

Exploring Remote Object Coherence in XML Web Services

Robert van Engelen^{1*} Madhusudhan Govindaraju^{2†} Wei Zhang¹

¹ Department of Computer Science and School of Computational Science, Florida State University

² Department of Computer Science, State University of New York (SUNY) at Binghamton

Abstract

Object-level coherence in distributed applications and systems has been studied extensively. Object coherence in platform-specific and tightly-coupled systems is achieved with binary serialization protocols to ensure data structures and object graphs are safely transmitted, manipulated, and stored. On the opposite side of the spectrum are platform-neutral Web services that embrace XML as a serialization protocol for building loosely coupled systems. The advantages of XML to connect heterogeneous systems are plenty, but rendering programming-language specific data structures and object graphs in text form incurs a performance hit and presents challenges for systems that require object coherence. Achieving the latter goal poses difficulties by a phenomenon that is sometimes referred to as the “impedance mismatch” between programming language data types and XML schema types. This paper examines the problem, debunks the O/X-mismatch controversy, and presents a mix of static/dynamic algorithms for accurate XML serialization. Experimental results show that the implementation in C/C++ is efficient and competitive to binary protocols. Application of the approach to other programming languages, such as Java, is also discussed.

1. Introduction

XML Web services technologies have proven to be excellent vehicles for bridging the platform and programming language gap in heterogeneous distributed systems. The expressiveness and simplicity of XML paired with SOAP Web services interoperability characteristics are highly appealing compared to binary protocols. However, object coherence in platform-specific and tightly-coupled systems has been achieved for years with binary serialization protocols. By contrast, Web services are loosely coupled and are not specifically designed to meet these strong requirements.

Yet, SOAP RPC [20] provides an object and data serialization format that clearly suggests a tight coupling between programming language types and XML constructs such as primitive types, arrays, structs, and multi-references. The SOAP RPC standard appears to be particularly well-suited for RPC-based messaging by mapping application data onto SOAP RPC-encoded XML. With the move to document/literal Web services and XML schema as a popular serialization meta-format, accurate programming language type mappings have become even more important, yet more difficult to achieve because there is no standard mapping between XML schema and program language types.

Several authors [8, 10] refer to the mapping problem as the Object/XML (O/X) *impedance mismatch*, a term that bears some resemblance with the heavily-studied Object/Relational (O/R) mapping problem, thereby relating the SOAP/XML interoperability issues to the imperfection of the O/X mapping. However, a major part of the O/X-mismatch can be attributed to the limitations of the widely-used JAX-RPC implementation of SOAP in Java, see Loughran et al. [8]. They also argue that the O/X-mismatch problem is compounded by the misconception among many developers that JAX-RPC *is* SOAP-RPC.

This paper debunks the O/X-mismatch controversy, examines the coherence problem, and presents a mix of static/dynamic algorithms for accurate XML serialization within the limits of XML schema constraints. In Section 2 we argue that trying to compare XML schema to programming language type systems is counter productive. Section 3 states the requirements for object-level coherence and presents a brief overview of systems that achieve coherence using binary protocols. We discuss the design and solution space for object coherence in XML services using specific examples from Apache Axis (Java) [1] and gSOAP (C/C++) [17]. The mapping problem of programming language types to XML schema is discussed in Section 4 and specifically applied to C/C++ and Java. Our XML serialization approach for object-level coherence is introduced in Section 5 with a presentation of the algorithms. Section 6 gives performance results to verify the efficiency of the implementation on various platforms. Finally, Section 7 summarizes our findings.

* Supported in part by NSF grant BDI-0446224 and DOE Early Career Principal Investigator grant DEFG02-02ER25543.

† Supported in part by NSF grants IIS-0414981 and CNS-0454298.

2. The O/X Impedance Mismatch Revisited

Any context-free language has an infinite number of grammars to describe the syntactically valid strings. Likewise, XML content can be described and constrained in many different ways [12], e.g. using XML schema, Relax NG, DTD, and LL(1) grammars [9] to name just a few. In fact, many different XML schemas can be constructed to describe the structure of a set of XML documents. In other words, we believe it is counter productive to compare a specific schema to a programming language type system.

The primary purpose of XML schema is to provide a meta language for (manually) defining valid XML content, where certain schema components are clearly intended to make this process simpler with convenient constructs to relieve the schema author from heavy text editing and by providing modular components for elements, types, and groups of these. Any XML schema with element references, groups, and substitutions can be translated into an equivalent schema (or schema-like language) without these, while maintaining the validity constraints. Once we obtain a desugared schema by translating the unnecessary extras to simpler constructs, the mapping is much clearer. Therefore, we reject the notion that the O/X mapping is intractable. However, we agree that the mapping is a challenge for each unique programming language, especially for serializing object graphs in SOAP/XML.

We summarize and comment on the arguments commonly stated in the context of the alleged O/X-mismatch:

- *The inability to support derivation by restriction, such as restricted value ranges and patterns.* However, restricted types basically stand on their own, so the derivation hierarchy is not relevant at runtime when serializing objects. Indeed, support for restriction is not fundamentally more difficult than defining restricted types in languages that support subtyping, e.g. subrange types in Pascal. Note that languages that do not support subtyping require other mechanisms, such as the program annotations with gSOAP [17] for C/C++.
- *Not being able to map XML names to identifiers.* Punctuation and Unicode characters in XML names can be encoded with simple conventions such as with hex `_xABCD_` codes in identifier names, as suggested in the SOAP specifications. Many modern programming languages also support Unicode identifier names.
- *No mechanism for implementing XML namespaces.* We agree that Java packages and C++ namespaces cannot adequately simulate the XML namespace concept, because XML namespace resolution is not limited to types, but also determines the namespaces of local elements and attributes. Schema import, include, and redefine constructs also do not translate to package imports. We believe that the incorporation of canon-

ical namespace prefixes in identifier names, such as struct/class members with gSOAP [17], is practical and effective to resolve namespaces. Annotations suggested in [8] are not sufficient, because name clashes cannot be resolved.

- *Unsupported XML schema components.* As we mentioned earlier, many schema components that are foreign to programming language types can be eliminated and translated to simpler structures. De-sugaring a schema yields an equivalent but simpler schema.
- *Unsupported types.* All built-in XSD types can be mapped to basic programming language types or to specialized types introduced to represent XSD types.
- *Serializing a graph of objects.* SOAP-RPC encoding provides multi-referencing to serialize (cyclic) object/data graphs. Most SOAP toolkits support this feature, but not all of the toolkits necessarily follow the SOAP specification that limits the use of multi-ref for data with multiple references only. Document/literal poses some further problems, see Section 3.

This section presented mostly a schema-centric view of Web services layered on top of a programming language and its type system, where we reflected on the primary concerns for finding suitable programming language types for XML schema components. However, the latter issue also presents a point of view from a language perspective, where the problem at hand has close similarities to the object-coherence problem.

3. Object-Level Coherence Requirements

Large-scale distributed systems require strong object coherence guarantees [2] to ensure that objects moved, cached, and copied across a set of nodes in a distributed system preserve their structure and state. Coherence is a basic requirement in tightly-coupled distributed systems, such as Java RMI [15], CORBA [13], and XDR-based RPC [6]. In these systems the consistency of the distributed objects is critical and leads to fragility of the system when kept unchecked, e.g. Java class loaders dynamically verify imported classes.

Similarly, SOAP/XML processors share schemas to verify XML content, but must also be aware of the object referencing mechanism used when (de)serializing object graphs. Accurately representing object graphs in XML is critical to achieve structural coherence. We consider resolutions to the following issues critical for achieving object-level coherence in XML Web services:

- SOAP 1.1 RPC encoded multi-ref accessors are placed at the end of a message, so that all references are *forward pointing*. Object copying or pointer back-patching must be used by the deserializer for each for-

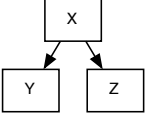
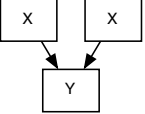
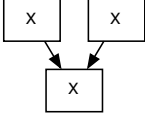
	Tree	DAG	DAG
Shape			
Schema	<pre> <complexType name="X"> <sequence> <element name="y" type="tns:Y"/> <element name="z" type="tns:Z"/> ... </sequence> </complexType> </pre>	<pre> <complexType name="X"> <sequence> ... </sequence> <attribute name="ref" type="REF"/> </complexType> <complexType name="Y"> <sequence>...</sequence> <attribute name="id" type="ID"/> </complexType> </pre>	<pre> <complexType name="X"> <sequence>...</sequence> <attribute name="id" type="ID"/> <attribute name="ref" type="REF"/> </complexType> </pre>
XML	<pre> <x> <y>...</y> <z>...</z> ... </x> </pre>	<pre> <x ref="123">...</x> ... <x ref="123">...</x> ... <y id="123">...</y> </pre>	<pre> <x ref="456">...</x> ... <x ref="456">...</x> ... <x id="456">...</x> </pre>

Figure 1: Three object referencing graph examples.

ward pointing edge to complete the edge references in the partially instantiated object graph. The SOAP 1.2 RPC encoding format is more natural, and allows both forward and back edges, but no requirements are defined to avoid excessive object copying and back-patching. Even by vigorously following the specifications and best practices, many serialization methods are too lossy to ensure object-level coherence. For example, *every* object in the graph is serialized with `id`-`href` by Apache Axis [1] using an inefficient non-scalable run-time algorithm [5] rather than the true multi-referenced objects, making it difficult to achieve object-level coherence.

- For SOAP document/literal encoding the most critical problem is the absence of a default referencing mechanism in XML. The recent publication of XML ID [22] helps and we suggest using `xml:id` and `ref` similar to SOAP 1.2 RPC encoding instead of explicitly adding `xs:ID` and `xs:REF` referencing to all components in the XML schemas, see below for more details. Also note that a means to classify pointers as reference (non-nil pointer), unique (can be nil and is the single reference to an object), or full (to shared and aliased objects) as in DCE IDL is only partially possible (i.e. using the `xs:nilable` attribute).
- To prevent loss of precision, nodes must avoid serializing floats and doubles as XSD types. We suggest using IEEE 754 floats encoded in hexadecimal or base64 by defining a `simpleType` restriction of `xs:hexBinary` or `xs:base64Binary`, respectively, see also [16]. In [3], the authors show that the conversion of floating point data to and from an XML representation can account for 90% of the end-to-end communication time. We can address this bottleneck by using hexadecimal or base64-encoded IEEE 754 floating point values. Encoding and decoding must be performed by a pre-processor prior to XML validation.

The coherence problem for serializing object graphs in SOAP document/literal format is compounded by the absence of a default referencing mechanism. Explicitly adding `xs:ID` and `xs:REF` referencing attributes to all schema components in the XML schema of a document/literal service to implement referencing is cumbersome and introduces an unnecessary layer of complexity.

To illustrate this problem, consider the three object referencing graphs shown in Figure 1. Regular tree-based XML document configurations do not require an explicit referencing mechanism, because graph nodes are simply nested in XML. DAGs and cyclic graphs must be serialized using explicit references to preserve their logical structure, e.g. using `id` and `ref` attributes. This requires additional declarations of these attributes to the schema components of the objects when document/literal style is used.

A default identifier for multi-referencing objects in XML, such as `xml:id`, helps to share schemas among organizations. Suppose that organization *A* defines a schema for an object and organization *B* wants to define object graphs where *A*'s object can be multi-referenced, e.g. in a DAG. Then, *A*'s schema must be changed to include `id` and `ref` attributes, assuming document literal encoding style is used. This is problematic, because after *A*'s schema is published it is detrimental to interoperability to change schemas.

4. Mapping C/C++ and Java Types to XML

Several Web services toolkits for SOAP/XML [20] are available for various programming languages, such as Apache Axis[1] for Java (based on JAX-RPC) and C++, SOAP Lite [7] for Perl, gSOAP [17] for C and C++, and the Microsoft .NET framework [11] for C#. In this section we specifically focus on C/C++ and Java mappings, but the principle is generally applicable to other imperative languages that are strongly typed. Even though C is

not an object oriented language, we include it in our discussion on object coherence as some C-based toolkits map XML Schemas to structs. Many of the type mapping problems apply to these toolkits as well.

The XML Web services standard supports two XML encoding styles: *SOAP-RPC encoding* style and *document literal* style [20]. The choice of encoding style is fixed in the WSDL (Web Services Definition Language) [21] interface definition of a service. The two styles differ significantly in the expressiveness of the serialized XML representation of application data, and consequently the algorithms for mapping application data to XML are different.

4.1. RPC Encoding Style

The SOAP RPC (Remote Procedure Calling) encoding style is a standard SOAP 1.1/1.2 [20] serialization format that can be viewed as the greatest common denominator of types among programming-language type systems. The encoding supports types that have equal counterparts in many programming languages, which greatly simplifies interoperability. To this end, SOAP-RPC encoding uses a subset of the XSD type system by limiting the choice of XML schema components to an orthogonal subset of structures to represent primitive types, records, and arrays. In addition, a mechanism for multi-referenced objects is available to support the serialization of object graphs.

However, there are two problems with RPC encoding. The first issue is that the multiref serialization with `href` and `id` attributes violates XML schema validation constraints, because these attributes are not part of the schema of a typical data structure. The second problem is that the serialization of nil references, multi-referenced objects, and (sparse) multi-dimensional arrays is not precisely defined, which leads to interoperability problems. More specifically, SOAP-RPC encoding requires the following considerations for effective object-coherent serialization:

- Representing primitive types as built-in primitive XSD types and vice versa;
- Mapping records (structs and classes) to `xs:complexType`;
- Mapping arrays to SOAP arrays by ensuring support for SOAP 1.1 partial and sparse arrays;
- Object graphs must be serialized with `id` and `href` multi-reference encoding;
- Support for polymorphism via schema extension;
- Choosing a mechanism for XML namespace resolution to avoid type name clashes.

Table 1 shows the mapping of primitive and compound C/C++ and Java types to XML schema types for SOAP-RPC encoding with gSOAP and Apache Axis. Note that the full set of XSD types is not covered. Additional XSD types,

C/C++ Type	Java Type	XML Schema Type (XSD Type)
bool	boolean	boolean
char	byte	byte
short	short	short
int32_t	int	int
int64_t	long	long
float	float	float
double	double	double
size_t	^a unsignedLong	unsignedLong
time_t	^b	dateTime
char*	String	string
wchar_t*	String	string
std::string	String	string
enum	^c	simpleType/restriction/enumeration
typedef T	class T	simpleType/extension
struct T	N/A	complexType/complexContent/extension
class T	class T	complexType/complexContent/extension
typedef T	class T	complexType/complexContent/extension
T [mmn]	T [mmn]	SOAP-encoded array of T
T*	N/A	the schema type of T

^a org.apache.axis.types.UnsignedLong

^b java.util.Calendar

^c org.apache.axis.enum.Enum

Table 1: Mapping C/C++ and Java types to schema types for SOAP RPC encoding.

such as `xs:decimal`, can be represented by other types, e.g. strings or subtypes (as explained in Section 1). Users can bind XSD types, and any restricted types derived from these, to C/C++ types using a `typedef` in gSOAP, e.g.:

```
typedef char *xs__decimal;
```

Each struct or class data member is mapped to a local `xs:element` of the `xs:complexType` for the struct or class. See Figure 2 for an example. SOAP-RPC encoding requires arrays to be encoded as "SOAP encoded arrays" [20], where each SOAP array is a type restriction of the generic SOAP array schema. Another disadvantage of mapping C arrays to XML is the absence of a true array type in C (arrays in C are pointers). Arrays are either declared as fixed-size arrays or have to be declared as a struct with a pointer `__ptr` and `__size` field to store the runtime array size, e.g.:

```
struct floatarray { float *__ptr; int __size; };
```

Languages that support arrays as first-class citizens, such as Java and C#, can map arrays to SOAP arrays without forcing users to adopt mapping structures.

C Source Declarations	XML Schema
<pre>typedef char *xs__decimal;</pre>	<pre><simpleType name="State"> <restriction base="string"> <enumeration value="OFF"/> <enumeration value="ON"/> </restriction> </simpleType></pre>
<pre>enum State {OFF, ON};</pre>	<pre><complexType name="Example"> <sequence> <element name="name" type="string"/> <element name="value" type="decimal"/> <element name="state" type="tns:State"/> <element name="list" type="tns:Example" minOccurs="0" nillable="true"/> </sequence> </complexType></pre>
<pre>struct Example { char *name; xs__decimal value; enum State state; struct Example *list; };</pre>	

Figure 2: Example mapping of C types to XML schema.

The XML schema standard adopted by the Web services architecture requires support for *XML namespaces* [19]. XML namespaces bind user-defined types to one or more type spaces, similar to C++ namespaces. However, C does not support namespaces. Therefore, an alternative mechanism is used by qualifying type names with a prefix:

```
enum prefix_name { ... };
struct prefix_name { ... };
class prefix_name { ... };
typedef T prefix_name;
```

Namespaces in XML Schemas are typically modeled as packages in Java, which appears more natural compared to the identifier prefixing convention. However, Java packages and C++ namespaces cannot adequately simulate the XML namespace concept, because XML namespace resolution is not limited to types, but also determines the namespaces of local elements and attributes.

4.2. Document Literal Style

Document literal style encoding is a significant departure from RPC encoding by promoting expressiveness as opposed to the simplicity of an orthogonal type system. On the one hand, the expressiveness allows variant records (unions) to be serialized and arrays can be serialized in-line instead of separately using the SOAP array encoding format. On the other hand, the absence of a standard out-of-band mechanism for object referencing, such as the SOAP-RPC multi-ref encoding with `href` and `id` attributes, is a concern for object graph serialization when graphs cannot be represented as trees. This poses additional challenges for object-level coherent serialization guarantees.

The liberation from the SOAP-RPC encoding constraints mainly affects the mapping of structs and classes to the `xs:complexType` schema component. Without loss of generalization, the specific mapping requirements are:

- Mapping XML attribute definitions within a `xs:complexType`, where XML attributes can be instances of primitive XSD types and instances of `xs:simpleType`;
- Support for repetitions of an `xs:element` in a `xs:complexType` sequence, i.e. elements that may have multiple occurrences indicated by `maxOccurs`;
- Support the use of `xs:choice` and `xs:any`;
- Support substitution groups. These can be replaced by collecting them in a choice-like structure.
- Support for `xs:group` and `xs:attributeGroup`. These macro structures have no effect on the mapping since they can be expanded within the schema and only serve as syntactic conveniences;
- Support for top-level schema element and attribute definitions and associated references. These references are immediately resolved at compile time by gSOAP,

Member <i>m</i>	Change	XML Schema Type
<i>T m</i> ;	@ <i>T m</i> ;	attribute/@type=" <i>T</i> "
std::vector< <i>T</i> > <i>m</i> ;	none	element/@maxOccurs="unbounded" element/@type=" <i>T</i> "
<i>T *m</i> ; (points to multiple elements)	int <code>__size</code> ; <i>T *m</i> ;	element/@maxOccurs="unbounded" element/@type=" <i>T</i> "
union <i>U m</i> ;	int <code>__union</code> ; union <i>U m</i> ;	choice
void * <i>m</i> ;	int <code>__type</code> ; void * <i>m</i> ;	element/@type="anyType"

Table 2: Data member changes to support document/literal style serialization.

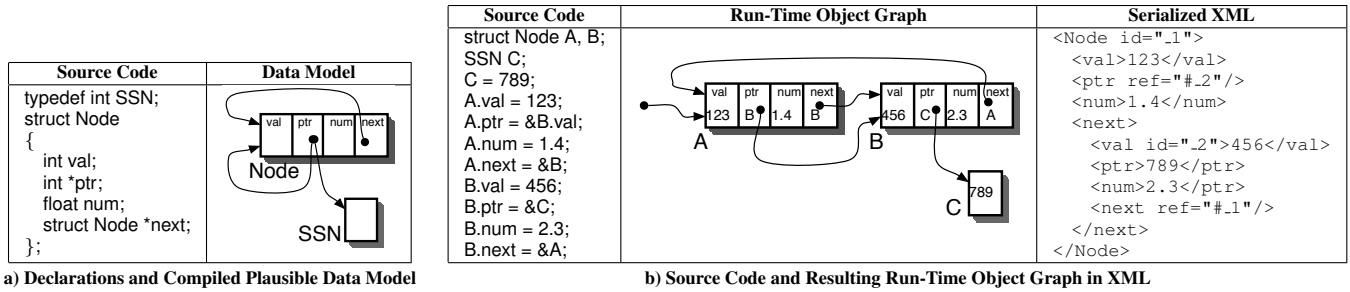
which converts them to in-lined namespace qualified local elements and attributes.

- Support for mixed content.

These `xs:complexType` content definitions are supported in gSOAP at the C/C++ source code level using the declarations shown in Table 2. Attributes are declared with a qualifier '@', STL containers such as vectors are mapped to (potentially) unbounded sequences of elements (without STL, members that point to dynamic arrays must be preceded by a `int __size` field that holds the runtime array size information). The use of STL containers and pointers to arrays as shown in Table 2 is preferred over SOAP RPC encoded arrays, see also WS-I Basic Profile [23]. A union must be preceded by a variant record discriminator `int __union` that holds the index to the union field to be serialized, and void pointers must be preceded by `int __type` field that holds the runtime type tag value of the object pointed to.

5. A Static-Dynamic Approach to XML Serialization

XML serialization of objects can be implemented with introspection, which requires a supportive run-time system. Because standard C and C++ run-time systems do not implicitly carry run-time type information on data structures and object instantiations, we implemented a hybrid form of static and dynamic type analysis for serializing C/C++ types in XML. Static analysis is used to build a plausible data model at compile time for representing the possible instances of object graphs by tracking down object relationships. This analysis is comparable to static shape analysis [4] and related to points-to analysis [14]. We then use the model to generate type-specific serialization algorithms. The generated serialization algorithms analyze the actual run-time object graph instances using compile-time hints to effectively serialize them in XML, and vice versa, using a mapping that guarantees object-level coherence. We implemented the approach in the gSOAP [17, 18] toolkit for C and C++ and tested the approach for interoperability against other toolkits such as Apache Axis and .NET.



a) Declarations and Compiled Plausible Data Model

b) Source Code and Resulting Run-Time Object Graph in XML

Figure 3: Compile-time and run-time data structure analysis.

5.1. Static Analysis

The gSOAP *soapcpp2* compiler builds a plausible data model at compile time by constructing a graph of object relationships. The model represents the possible instances of object graphs. The model is then used to generate type-specific serialization algorithms that analyze the actual runtime object graph instances to effectively serialize them in XML, and vice versa, using a mapping that guarantees object-level coherence.

Consider for example the type declarations shown in Figure 3(a) and the data model derived from it. The model shows the pointer edges necessitating the placement of pointer-checking code in the serializer and deserializer routines. For example, when serializing a `Node` instance, two addresses have to be checked for pointer references: the `Node` instance address and the `val` member address. Since these could be identical, type disambiguation is necessary to distinguish the references by keeping track of pointer's target types at run time. The runtime storage overhead of this approach is minimized, because only relevant pointers are checked and stored in a hash table for comparison to detect multi-referenced objects and graph cycles. In the example, also all integer nodes must be checked in this case for inbound pointer edges, e.g. the `val` member and the `SSN` nodes. However, no pointer checks are required for floats, e.g. the `num` member. Note that `val` is embedded within `Node`, which means that it must be annotated with an XML `id` attribute in the serialized stream.

The *soapcpp2* compiler generates optimized serialization code based on the model. These runtime serialization algorithms use two phases. The first phase determines which data components belong to the data structure and which pointers are used to reference them. This phase is only relevant for pointer-based data structures. The second phase emits the XML encoded form of the data structure by recursively serializing the sub-components.

5.2. Dynamic Serialization Analysis Phase 1

Phase 1 traverses the runtime data structure graph by visiting each node and by following the pointer references to all sub-nodes recursively. For each pointer that was fol-

lowed to a sub-node, it stores the pointer's target address in a hash table together with an identification of the data type referenced by the pointer. The hash table key is a triple of $\langle PtrLoc, PtrType, PtrSize \rangle$, where $PtrLoc$ is the pointer's target address, $PtrType$ is the type of data referenced by the pointer, and $PtrSize$ is the size of the object in bytes, which is statically determined with `sizeof`. The $PtrSize$ of dynamic arrays is computed from the array size and element size, where dynamic arrays are defined with a struct/class containing a pointer field and size field. Node pointers are only followed through to visit sub-nodes when the key $\langle PtrLoc, PtrType, PtrSize \rangle$ is not already contained in the hash table. When the key is already contained in the hash table, then the hash table entry is marked `RefType=multi` to indicate that a multi-referenced sub-node has been found. Entries in the hash table are marked `RefType=embedded` when a data element that is pointed to is embedded in larger structure, such as a field of a struct or class, or an element of an array. It is noteworthy to mention that a hash table entry is created at run time only for each pointer in a pointer-based data structure. No additional space is required to serialize non-pointer-based structures.

5.3. Dynamic Serialization Phase 2

Phase 2 emits the data in XML by visiting each node in the data structure graph and by following the pointer references to the sub-nodes for recursive serialization. Multi-referenced nodes (those whose hash table entry is marked `RefType=multi`) are serialized as multi-referenced objects referenced with `id` and `ref` in the XML stream. The serialization settings can be set to SOAP 1.1/1.2 RPC encoding. Special care is taken to serialize data elements that are embedded within structs or arrays (that is, the hash table entry for these elements are marked `RefType=embedded`) to preserve object graph coherence. The embedded property of data elements affects the placement of `id` and `ref` attributes in the encoded form of XML.

The two-phase serialization is illustrated with the example data structure shown in Figure 3(b) based on the data type declarations shown in Figure 3(a). The structure consists of three nodes, two structs `A` and `B`, and a node `C` that contains a single integer value. The serialization starts at the

<i>ID</i>	<i>PtrLoc</i>	<i>PtrType</i>	<i>PtrSize</i>	<i>PtrCount</i>	<i>RefType</i>
1	A	Node	16	2	multi
2	B	int	4	1	embedded
3	B	Node	16	1	single
4	C	int	4	1	single

Table 3: Runtime table for points-to analysis.

root struct stored at address A. The first phase consists of a pass over the entire data structure graph to collect the properties of the pointers used in the data structure and to store these in the runtime points-to table shown in Table 3.

Each entry has a unique index *ID*, a hash table key $\langle \text{PtrLoc}, \text{PtrType}, \text{PtrSize} \rangle$ consisting of a target pointer address *PtrLoc*, pointer type *PtrType*, and size *PtrSize* of the object pointed to, an indication of the number of references *PtrCount* made to the target address and the kind of reference *RefType*, which is either *single*, *multi*, or *embedded*.

The serialized XML output is shown in Figure 3(b). The root node is serialized with `id="_1"`, because it is multi-referenced. The second struct at location B is serialized in XML as a nested element of the first node struct, because it has only a single reference. Note that the `ptr` field in the first struct points to the `val` field in the second struct, which is stored at B. Because the `val` field is embedded within a struct, the `ptr` is serialized with a *forward* pointing `ref="#_2"` attribute. This ensures that the receiving side can decode the XML and backpatch the `ptr` pointer field to point to the `val` field *after* the contents of the second struct are decoded. The `ptr` field in the second struct points to a single-referenced integer located at C. The XML serialized value is placed directly in a `ptr` element without a `ref` attribute, because it is a single reference.

The gSOAP *soapcpp2*-generated deserialization routines decode the contents to reconstruct the original data structure graph. The parser takes special care in handling the `id` and `ref` attributes to instantiate pointers. When the data structure is reconstructed, temporarily unresolved forward references are kept in a hash table. When the target objects of the references have been parsed and the data is allocated in memory, the unresolved references are replaced by pointers. In effect, the unresolved pointers in the `Node` structures are back-patched with pointer values to link the separate parts of the (cyclic) graph structure together.

Pointers to polymorphic objects that are instances of derived classes are serialized using C++ dynamic binding. The gSOAP *soapcpp2* compiler augments classes with virtual serializers to enable pointer-based polymorphism via single inheritance. The static analysis determines whether pointers to instances need to be treated differently. If so, serialization and deserialization routines are generated for runtime encoding and decoding of object graphs that have pointers to derived instances, which requires the XML serialization of these instances with schema-compliant `xsi:type` attributes to hold type information so that the decoders can accurately reconstruct the object graphs.

6. Results

The use of XML as a data transport protocol comes at a price and the serialization of internal application data to XML can lead to performance loss. Furthermore, scanning an object graph for co-referenced objects can be expensive. An implementation that is compliant with the SOAP specification ideally stores serialized objects in a table. Before an object is serialized, the table is searched to determine if the object is a repeated occurrence of a previously serialized object, in which case `id-href` should be used. In Java it is important to use the *IdentityHashMap* class for reasons of efficiency. This class is specifically designed for cases where reference-equality semantics are needed.

For the deserializer, putting the multi-references object back together is also expensive. As multi-ref accessors are placed at the end of a message in SOAP 1.1 RPC encoding, all multi-references are *forward pointing*. When a streaming parser, such as SAX or XPP, is used, a co-referenced object can only be deserialized *after* the parser has processed the multi-ref objects at the end of the message.

The performance in the number of roundtrip messages achieved per second of a gSOAP benchmark client and server application is shown in Figure 4. The performance of a round-trip message containing a struct of size 1.2K is shown over HTTP 1.1 without using keep-alive connection persistence. The serialization algorithms are efficient, because runtime pointer checks for serializing object graphs are restricted to pointer-based objects only as determined from the static data model.

Table 4 compares the end-to-end performance of the gSOAP toolkit for an array of strings of various sizes with and without the multi-ref enabled (for strings) to analyze the overhead of the algorithm that ensures object coherence. The performance data in the table shows that the overhead of 2% to 3% on the total time is minimal. This test was con-

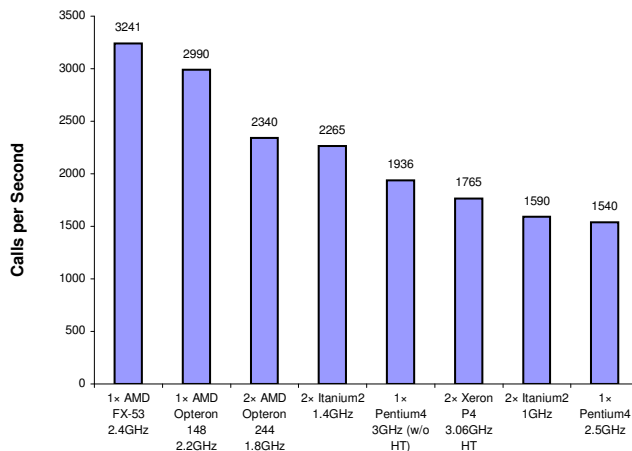


Figure 4: Performance results in number of roundtrip messages per second of a benchmark client/server application on various platforms.

Array Size	gSOAP without Multi-ref	gSOAP with Multi-ref
1	0.576	0.596
10	0.589	0.617
100	0.791	0.811
1000	2.718	0.2760
5000	10.647	10.849
10000	20.681	20.808
25000	52.177	53.025
100000	208.479	213.669

Table 4: End-to-end performance (in milliseconds) of a gSOAP benchmark application with and without multi-ref encoding. The data show that gSOAP’s multi-ref implementation adds just 2% to 3% overhead, which is minimal.

ducted on two dual processor Linux machines, each configured with 2.0 GHz Pentium 4 Xeon with 1GB DDR Ram and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux 3.1 (“sarge”) with the 2.4.26 kernel. The machines were connected by Gigabit Ethernet. gSOAP version 2.7e was compiled with gcc/g++ version 3.3.4. Relevant socket options include SO_KEEPALIVE, TCP_NODELAY, SO_SNDBUF = 32768, and SO_RCVBUF = 32768.

7. Conclusions

This paper demonstrated that object-level coherence in XML Web services can be achieved with specialized serialization algorithms to ensure data structures and object graphs preserve their state and structure when passed from one service application to another or when stored and retrieved in XML format. This paper discussed the design space for mapping issues, along with algorithms for serializing non-primitive data structures in C/C++ and Java into XML format, while providing object-level coherence and performance guarantees. The presented approach works best with SOAP 1.2 RPC encoding as opposed to the document/literal serialization style, which lacks a default ID-REF referencing mechanism. We hope our work will spur the Web services community to converge on an XML ID standard and a universal reference mechanism.

Acknowledgments

The authors would like to thank Martin Kuba from the Supercomputing Center, Institute of Computer Science, Masaryk University, Brno, Czechoslovakia, for testing the performance of gSOAP on a wide range of machines.

References

- [1] Apache Foundation. Apache axis project. Available from <http://ws.apache.org/axis>.
- [2] A. Bakker, M. van Steen, and A. S. Tanenbaum. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, 1999.
- [3] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *proceedings of the 11th IEEE International Symposium on High-Performance Distributed Computing*, 2002.
- [4] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages*, pages 1–15, 1996.
- [5] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. A benchmark suite for SOAP-based communication in grid web services. In *proceedings of Supercomputing*, November 2005.
- [6] IETF. XDR specification. www.ietf.org/rfc/rfc1014.txt.
- [7] P. Kulchenko. SOAP::Lite for Perl. Available from <http://www.soaplite.com>.
- [8] S. Loughran and E. Smith. Rethinking the Java SOAP stack. In *IEEE International Conference on Web Services (ICWS)*, pages 12–15, 2005.
- [9] W. Lowe, M. Noga, and T. Gaul. Foundations of fast communication via XML. *Annals of Software Engineering*, 13:357–379, 2002.
- [10] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles, and rectangles. Available from <http://research.microsoft.com/~schulte>.
- [11] Microsoft. .NET framework. Available from <http://www.microsoft.com/net>.
- [12] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.
- [13] OMG. CORBA component model. <http://www.omg.org>.
- [14] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- [15] Sun Microsystems. Java programming language. Available from <http://java.sun.com>.
- [16] R. van Engelen. Pushing the SOAP envelope with Web services for scientific computing. In *proceedings of the International Conference on Web Services (ICWS)*, pages 346–352, 2003.
- [17] R. van Engelen and K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In *proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid*, pages 128–135, May 2002.
- [18] R. van Engelen, G. Gupta, and S. Pant. Developing web services for C and C++. *IEEE Internet Computing*, pages 53–61, March 2003.
- [19] W3C. Namespaces in XML specification. Available from www.w3c.org.
- [20] W3C. SOAP 1.1 and 1.2 specifications. Available from www.w3c.org.
- [21] W3C. WSDL web services description language specification. Available from www.w3c.org.
- [22] W3C. XML ID 1.0 specification. Available from www.w3c.org.
- [23] WS-I Organization. Basic Profile BP1.0a. Available from www.ws-i.org.