A Lightweight XML-based Middleware Architecture

Welf Löwe Markus L. Noga
University of Karlsruhe
Program Structures Group
Adenauerring 20a, D-76131 Karlsruhe, Germany
{loewe|noga}@ipd.info.uni-karlsruhe.de

Abstract

Components are built for reuse, so component integration is usually personally, physically, and temporally separated from component design. For components from different sources to interoperate, adaptations are usually required. Unfortunately, most existing architectures treat them as mere work-arounds for design problems. This paper presents a lightweight XML-based middleware for component communication. In our architecture, component adaptation is considered an integral part of component deployment.
Keywords: Software Architecture, Internet Computing, Component Systems, XML.

1 Introduction

Component systems are usually built in two stages. First, preexisting components are bought or extracted from legacy systems, while missing components are designed and implemented. In the second stage, components are assembled to a system. Systems integration is personally, physically, and temporally separated from component design. Hence, mismatches between components are the rule, not the exception. This makes adaptation an integral part of component-based systems design.

Classic middleware architectures for components include CORBA [19], (D)COM [17], and Enterprise JavaBeans [22]. They power scalable distributed systems whose processing performance is crucial. Thus, they employ highly efficient binary encodings to marshal data structures and method calls in a distributed system. Unfortunately, these dense encodings are not easily amenable to adaptation. A different approach is necessary.

In addition to supporting adaptations, a new component system should satisfy the requirements to existing ones. That is, it should be general-purpose, independent of platform and language, support both value and reference types, support serialization and deserialization, support remote invocation, and offer good performance.

1.1 State of the art

XML – Extensible Markup Language [25] – has emerged as the standard in general purpose, platform-independent data exchange. XML documents carry logical markup: they are tagged according to their content structure. Document Type Definitions (DTDs) and XML Schemas [26, 27] provide type-checking for XML documents. Documents are easily accessible to standard transformation techniques, e.g., XSLT – Extensible Stylesheet Language Transformation [28] – processors. This makes XML a good candidate to connect components and host adaptations.

Most available XML parsers and XSLT transformation processors are interpreters: they read arbitrary well-formed XML, read and analyze the definition of the required structure and validate the document against it. Along with many others, the parsers provided by the Apache project [3], James Clark [9] and IBM [2] fall into this category. Most XSLT processors are interpreters as well, e.g., xt [9] and saxon [14].

In a component context, the data definitions (DTDs or Schemas) for individual components are available at compile time. Respectively, adaptation transformations (XSLT scripts) must be provided at deployment time for the system to work. These data allow considerable optimizations in parsers [18] and transformers [20, 10]. They reduce the considerable performance penalty of XML versus binary encodings.

In [8], Chen et. al use Enterprise JavaBeans runtime introspection to serialize bean state to XML. Their approach is interpretative, although they formulate generation of specific serializers as a possible future enhancement. They discuss typing and DTD generation, but do not mention Schemas. Their application context is web page generation, which does not require deserialization. Consequently, this topic is not discussed.

The Castor project [7] provides a generic framework for Java data bindings. Among other targets, Castor supports relational databases and XML. The project employs generator techniques to create specific serializers and deserializers, but does not create specific typed DTDs and Schemas for Java classes.

Modern component architectures like MS.NET [1] use XML to encode calls between components and XML serialization and deserialization techniques to encode and decoder, resp., the data to exchange. Emerging web service standards like SOAP and WSDL [23, 24] guarantee interoperability with other architectures: SOAP defines a communication envelope for XML data exchange, and may thus

carry messages originating from our infrastructure. WSDL is a web services definition language, which widely peruses XML Schema. It may be used to encapsulate schema definitions when calling via SOAP. However, it provides very little in the way of extensions: typed references to services are a missing concept, as well as standardized language mappings.

1.2 Overview

Component IDLs include typed references to components. They support both call-by-value and call-by reference, using generators to supply stubs and skeletons for remote invocations. In contrast to all other XML based approaches, we provide these classic concepts while using XML as our interchange format. Unlike all techniques mentioned above, we additionally support the adaptation of the calls and parameters by transforming the XML streams.

We distinguish components and data. For call-by-value data, object graphs are serialized to XML by the producer, transported via some channel and rebuilt from the stream by the consumer. This way, widely available XSLT processors can be employed for data adaptation. For call-by-reference data, the inverse transformations are also required. In general, they cannot be derived automatically and must thus be provided to the system. Components are always communicated by reference. We provide proxies in the consumer's address space. Of cause, transformations may apply recursively to their method arguments.

As components and deployment contexts may change rapidly in software evolution, manual implementation is not an option. Our generative approach analyzes component sources with the *Recoder* system [16] to generate DTDs and Schemas. In contrast to generic solutions, this approach allows for application-specific DTDs and Schemas which carry full type information. Specialized serializers and deserializers are automatically generated and subsequently woven into the components with *Recoder*. To optimize the entire processing pipeline, we also employ XSLT script compilers [20, 10].

The remainder of this paper is organized as follows: first, we introduce the basic model for communication between components and discuss the problems to solve, Section 2. Section 3 specifies the processing pipeline to serialize a data structure, to transform it, and rebuild the new data structure. Finally, section 4 summarizes our results and outlines directions for future work.

2 Basic Model

We first define a model for components building on existing object oriented type systems. Then we model the problem of communicating data structures and subsequently consider the impact of adaptations and transformations.

2.1 Component and Type Model

We define components to be software artifacts with typed input and output ports. Input ports are connected to output ports via communication channels called connectors. The notion of ports and connectors are known from architecture systems [21, 6]. While connectors may be as complex as components, and thus require the same amount of consideration in design, we assume most connectors to be simple point-to-point data paths. Components are active, i.e. they execute their code autonomously. At input ports they require data from other components. At output ports they provide data to other components. Figure 1 sketches this basic model.

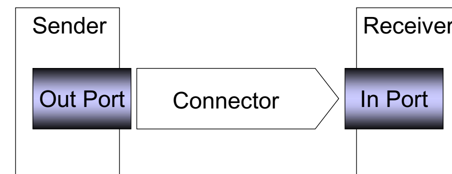

Figure 1. Basic Component Model

In general, ports are implemented by sequences of basic constructs like method calls, RPCs, RMIs, input – output routines etc. provided by the implementation language or the component system. In contrast to those basic features, a port abstracts from implementation details. A connector is defined by an out-port and an in-port sharing the same connector identity. We refer to [4] for automatic component adaptation by changing port implementations and to [12] for extracting ports from legacy code.

Most type systems in modern programming languages are roughly equivalent. They distinguish:

- value types (where the identity and the value of an object is the same) vs. reference types (where identity and value of objects can be distinguished),
- concrete types (with implementation of methods) vs. abstract types (interfaces),
- language defined types (where the implementation of methods is not accessible) vs. user defined types (where we consider the implementation of methods available).

Users define types with classes. In fact, a class defines two types: a class type with the shared (or static) attributes and an object type capturing the attributes of the instance objects of that class. There are inheritance relations between types, subtype inheritance (the subtype declares to implement the supertype interface) and implementation inheritance (the subtype gets the supertype's method implementations, conflicts are resolved, however.). Only the former pertains to component systems, the latter only offers convenient editing and consistency.

Our middleware architecture communicates data as well as references to components. Therefore, we need component types suitable to out type system. Given an active component with ports in its code, it can be transformed into a passive service by the technique of *program inversion*, cf. [13]. Each in-port is represented by a method, each out-port by a method call. The set of methods is the type of the service. This notion of components as passive services corresponds to the IDL or WSDL view.

2.2 Data Structures and Services Provided

In analogy to the EJB concepts, we distinguish data structures and services. The former capture state and only define access methods, the latter only provide interfaces. Implementations of services must not be communicated – only interfaces.

Let A and B be two different components. If A provides a data structure (object) x to its environment at an out-port, there exists a data type (class) X in the scope of A to define x. In addition to the attribute fields, X may also define some methods on them. An x object may be communicated to an in-port of B expecting a data structure y defined by a class Y. Provided x captures all necessary information to generate y objects, we can define an adaptation routine, however. Adaptation may involve filter, rename, permutation and other transforming operations. As the access pathes change, X methods may become ill-defined. Instead, Y methods apply. Therefore, there is no need to transmit field names or methods along with the pure data objects.

Now consider communicating the service A itself. In general, its methods require local state and sub-services, both captured in attributes of A. Beside possibly being private, access to those attributes could be restricted for synchronization reasons. Hence, we cannot communicate local attributes. Thus, service method implementations cannot be communicated either. Apart from a service interface description, we only provide an identifier for the service object. Some adaptations are allowed on services: renaming of methods, permutation, binding and filtering of parameters as well as filtering of results. Recursively, further adaptations may apply on the parameters and results themselves.

For data, there are two main approaches to communication: value or reference semantics. With value semantics, the producer passes a copy of the data structure to the consumer. As data may consist of structured objects with mutual references, deep copies are required. This makes for large initial transmissions, but all subsequent consumer operations act on the local copy. Using reference semantics, the producer passes a global identifier to the consumer and exposes a remote access interface. While identifiers are usually of fixed and small size, all subsequent consumer operations must traverse the network to act in the producer's address space. Depending on access profiles, communication overhead and latency, either approach may

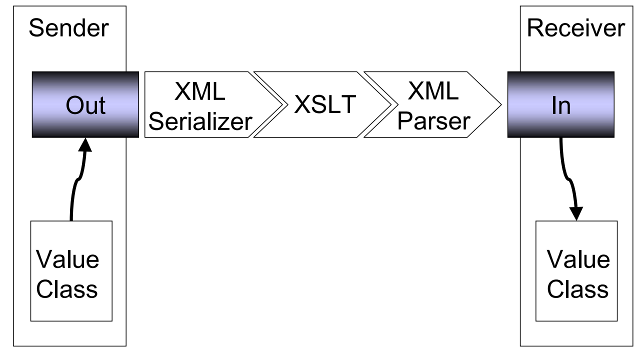

Figure 2. Adapted value communication between components.

be optimal. [15] discusses general data structure transformations to optimize w.r.t. an access profile.

If we went for marshalling only, we could handle reference objects and services in the same way. However, adaptations and transformations complicate the architectural choice. With reference semantics, the remote access interface must provide access to the transformed data structure. Not every data transformation can be expressed by a service transformation, i.e. a transformation of the access routines to the attributes. E.g., a transformed data structure contains the sum of two integers in the original structure. Furthermore, this example shows that the semantics of the inverse transformation are not unique. In general, it has to be explicitly defined.

The straightforward approach to implement reference data structures is to trigger complete transformations (forth and back) atomically for every access on either side of the communicating components. This preserves reference semantics, but leads to huge inefficiencies for repeated accesses. In general, there are no partial transformations to alleviate this problem[1]. A modification of this approach caches the transformed data structure in the producers' address space. To preserve reference semantics, cache consistency must be maintained, however.

3 Architecture

We present our XML-based architecture for connectors. Then, we discuss the generator architecture to automate connector generation. Finally, we briefly discuss the efficiency of the generated code.

3.1 Connector Architecture

For two components to communicate, an output port must be linked to an input port via a connector. Often, this connector will be an entirely passive construct. In general, it may perform data adaptations. Until now, we have abstracted from connectors' internal structure. We propose the connector architecture shown in Figure 2.

In addition to a basic transport channel capable of transmitting byte streams, e.g., a UNIX pipe, a TCP/IP socket or an HTTP connection, a connector consists of the following parts:

1. A serializer that traverses the internal data structures of the producing component and captures their state in an XML document written to the channel. In general, state may extend over an entire object graph — individual objects and references are just a simplification.

2. An optional XSLT processor that reads, filters and transforms the generated XML document to make it appropriate for the consuming component. For services, some restrictions guarantee, that the transformed structure has still a well-defined semantics. It may be operate either at the producing or the consuming end of the channel.

3. A deserializer parses the incoming XML document and reconstructs the object graph or a transformed version thereof in the consumer's address space.

In our component model, both input and output ports are typed. This translates into grammatical restrictions on the possible outputs of a serializer and the legal inputs for a deserializer. We obtain tight bounds on the admissible data exchange language by generating specific DTDs from the respective input and output port types.

While value objects are simply copied and passed form the producer to the consumer, references and services establish a permanent channel between them: an access on either side to a reference and an access of the remote side to a service (could) trigger communication. Moreover, depending on synchronization, both sides could access the referenced data or service at the same time. However, the implementation of synchronization is outside our current scope. We assume local protection against unintended use.

According to the above discussion, the implementation of the general architecture now differs for each type of object to transmit:

- Value data are copied to an XML representation, possibly transformed, and then reified on the remote side, cf. Fig. 2.

[1]We can reduce the number of transformations in accordance with the consistency model of the programming language or system. E.g., Java requires a back propagation only at synchronization points. In between, each side may work on its local copy.

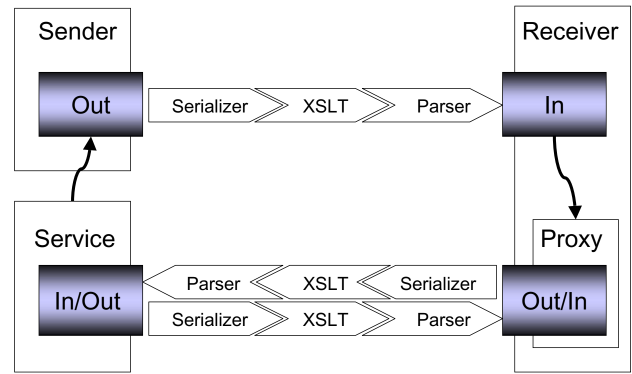

Figure 3. Adapted service communication between components.

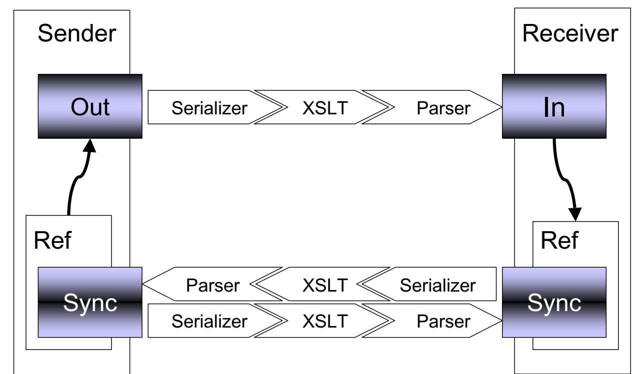

Figure 4. Adapted reference communication between components.

- Service objects are given an identifier (encoded in XML) and an XML description of the service interface. The latter may be transformed by renaming and filtering the service methods, and by reordering, binding or transforming the parameters. (Note that parameters are in-ports and the calls binding them are out-ports; the architecture recursively applies on those ports, as well.) Services are implemented with proxies on the remote side providing the transformed interface, cf. Fig. 3.

- Reference data are copied to an XML structure, transformed and reified on the remote side as values. Additionally, we define a synchronization channel. Accesses on either side operate on the local copy of the reference. At synchronization points, transformations are triggered as for the value data, cf. Fig. 4.

Synchronization points depend on the memory consistency model. If sequential consistency is required, each access to the data can trigger a communication. This could be implemented by defining an owner service for the data, which has two views on it: the original and the transformed view.

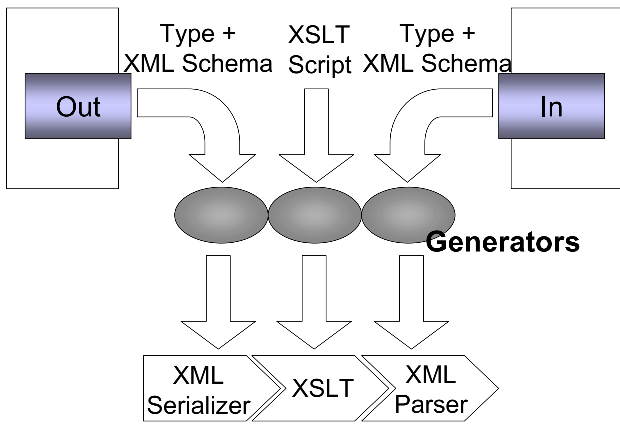

Figure 5. Generating a connector

Changes are monitored; remote view accesses trigger transformations. Alternatively, we could define a service capturing a dirty bits and communicating changes at the remote side whenever a read from an inconsistent (dirty) structure is requested. Otherwise, we operate locally. We choose this one for our example implementation. Weaker consistency models would require less communication.

3.2 Generating Connectors

The availability of formal specifications for both data types and service interfaces greatly simplifies the implementation of connectors. Writing them by hand is time consumptive and prone to errors. Using specifications, we can generate them automatically.

At deployment time, we analyze the types of data and services to communicate. This code inspection is done with a static meta programming tool[2]. Our implementation uses *Recoder* [16], a tool to analyze and transform Java code. For each data type to communicate, our *Recoder* application derives an XML Schema representation.

Using this type information, we generate the XML data serializer for the sender. The *aXMLerate* project [5] provides a toolkit to generate parsers from DTD and XML Schema definitions used for the deserialization on the receiver side. [11] demonstrates the performance benefits of generated over traditional, interpreted XML processing. The optional transformation step is defined by an XSLT script. Again, we use an *aXMLerate* generator to compile the script [20] instead of using a standard interpreting XSLT processor. Figure 5 shows how connectors are generated in a preprocessing phase.

[2] A meta programming tool analyzes and transforms program terms. We distinguish static and dynamic meta programming depending on the point in time the analyzes and transformations are performed. While static meta programs operate on the code before it is executed, dynamic meta programs analyze and change it while execution. For our purpose, static meta programming is sufficient.

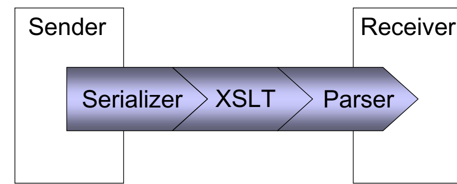

Figure 6. Connector woven into the components

A service is represented by an interface description and an object identifier. The interface description is derived from its implementation interface. With an optional XSLT script defining the service transformations, this allows to generate the remote proxy object. Note that parameters and results are value data, reference data or services. The generation of serializers, deserializers or proxies recursively applies to them.

At runtime, service descriptions are not communicated. Raw data and service identifiers are sufficient, as type information is already encoded in serializers, deserializers and proxies.

For deployment, the generated parts of the channel are closely integrated with the consuming and producing components. Again, we use *Recoder* to weave the appropriate method calls and implementations into the components, replacing the abstract ports. Figure 6 sketches the result.

3.3 Efficiency of the Generated Code

Due to generated serializers, deserializers and transformations and their integration into the components, the efficiency of our middleware is already acceptable.

The generated serializer code traverses an object graph once in depth-first order and prints XML in document order, which is depth-first. It maintains a hash table of serialized objects for the active session to detect back edges in $O(1)$. The generated deserializer code parses an XML document once in document order and reconstructs the object graph in depth-first order. It maintains an array of deserialized objects for the active session to reconstruct back edges in $O(1)$. A second pass to resolve references is not required.

XSL transformations may traverse the input document in an arbitrary fashion, although depth-first is an important special case. They may cache intermediate results of any size, but the output document is written in depth-first order. Hence, we can omit the DOM structure for the output document. Instead of constructing an XML element in the transformation, we immediately call the appropriate constructor for the output structure.

4 Conclusion

As components are usually integrated into environments they are not customized for, the communicated data is to

adapt to make the components interact correctly. We defined an XML-based architecture for component connection with adaptation as an integral part.

Connectors and their descriptions, i.e. document types, are generated from specifications. We use static type information on the provided and required data to serialize and deserialize the XML representations of services and data. XSLT specifications define the necessary transformations.

In contrast to standard XML processing, we exploit the generated document types encoding service and data objects. XSLT design tools benefit these from precise input and output document types. Moreover, it improves the software process for the design of connectors in general, introducing a notion of static type safety. Last but not least they enable optimized serializers, deserializers and transformers.

However, not all potential optimizations are implemented yet. E.g. the static analysis of the XSLT scripts does not exploit the abstract interpretation of the XSLT scripts on the DTD. Future work will also focus on optimization techniques. If we could establish a lazy construction of the XML document encoding the sender object structure, we could go beyond the static analysis of the XSLT scripts and exploit dynamic information.

References

[1] *Microsoft .NET*. Microsoft, `http://www.microsoft.com/net`, 2001.

[2] *XML Parser for Java*. IBM AlphaWorks, `http://alphaworks.ibm.com/aw.nsf/techmain/xml4j`, 2001.

[3] *Xerces Java Parser*. Apache XML Project, `http://xml.apache.org/xerces-j/`.

[4] U. Aßmann, T. Genßler, and H. Bär. Meta-programming Greybox Connectors. In Richard Mitchell, Jean Marc Jézéquel, Jan Bosch, Bertrand Meyer, Alan Cameron Wills, and Mark Woodman, editors, *Proceedings of the 33th TOOLS (Europe) conference*, pages 300–311, 2000.

[5] *aXMLerate Project*. B2B Group, University of Karlsruhe, `http://i44pc29.info.uni-karlsruhe.de/B2Bweb/`.

[6] Len Bass, Paul Clement, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[7] *The Castor Project*. ExoLab Group, `http://www.castor.org/`, 2001.

[8] Li Chen, Elke Rundensteiner, Afshan Ally, Rice Chen, and Weidong Kou. Active page generation via customizing xml for data beans in e-commerce applications. *LNCS*, 2040:79–97, 2001.

[9] James Clark. *XT*. `http://www.jclark.com/xml/xt/`, 1999.

[10] Johannes Dieterich. *Generierung von Graphersetzern als XML-Transformatoren*. Universität Karlsruhe, IPD Goos, Jun 2001.

[11] T. Gaul, W. Löwe, and M. Noga. Foundations of Fast Communication via XML. *Annals of Software Engineering — Special Volume on Object-Oriented Web-Based Software Engineering*, 2002. in revision.

[12] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-oriented configuration and adaptation of component communication. In Jan Bosch, editor, *Third International Conference on Generative and Component-Based Software Engineering, GCSE*, page 58 ff. Springer, LNCS 2186, 2001.

[13] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[14] Michael Kay. *The SAXON XSLT Processor*. `http://saxon.sf.net/`, 2001.

[15] W. Löwe, R. Neumann, M. Trapp, and W. Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS-Europe — Technology of Object-Oriented Programming*, 1999.

[16] Andreas Ludwig. *RECODER Homepage*. `http://recoder.sf.net`, 2001.

[17] *Component Object Model*. Microsoft, `http://www.microsoft.com/com/`.

[18] Markus Noga. *Erzeugung validierender Zerteiler aus XML Schemata*. Universität Karlsruhe, IPD Goos, `http://www.noga.de/markus/XMLSchema/Diplomarbeit.pdf`, Oct 2000.

[19] *Corba 2.4.2 Specification*. OMG, `http://www.omg.org/technology/documents/formal/corbaiiop.htm`.

[20] Tobias Schmitt-Lechner. *Entwicklung eines XSLT–Übersetzers*. Universität Karlsruhe, IPD Goos, May 2001.

[21] M. Shaw and D. Graham. *Software Architecture in Practice – Perspectives on an Emerging Discipline*. 1996.

[22] *Enterprise JavaBeans 1.1 Specification*. SUN Microsystems, `http://java.sun.com/products/ejb/docs.html`.

[23] *Simple Object Access Protocol (SOAP) 1.1*. W3C Note 08 May 2000, `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`, 2001.

[24] *Web Services Description Language (WSDL) 1.1*. W3C Note 15 March 2001, `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, 2001.

[25] *Extensible Markup Language (XML) 1.0*. W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[26] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.

[27] *XML Schema Part 2: Datatypes*. W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-2-220010502`, 2001.

[28] *XSL Transformations (XSLT)*. W3C Recommandation, `http://www.w3.org/TR/xslt`, 1999.