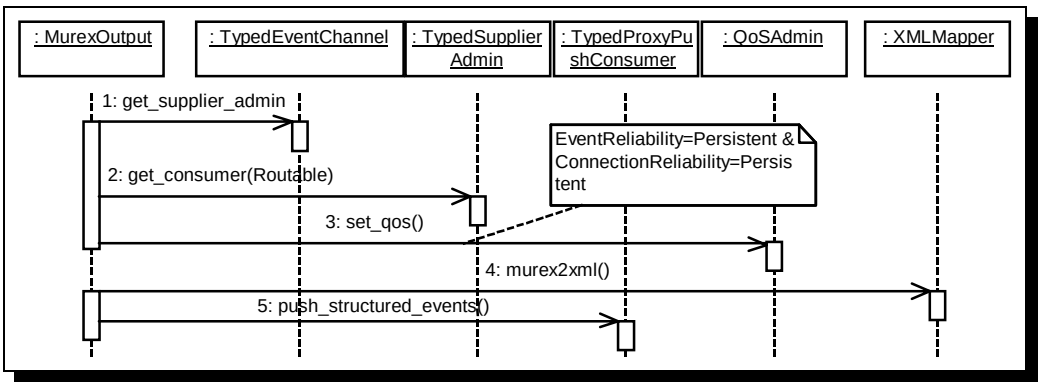

Figure 5. Output Adapter Interacting with Notification Service

mantic data conversions was seriously hampered by the lack of tool support and higher-level abstractions. The mapping of data structures needs to be done by business analysts, who understand the semantics of the different XML markups. For analysts, rule based XSL specification is too low a level of abstraction.

3. Non-functional requirements determine most of the choices during the selection and design of the architecture. The strong demand for scalability, reliability and high availability drove the development of the architecture and the selection of products that were deployed in the architecture.

4. The remaining freedom for architectural design choices were further restricted by constraints imposed through standardized components, such as the CORBA Notification service. In particular, the service demanded that its filtering mechanism has to be used for distributing event data.

6. Conclusion

The strength of middleware and markup languages are complementary. Based on the experience with this trading architecture, we would expect this combination to be used in those future distributed systems where complex data structures need to be transmitted between distributed off-the-shelf components and semantic transformations have to be performed. Such architectures will utilize middleware for achieving reliable transport of data between multiple distributed system components. They will leverage markup languages to express the structure of data so that semantic data transformations can be determined at appropriate levels of abstraction using standards and performed using off-the-shelf technology.

Acknowledgements
The architecture described in this paper was developed jointly with Jürgen Büchler, Henry Fieglein, Rolf Köhling, Harald Viel and Stefan Walther.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[2] J. Clark and S. Deach. Extensible Stylesheet Language (XSL). Technical Report http://www.w3.org/TR/1998/WD-xsl-19980818, World Wide Web Consortium, August 1998.

[3] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.

[4] FpML. Introducing FpML: A New Standard for e-commerce. http://www.fpml.org, 1999.

[5] Infinity. Infinity Network Trade Model Overview. http://www.infinity.com/ntm/pdf/ntmOverview.pdf, 1999.

[6] MINT Technologies. MINT Rule Manager. http://www.mintech.com/rule.html, 1998.

[7] T. Mowbray and R. Zahavi. *The Essential CORBA – Systems Integration Using Distributed Objects*. Wiley, 1995.

[8] OMG. CORBA Messaging – Revised Joint Submission. ftp://ftp.omg.org/pub/docs/orbos/98-03-11.pdf, MAR 1998.

[9] OMG. *CORBAservices: Common Object Services Specification, Revised Edition*. 492 Old Connecticut Path, Framingham, MA 01701, USA, December 1998.

[10] OMG. *Notification Service*. 492 Old Connecticut Path, Framingham, MA 01701, USA, January 1998.

[11] OMG. *The Common Object Request Broker: Architecture and Specification Revision 2.2*. 492 Old Connecticut Path, Framingham, MA 01701, USA, February 1998.