TIGRA– An Architectural Style for Enterprise Application Integration

Wolfgang Emmerich
Dept. of Computer Science
University College London
Gower St, London WC1E 6BT, UK
W.Emmerich@cs.ucl.ac.uk

Ernst Ellmer
Zühlke Engineering GmbH
Düsseldorfer Strasse 40a
65760 Eschborn, Germany
ee@zuehlke.com

Henry Fieglein
DG Bank
Am Platz der Republik
60325 Frankfurt, Germany
henry_fieglein@dgbank.de

Abstract
We report on experience that we made in the Trading room InteGRation Architecture project (TIGRA) at a large German bank. TIGRA developed a distributed system architecture for integrating different financial front-office trading systems with middle- and back-office applications. We generalize the experience by proposing an architectural style that can be re-used for similar enterprise application integration tasks. The TIGRA style is based on a separation of data representation using domain-specific XML languages from transport of those data with an appropriate middleware. We show how Markup languages, particularly the eXtensible Markup Language (XML) and eXtensible Stylesheet Language Transformations (XSLT), can be used to solve semantic data integration problems. We discuss that the strengths of middleware and markup languages are complementary and indicate the synergies yielded by deploying middleware and markup in the TIGRA style.

1 Introduction

An increasing number of distributed systems are not built from scratch but rather integrate existing applications or commercial off-the-shelf (COTS) applications. These applications may not have been built to be integrated; they are often procured from different vendors and they are likely to be rather 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. Heterogeneity, scalability and fault-tolerance requirements also lead to distributed deployment of applications. Yet, IT departments of large organizations are often expected to provide an integrated computing facility to their users.

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

The construction of heterogeneous distributed systems is simplified by distribution middleware systems, such as message queues, object-oriented middleware or transaction monitors [5]. These systems resolve hardware, operating system and programming language heterogeneity and provide primitives, such as message transfer, object requests and transactions that can be used for communication across distributed hosts.

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, however, go far enough in resolving semantic data heterogeneity. The integration of enterprise applications in general, and financial trading systems in particular, often demands *semantic conversions* between different data formats. Some systems, may for example, identify counter-parties (the customers of the bank) by name, while others use an account code. When a trade data representation has to be transmitted from one system to another, a semantic transformation of the counter-party identification has to take place.

The latest generation of markup language standards, most notably XML [2] 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. These structured data can then be transformed using transfor-

mations that can be specified as eXtensible Stylesheet Language Transformations [3], a standard related to XML.

The main contribution of this paper is the discussion of experience that we made in the TIGRA enterprise application integration project, where we used both middleware and markup to achieve integration of distributed and heterogeneous applications. While several interest groups have defined XML markup languages for particular domains, the novel contribution of TIGRA is the strict separation of data representation in XML from transport of those XML data with an appropriate middleware. We discuss how we developed TIGRA and aim to present our experience in a reusable way by formulating an architectural style. As part of the presentation of the architectural style, we describe the class of requirements that led to the adoption of the style in order to enable readers to identify similarities with integration problems they may face. We conclude by indicating further needs for software engineering and web engineering research.

In Section 2, we describe the need for enterprise integration that our financial institution has in common with many other organizations. We then describe our process for developing a new architecture for enterprise application integration in Section 3. In Section 4, we discuss the non-functional requirements that guided the development of TIGRA and show how markup languages, and in particular XML, are used in this trading architecture to resolve semantic differences between different trade data representations. We also discuss in that section how we use middleware in order to control the reliable trade data transport between front, middle and back office systems. In Section 5 we outline the experience that we made with the TIGRA architecture. We conclude by indicating research directions for software architectures in Section 6.

2 The Problem

Before we started the TIGRA project, our financial institution performed application integration in a rather ad-hoc manner. The IT department for trading had to implement, maintain and integrate about 120 different applications. Due to a lack of an enterprise application integration approach, most of these applications had two or more direct interfaces with other applications. There is, and probably never will be, a common trading data format that every application supports. Therefore each interface was unique, which lead to large interface development costs: developing an interface required between three and ten person years of effort and took between one and three years. Neither the development costs nor the time that was needed to integrate new applications were acceptable any longer.

Before TIGRA, interfaces were executed in overnight batches, which meant that trading data entered in a front-office system only became available in middle- and back-office systems during the next day. With stock exchanges opening longer hours and clients expecting to trade over the Internet also in foreign exchanges, the batch window that was used when no exchanges were open has disappeared. Moreover, in the future traders will expect near real-time integration with middle-office systems, such as market and currency risk management systems so that risk increase or mitigation can be factored in when quoting a security or derivative price. In order to meet these demands all interfaces would have to be changed.

Finally, the quality of trading data was a problem. Because it was expensive to automate all interfaces that were needed, less frequently used interfaces were operated manually (by clerical staff reading from one screen and typing data into a user interface of another application). This caused obvious "transmission errors" that had to be detected by periodic reconciliation of front-office and back-office data. Again due to lack of automated interfaces, reconciliation was commonly done manually by comparing print-outs.

To overcome the above difficulties provided the motivation for our financial institution to invest in a systematic enterprise application integration architecture. When discussing the project results at trade shows, we were repeatedly approached by organizations from the financial, telecommunications and transport domain that told us about very similar problems. We therefore believe that the TIGRA solution to enterprise application integration may be of wider interest to the scientific community at large.

3 Architecting Process

The TIGRA project used an incremental and iterative architecture development process in order to mitigate risks and be able to demonstrate benefits to key decision makers early on. We now discuss our experience with this process in a little more detail.

Requirements Analysis: In our experience software architectures are determined by requirements. In fact, they are often determined by global or non-functional requirements that stakeholders expect from their system. It was therefore natural to start the TIGRA project with a thorough requirements analysis exercise.

The business requirements elicitation served two purposes. The obvious one was an outline of the high-level requirements for the TIGRA project. The more subtle objective was to obtain buy-in for the architecture project from the different divisions of the bank that were affected by the architecture development. The stakeholder analysis and their subsequent involvement ensured that the stakeholders felt they had a say in how the applications that they run are going to be integrated in the future and thus they were willing to contribute to the funding of TIGRA.

The business requirements themselves are not directly operationalizable, but provided fertile ground to elaborate the system requirements. We paid particular attention on non-functional requirements, such as openness, standards-

compliance, security, scalability, availability and performance that directly influenced the shape of TIGRA.

Explorative Architecture Prototyping: The business requirements determined that TIGRA should perform as little in-house development as possible. It was deemed necessary to rely on off-the-shelf integration technologies as much as possible. The business requirements identified a need to avoid vendor tie-in and instead demanded the use of open standards so as to remain as vendor independent as possible.

These goals meant that TIGRA had to identify relevant middleware standards and their implementation. This was achieved by inviting vendor presentations and organizing reference site visits. The team then had to familiarize themselves with candidate products. We achieved this during an explorative architecture prototyping stage, where we developed prototypes that demonstrated the required goals for a simple interface between a bond trading and a settlement system. Altogether, we developed six prototypes, evaluating a transaction monitor product, object request brokers and a message-oriented middleware. Moreover, we explored the use of a proprietary message broker that supports semantic data transformation as well as the use of XML and XSLT. The development of these prototypes each took two to eight weeks.

Middleware selection: We analyzed each of these prototypes against the requirements. Some requirements (e.g. security, standards compliance, openness) were assessed analytically, while performance and scalability was analyzed by benchmarking and stress testing. The fact that prototypes were available for quantitative measurements allowed us to gain confidence in our selection. We will reason about the selection further in the next section.

Pilot development: The project then developed a pilot application of the architecture, where a bond trading system is integrated with a market risk management system, a reconciliation service and a trade settlement system. The pilot was developed in six months and is in production now. Developing the pilot proved to be invaluable for convincing other divisions in the bank about the benefits that can be derived from a systematic enterprise application integration approach.

Large scale deployment: TIGRA is now generally accepted in the bank and has become the standard any for further integration projects. The bank has planned to develop 13 interfaces in 2001, a previously insurmountable undertaking. The team that developed the style and the pilot is now acting as an internal consultancy organization that trains and mentors staff from other IT divisions to use and adopt the style. We are currently developing training material on the use of TIGRA and start-up kits to assist staff from other IT divisions to instantiate the TIGRA style.

4 The TIGRA Architectural Style

Requirements:
The TIGRA software architecture is determined by a number of non-functional requirements. We describe the requirements in some detail here as the similarity of our requirements and requirements the reader may have determine whether the reader can re-use the TIGRA style.

Scalability: TIGRA has to be *scalable*. In particular, it has to cope with the transaction load of the entire securities and derivatives trading department of our financial institution. The load is lower than in the retail sector and based on past experience, we estimated a maximum of 100,000 transactions per day for the lifetime of the architecture. The peak daily transaction load is reached when exchanges in both Europe and the US east coast are open and we estimated a peak of 10 transactions per second.

Performance: A main aim of TIGRA is to overcome the delays of batch processing. In the finance industry, this requirement is sometimes also referred to as *straight through processing* (STP). It means that trading data are exchanged whenever the trading occurs rather than only at the end of the trading day. We elicited the requirement that the elapsed real time should be below 10 seconds from when trade processing is completed by the front-office system until it has been delivered at all back-office systems.

Reliability: It is of highest importance that trade details are *reliably sent* from front-office to middle- and back-office systems. They must not be lost or otherwise modified while they are exchanged. TIGRA, therefore, has to guarantee the delivery of a trade at all intended destinations. Moreover, traders have to be able to use front-office systems to quote prices and complete trades regardless of the state of middle- and back-office system components. This means that TIGRA has to de-couple front-office systems from middle- and back-office system and avoid using blocking forms of communication between front-, middle- and back-office systems.

Availability: The components and connectors of TIGRA have to be available throughout the trading day. Moreover, some of the integrated systems are still batch-systems, which means that TIGRA also has to be able to deliver trade data over-night after trading has been completed. However, it is possible to shut down trading systems for maintenance and upgrades during bank holidays and weekends. This means that *availability* requirements are not as strict as for safety-critical systems, such as power plant controllers or in some health care applications, but yet they demand that TIGRA remains operational non-stop for at least six days a week.

Security: The financial institution operates a strict security regime with a very tightly controlled fire wall between public networks and its own private network. The TIGRA project assumes that this firewall protects security against attacks from outside the bank. However, TIGRA has to implement measures for ensuring security against attacks from

users that are authorized to use the private network. This involves three concerns. Firstly, TIGRA has to *authenticate users* and associate security credentials, such as access rights and privileges with users. Based on the security credentials, TIGRA has then to *control access* to the services that it implements. In particular, it has to be avoided that some rogue program, written by e.g. a contractor, sends false trade data to the back-office for settlement. Finally, auditors of the bank want to be notified of security relevant incidents and TIGRA therefore has to gather an *audit log* of security relevant events.

Changeability: The trading IT department is faced with constant *change*. Changes originate in, for example, new derivative contracts that are invented by financial institution on a very regular basis. From past experience, new contracts are defined at least once a month and then trading system components have to be adapted to support dealing in those products. To support this change it was found necessary that TIGRA implements and leverages both *standards* in the financial industry, but also domain-independent standards so that components can be exchanged if necessary.

Use of COTS: The financial industry heavily relies on *COTS components* that are procured from specialized vendors and prefers to buy rather than build components. TIGRA has to integrate these components and has to resolve heterogeneity and distribution. Firstly, components are executed on distributed machines. The machines are often rather heterogeneous and in our particular case, we have components executing under the Windows-NT, Solaris, VMS and OS/390 operating systems. Moreover, the way that trading data are exported and imported among these components varies significantly. Some components just define a file format from which TIGRA has to read changes, others publish a database schema and TIGRA can interface using ODBC or JDBC. Yet other components have socket-based APIs, define message formats that are to be read and written from and to message queues or even have an object request broker interface. In any case, different programming languages are needed.

Overview of TIGRA Style

Figure 1 shows an overview of the TIGRA style and the data flow between the different architectural components. Rectangles in the figure show components and arrows denote trading data that is sent from one component to another. We have to assume that the different front-, middle- and back-office systems cannot be modified, but rather have to be integrated using their heterogeneous interfaces. Input and output adapters achieve this integration. An *output adapter* obtains data from a component and converts them into a common semantic representation. An *input adapter* provides data to a component in its native representation. Thus, both input and output adapters wrap [9] existing or COTS applications and hide the complexities of interfacing with them. Adapters use data mapping components for semantic data conversion.

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

The purpose of *mapping components* is to perform semantic data conversion between the native formats that front-, middle- and back-office components support in order to resolve data heterogeneity. TIGRA defines a common semantic data representation for financial trading data and while in transit through the architecture, any trade is represented in that common representation. This reduces the need from $O(n^2)$ (with n being the number of components) to $O(n)$ mapping components.

Initially, we thought of building and integrating the mapping components using object-oriented middleware, such as an implementation of the CORBA standard [13]. That would, however, require modelling the complete trade data format in the interface definition language (IDL) of the middleware and in principle, IDLs are expressive enough for that purpose. The data structures of trading data are, however, large and complex. When complex and large data structures were to be transmitted between conversion components using middleware there would be a run-time performance penalty to be paid if the data structures needed to be marshalled and unmarshalled [4]. Because of the need for incorporating new security and derivative products, the trading data structures are not stable. When using a middleware IDL, this means that any architectural components, such as the client and stubs server stubs that are derived from the IDL would need to be changed, too. This implies that whenever one front-office system introduces a new product, every component of the architecture needs to be re-compiled and the version and configuration management demands make this impractical. The incompatibility of the different IDLs is another argument against this approach. If we used a particular IDL, say OMG/IDL we would lock the entire trade data representation into CORBA and it would become next to impossible to change the middleware to, say a message queue. Finally, the business analysts, who know how to implement semantic mappings would never be able to build mappings because they need to be implemented in a programming language.

We therefore decided that using middleware primitives to express the complex trading data structures was not a viable option. It became the *leitmotif* of the TIGRA style to separate semantic trade data integration from trade data transport as strictly as possible. TIGRA uses XML and related technologies for achieving trade data integration and it uses object middleware for achieving reliable, scalable and secure trade

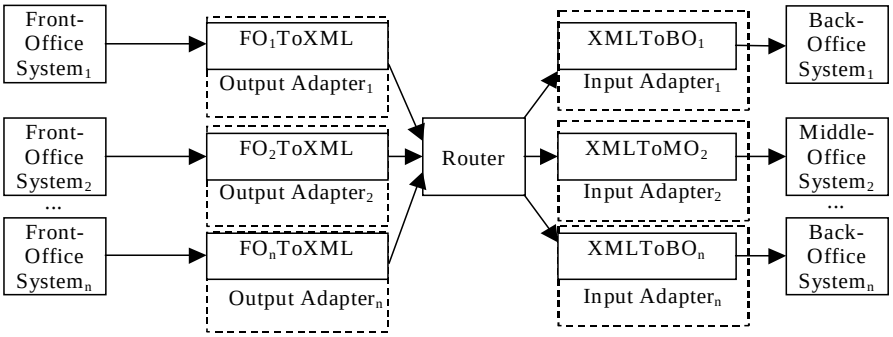

Figure 1: Overview of Trading Architecture

data transport. We, therefore, discuss these two aspects of the TIGRA architectural style separately now.

Data Integration Using XML and XSLT

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 systems are starting to provide XML based interfaces to their systems, which will further simplify future integration.

TIGRA defines a common trade data representation. The representation has been developed starting from international financial standards, most notably the XML version of the Fixprotocol (FixML) [7] and the Financial Products Markup Language (FpML) [8]. The Fixprotocol was defined to support electronic exchange of securities information and is therefore well suited for representing standard products, such as bonds, equities and options, which are traded on international exchanges. FpML has recently been proposed for derivatives that are generally not traded at an exchange but that banks trade directly "over-the-counter". Jointly the two markup languages cover the 90% of the spectrum of products traded at our financial institution. However, several customizations of the FixML standard are necessary because the Fix consortium of American investment banks did not cater for specialties and recent developments in the European market. FixML, for example, does not include a currency code for the Euro and various new Eastern European currencies. Figure 2 shows a small excerpt of the FixML DTD that we use. For reasons of clarity, the FIX protocol attribute tags are omitted.

An XML DTD defines the syntax of a markup language. `ELEMENT` definitions are similar to productions in context free grammars and define the other elements that can be included in an element. The DTD in Figure 2, for example, defines that an `Instrument` can either contain a `Security`, a `Future` or an `Option`. The `ATTLIST` definition declares the attributes that an element can have together with their types and possibly default values. It defines, for example that an `Option` element has to have an attribute `Side` that determines whether the option is a buy or sell option. Figure 3 shows an excerpt of the data representation for a very simple

```
<!ELEMENT Instrument (Security | Future | Option)>
<!ELEMENT InstrumentList (Instrument+)>
<!ELEMENT Future (Security,Maturity)>
<!ELEMENT Option (Security, Maturity, StrikePrice,
                  OptAttribute?)>
<!ATTLIST Option
  Side (Put|Call) #REQUIRED
  Cover (Covered|Uncovered) #IMPLIED
  Type (Customer|Firm) #IMPLIED
  OpenClose (Open|Close) #IMPLIED
>

<!ELEMENT Security ((Symbol|RelatedSym),SymbolSfx?,
  SecurityID?,SecurityExchange?,Issuer?,SecurityDesc?)>

<!ATTLIST Security
  Type (BA|CD|CMO|CORP|CP|CPP|CS|FHA|FHL|FN|FOR|GN|
        GOVT|IET|MF|MIO|MPO|MPP|MPT|MUNI|NONE|PS|RP|
        RVRP|SL|TD|USTB|WAR|ZOO|FUT|OPT) "CS"
>

<!ELEMENT Symbol (#PCDATA)>
<!ELEMENT RelatedSym (#PCDATA)>
<!ELEMENT SymbolSfx (#PCDATA)>
<!ELEMENT SecurityID (#PCDATA)>
<!ATTLIST SecurityID
    IDSource (1|2|3|4|5|6|7) #REQUIRED
>

<!ELEMENT SecurityExchange (#PCDATA)>
<!ELEMENT Issuer                (#PCDATA)>
<!ELEMENT SecurityDesc          (#PCDATA)>
<!ELEMENT Maturity (MonthYear,Day?)>
<!ELEMENT StrikePrice (#PCDATA)>
<!ELEMENT OptAttribute          (#PCDATA)>
```

Figure 2: Excerpt of FixML DTD for Securities Data

bond trade, which is an instance of the DTD in Figure 2.

We would now like to highlight some more general observations about using XML to structure data. The ability to define data structures by way of a DTD, or in the future by using XML schemas [6], enables us to use a general-purpose XML parser to validate the correctness of data against its type definition. This proves invaluable as data quality issues can be detected very early. We also note that the availability of markup language definitions is not restricted to the financial domain, but that there is a wealth of languages defined across different application domains. This enables the TIGRA style to be reused in different settings, too.

The architecture has to implement mappings between the

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE FIXML SYSTEM "fixmlmain.dtd" []>
<FIXML>
 <FIXMLMessage>
  <Header>
    ...
  </Header>
  <ApplicationMessage>
   <Allocation AllocTransType="0">
    <AllocID>564548</AllocID>
    <RefAllocID>0</RefAllocID>
    <Side Value="2"/>
    <Instrument>
     <Security Type="CS">
      <Symbol>WKN</Symbol>
      <SecurityID IDSource="1">352058</SecurityID>
      <Issuer/>
      <SecurityDesc>DGBK 4.96DE03EU</SecurityDesc>
     </Security>
    </Instrument>
    <Shares>1000000.00000000</Shares>
    <AvgPx>99.52000000</AvgPx>
    <Currency Value="EUR"/>
    <TradeDate>20000414</TradeDate>
    <TransactTime>11:01:11</TransactTime>
    <Settlement>
     <FutureSettlmnt>
      <FutSettDate>20000118</FutSettDate>
     </FutureSettlmnt>
    </Settlement>
    <NetMoney>1001691.80327869</NetMoney>
    <Text>mit Boni</Text>
    <AccruedInterestRate>0.64918033</AccruedInterestRate>
    <AllocationGroupList NoAllocs="1">
     <AllocationGroup>
      <AllocAccount>9802352058</AllocAccount>
      <AllocShares>1000000.00000000</AllocShares>
      <BrokerOfCredit/>
      <ClientID>KRGEN H  BG204</ClientID>
      <Commission CommType="2">0.10000000</Commission>
      <AllocNetMoney>1001691.80327869</AllocNetMoney>
      <AccruedInterestAmt/>
     </AllocationGroup>
    </AllocationGroupList>
   </Allocation>
  </ApplicationMessage>
 </FIXMLMessage>
</FIXML>
```

Figure 3: A Bond Trade in FixML

proprietary formats that front, middle and back office systems produce or expect and the standardized XML based format shown above. At the time of writing this paper, front-, middle- or back-office system do not support export or import of well-formed XML. However, in our experience it is not difficult to create application-specific DTDs in such a way that application specific formats can be transformed into a marked up representation and vice versa by very simple scripts that exchange appropriate markup tags with application-specific delimiters, such as commas in comma separated files. Thus, the first stage of any output adapter is to read the native data representation of the component and produce a marked up version in an application-specific markup language. Then we can use primitives that have been built for transforming an XML documents from one markup language into another.

TIGRA uses eXtensible Stylesheet Language Transformations (XSLT) [3] to translate application-specific markup languages into the standard FixML/FpML notation and vice versa. XSLT defines 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. Our prototyping stage found XSLT programming support to be sufficient and the available Xalan XSLT processor to be stable and sophisticated enough for mission critical use. Figure 4 shows an XSLT sample taken from the TIGRA instantiation that is currently in production.

```
<xsl:template name="insertSecurity">
<xsl:param name="DealType"/>
<xsl:param name="SecurityType"/>
<Security>
 <xsl:attribute name="Type">
  <xsl:value-of select="$SecurityType"/>
 </xsl:attribute>
 <xsl:if test="$DealType = 'SEC'">
  <xsl:call-template name="insertSecurityDetails">
    <xsl:with-param name="WKN"
     select="string(OLK_CLASS_SEC/OLK_SEC_CODE)"/>
    <xsl:with-param name="SecLabel"
     select="string(OLK_CLASS_SEC/OLK_SEC_LABEL)"/>
  </xsl:call-template>
 </xsl:if>
 <xsl:if test="starts-with($DealType,'REPO')">
  <xsl:call-template name="insertSecurityDetails">
   <xsl:with-param name="WKN"
    select="string(OLK_CLASS_REPO/OLK_REPO_SEC_CODE)"/>
   <xsl:with-param name="SecLabel"
    select="string(OLK_CLASS_REPO/OLK_REPO_SEC_LABEL)"/>
  </xsl:call-template>
 </xsl:if>
</Security>
</xsl:template>

<xsl:template name="insertSecurityDetails">
<xsl:param name="WKN"/>
<xsl:param name="SecLabel"/>
...
<SecurityID>
 <!-- IDSource has to be one of 1,2,3,4,5,6,7 -->
 <xsl:attribute name="IDSource">1</xsl:attribute>
  <xsl:value-of select="$WKN"/>
</SecurityID>
<Issuer/>
<SecurityDesc>
  <xsl:value-of select="$SecLabel$"/>
</SecurityDesc>
</xsl:template>
```

Figure 4: Transforming Trades with XSLT

The figure shows two named templates `insertSecurity` and `insertSecurityDetails`. The `insertSecurity` template creates the `Security` element of a FixML Bond Trade and then calls the `insertSecurityDetails` template. It passes two parameters `WKN` and `SecLabel`, whose values it obtains from the input tree by following different path expressions, depending on whether the security is a bond or a repo (i.e. a bond loan). The second template then inserts the `SecurityID` element, an empty `Issuer` element and a `SecurityDesc` element.

We have managed to express 80-90% of the transformation concerns in XSLT. We, however, also found various needs for using paradigms other than XSLT template-based tree transformations. These were, for example complex computations of attribute values for calculating accrued interest of a bond,

transforming one date representation into the other, or table-based mappings of account identifications. The ability to 'escape' to Java and Javascript for defining application-specific XSLT extension functions proved very important.

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:lxslt="http://xml.apache.org/xslt"
 xmlns:TermsAndConditions="PriceParameter"
 extension-element-prefixes="TermsAndConditions">

<lxslt:component prefix="TermsAndConditions"
       functions="getAccruedInterestAmount ...">
 <lxslt:script lang="javascript">
  function getAccruedInterestAmount(nominal,rate){
    accruedAmount = nominal * rate * 0.01;
    return "" + (accruedAmount);
  }
 </lxslt:script>
</lxslt:component>

<xsl:template name="insertAccruedInterestAmount">
<xsl:param name="NominalAmount"/>
<xsl:param name="AccruedRate"/>
 <xsl:value-of select="
   TermsAndConditions:getAccruedInterestAmount(
   $NominalAmount, $AccruedRate)"/>
</xsl:template>
```

Figure 5: XSLT Extension Function in JavaScript

Figure 5 shows as an example how Javascript is used in an XSLT stylesheet in order to define and apply the extension function `getAccruedInterestAmount`.

XML has originally been defined by the W3C as the next generation Markup Language for the World-Wide-Web. Hence, it is assumed that XML data is distributed using the HTTP protocol. However, the HTTP protocol is 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 software architectures, such as the one for the financial trading system. To obtain the necessary qualities of service to meet our non-functional requirements, TIGRA uses middleware rather than HTTP for data transport.

Object Middleware for 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 temporarily 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 a failures occur. Transaction monitors use the two-phase commit protocol to achieve consensus between distributed components about the success of a transaction.

This large number of available middleware approaches and the even bigger number of vendors offering middleware products leads to a package selection problem [1]. Our approach for selecting a middleware for this architecture is based on the non-functional requirements for the architecture that we discussed above.

In order to select a suitable middleware approach we compared message-oriented middleware, transaction-oriented middleware and object-oriented middleware first analytically and then using prototypes. 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 is more difficult to use and to administer than object-oriented middleware, mainly because of the need to hand-code marshalling. As a result, a hybrid approach was pursued and middleware that implements the CORBA standard was selected with the constraint that the middleware had to implement the Object Transaction service in order to be able to execute sequences of operations in an atomic manner [11]. Then transactions can be used to reliably forward trade data to the Router and also for the broadcast of Routing information to input adapters. To implement the security requirements, we employ the CORBA Security service Level 1, which is well supported by several CORBA products. It provides us with primitives to define credentials for principals and to authenticate principals in order to establish a security association. It also allows us to set up access rights for principals so that we can prevent non-authorized use in TIGRA instantiations. The most difficult issue with CORBA was reliable trade delivery in a non-synchronous fashion. We could not use CORBA's synchronous object request primitive as this would have delayed the execution of front-office systems. Moreover, object request do not directly support the multicast representation that is needed in TIGRA.

The Router of the TIGRA architectural style implements a selective and reliable multicast of trade data represented in XML. The Router can only determine the group of receiver components dynamically based on the value of attributes and entities in the XML trade data representation (e.g. the `Security Type` attribute). In principle, this would require the Router to understand the XML representation, but this is undesirable for reasons of both maintenance and efficiency; the Router should not have to be changed when a DTD changes and also we should not waste time parsing the XML string during routing. Fortunately, only a limited amount of trade data information is needed for making routing decisions and we hold these data redundantly in both 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 use CORBAservices for transactions and security, a number of design options remained open for the router.

These are:

- use of the CORBA Event Service as basis of the Router implementation;
- use of the CORBA Messaging Service for 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 [11] 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 TIGRA, as the trade data that needs to be multicast from one front office system to multiple middle- and back-office systems can be regarded as typed events. Furthermore, TIGRA 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 will use the push rather than the pull approach. The Event service supports both typed and non-typed event communications. In the trading architecture event communication will be typed (using the `Routable` interface) 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. TIGRA therefore does not use the Event service.

The CORBA Messaging service is specified in [10] 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 declarations for asynchronous operations in IDL. CORBA implementations are expected to invoke call back objects transparently for the application programmer when the server object finishes the request to deliver results. 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 and secondly the Event service supports one-directional communication, while the Messaging service supports bi-directional communication.

The Messaging service, however, is unsuitable for another reason. 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 call-back objects for all those trades in order to wait for acknowledgement of the receipt of the trade objects in all middle and back-office systems. We would expect that there will be a substantial overhead involved in managing these call-back 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 by a bank as it requires modifications of the core of an object request broker, such as the IDL compiler. The trading architecture therefore does not use the Messaging service.

The CORBA Notification service was adopted by the OMG Telecommunication Task Force [12] 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 prioritization, event expiry and event filtering. This makes the service very suitable for the implementation of trade data transport. In particular, it is 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 6, the trade data is 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 Adapter from a receiving middle or back office system. The Input Adapters may also contact receiving back and middle office systems without involving an Event Channel. This is appropriate if the legacy interface to the middle or back office system already contains a queuing mechanism.

Figure 7 shows in more detail 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 relia-

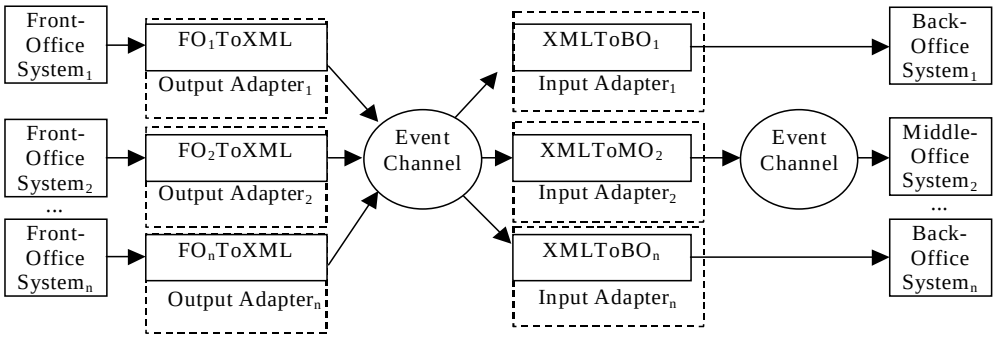

Figure 6: Use of CORBA Notification Service

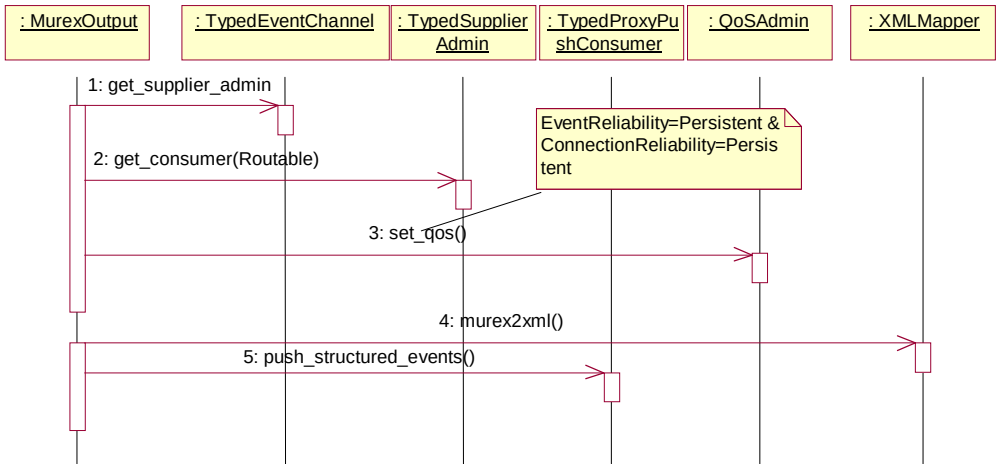

Figure 7: Output Adapter Interacting with Notification Service

bility, 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 system. The Notification service supports the administration of these filters with a constraint language.

5 Experience With TIGRA

Our experience with the TIGRA architectural style has been largely positive. We stress tested the pilot implementation before it went into production and we were easily able to scale the system up to 25 trades per second (i.e. 2.5 times the required peak throughput) by using replicated mapper objects on a cluster of Sparc Ultra Enterprise Servers. Even higher scalability could have been achieved by adding further replicated mappers and router objects. We have achieved an elapsed real-time for the multicast of a trade, i.e. the time between an output adapter taking the trade and all input adapters forwarding the trade to their receivers, of about 950 milliseconds. The time is actually an order of magnitude faster than was required, which surprised us, as we initially thought that XSL stylesheet transformations are rather inefficient.

The TIGRA *leitmotif* of addressing data integration and reliable transport separately proved highly successful. Not only has it enabled us to address semantic data integration by transforming XML documents using XSL, but it also has allowed us to keep the skill set down. XSL stylesheets that perform the mapping are now happily written by business analysts, who would have not been able to implement CORBA objects.

We have managed to implement a pilot project using the TIGRA style in about six months, both within budget and on time. We have thus shown that the architecture has been successful and can reduce cost of enterprise application integration. The pilot project further produced a re-usable support framework with common classes for Web-based user interfaces, database access and for building input and output adapters. Re-use of this framework and the general experience we made in this project will further decrease the effort that future TIGRA instantiations will demand.

6 Research Directions

We now review the implications of our findings for the research agenda on software architecture.

The combination of markup languages and middleware is

largely successful, though further work will be needed to achieve tighter integration. The use of middleware enabled us to isolate functional concerns in the mapping components. 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 `Routable`.

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 packages that were deployed in the architecture. We need to better understand the relationship between non-functional requirements and software architectures. Moreover, we need to find systematic ways to quantify non-functional requirements. The fact that scalability and performance requirements were largely estimates based on past experience put the stability of the TIGRA architecture at risk if those estimates are wrong.

7 Conclusion

We have now used the TIGRA style that we introduced in this paper in the trading department of a different bank for a similar enterprise application integration project. In this project, we have exchanged the middleware but otherwise employed the same architectural style. Instead of CORBA, we used a Java Messaging Service (JMS) for reliable trade delivery and Enterprise Java Beans (EJB) for scalable deployment of mapping components. This leads to suggest that the style is actually quite general and can be employed not only in one but in many different institutions.

The strength of middleware and markup languages are complementary. Based on the experience with this trading architecture, we expect this combination to be used in 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 use 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 expressed at appropriate levels of abstraction using standards and performed using off-the-shelf technology.

Acknowledgements

We would like to thank all members of the TIGRA team. Jürgen Büchler drove the initiative for a "new trading architecture", lead the initial research, obtained the funding and provided the atmosphere in which our ideas could mature. Walter Schwarz evaluated CORBA Notifications and developed the Router. Stefan Walther conducted the business analysis and wrote most of the semantic transformations in XSLT style sheets. Frank Wagner provided the environment for testing many of the TIGRA components. Rolf Köhling implemented a reusable set of classes to simplify the construction of input and output adapters. Andreas Heyde and Michael Koch implemented the Web-based user interfaces.

REFERENCES

[1] M. Ryan A. Finkelstein and G. Spanoudakis. Software Package Requirements and Procurement. In *Proc. of the 8^{th} Int. Workshop on Software Specification and Design, Schloss Velen, Germany*, pages 141–146. IEEE Computer Society Press, 1996.

[2] 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.

[3] 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.

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

[5] W. Emmerich. Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *Future of Software Engineering*, pages 117–129. ACM Press, June 2000.

[6] David C. Fallside. XML Schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium, April 2000.

[7] Fix Protocol. FIXML – A Markup Language for the FIX Application Message Layer. http://www.fixprotocol.org, 1999.

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

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

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

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

[12] Object Management Group. *Notification Service*. 492 Old Connecticut Path, Framingham, MA 01701, USA, January 1998.

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