AD HOC INTERFACE EVOLUTION IN SOAP

Nikita Schmidt, Ahmed Patel, and Mikhail Sogrine
Department of Computer Science, University College Dublin
Belfield, Dublin 4, Ireland
apatel@cnds.ucd.ie

ABSTRACT

Modern distributed systems often allow independent development of their components. Successful evolution of such systems depends on the flexibility of component interfaces. This paper puts forward a framework for defining and implementing interfaces that can independently evolve while preserving backward and forward compatibility. This framework is based on the SOAP data model and remote procedure call representation.

KEYWORDS

Distributed system; SOAP; Procedure; Interface; Evolution.

1. INTRODUCTION

Distributed systems and applications on the Internet are becoming increasingly common. Such systems comprise a number of independently operating components, often residing in different geographical locations. These components usually communicate with each other using an application-specific protocol.

As the systems evolve, so does the protocol, adapting to the changing needs of the applications. Because most of such systems have traditionally enjoyed a centralised development model, protocol evolution is usually governed by a central authority. This authority integrates new features and implements other changes in the protocol, releasing new protocol versions when needed.

The appearance of decentralised distributed systems makes it impractical to depend on a central authority to update the protocol. Rapid evolution of independent components of a global distributed system may be a key to their success in the competitive Web environment. This evolution would be hampered by a dependency on centralised protocol management.

Frameworks such as ASN.1 (ISO/IEC 8824-1:1998), CORBA (OMG, 2002), and XMLP/SOAP (W3C, 2003) allow an application developer to define communication protocols by specifying component *interfaces*, following the remote procedure call concept. All these techniques provide protocol and interface evolution facilities to a certain extent. However, ASN.1 and CORBA depend on a central authority to implement protocol modifications. SOAP, on the other hand, offers more flexibility and, based on XML, is widely used in Web applications, partly due to its native integration with XML-based technologies.

This paper presents an approach to specifying SOAP-based interfaces for distributed systems that can evolve in a decentralised, ad hoc fashion. It draws on the authors' experience in building a distributed search and advertising system for the Web, which allows independent ownership and management of its components in a (generally) non-cooperative environment (Khoussainov et al, 2001). Similar problems have been studied in the context of software reuse (Zaremski and Wing, 1995; Christiansen et al, 1997; Hemer and Lindsay, 2002) and database schema evolution (Lerner, 2000). We harness the flexibility of the hierarchical SOAP data model to improve robustness of the system. Forward and backward compatibility is also discussed.

2. SOAP AND PROTOCOL EVOLUTION

The Simple Object Access Protocol (SOAP) is a product of the World Wide Web Consortium's XML Protocol Working Group. SOAP "is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment". It specifies how method calls and return values can be transmitted as XML messages. This framework is extensible and independent of any particular programming model.

When speaking of the evolutionary ability of a communication protocol such as SOAP, two concepts must be distinguished: the evolution of the *protocol* itself and the evolution of application *interfaces*. SOAP evolves in a traditional way, using a centralised versioned model. Application interfaces are independent of this process and are generally controlled by the application developer. This paper is concerned with the interface evolution only.

The data model used by the SOAP RPC (Remote Procedure Call) Representation (W3C, 2003) specifies representation of three different categories of values: *simple values*, *arrays*, and *structs* (structures). Simple values are encoded as character strings, while arrays and structures are encoded as XML tree nodes.

A structure is a compound value whose constituent sub-values are distinguished solely by their names. An array is a compound value whose sub-values are distinguished solely by their position (their names are ignored). Although the names of array elements are ignored, they still must be specified in the XML model.

If interfaces are allowed to evolve independently, an application may not always be able to reliably detect whether a node is a structure or an array in the SOAP message it received from the network, because that message may have been formed according to a different interface specification. The `nodeType` attribute defined by SOAP may not be present to identify the type of a node. To reduce possible ambiguity, for the purposes of this study we assume that all array elements have the same name in the same array instance.

3. COMPATIBILITY ISSUES

Two communicating applications, potentially using different interfaces, may pursue the following goals:

- *Correctness*: the receiver must understand the sender's message as intended by the sender, or it must return an appropriate failure code without attempting any action specified in the message.
- *Backward compatibility*: the ability of the receiver to detect the sender's interface version if that version is known to the receiver, and to interpret the message accordingly.
- *Forward compatibility*: the ability of the receiver to understand the message if the receiver does not know the interface used by the sender.

Of these, correctness is usually the primary requirement. Backward compatibility is traditionally achieved via interface (protocol) versioning, when the sender supplies a unique ID for the interface version it is using along with the message. The receiver then uses this ID to determine whether it supports this version.

Forward compatibility is trickier. A possible solution is to make the sender fall back to an interface version understood by the receiver. For this, the sender needs to know which versions the receiver understands — either by trial and error (assuming that the correctness property holds), or by asking the receiver through a separate mechanism (e.g., a dedicated interface call).

Versioning, however, introduces additional burden, as it requires that both sender and receiver support many different versions. This burden is a serious obstacle on the way of independent, decentralised interface evolution. For example, adding a new parameter to a procedure would require creating a new interface version, adding a new version check in the receiver, and implementing a fallback algorithm in the sender.

An alternative approach which does not suffer from these deficiencies is based on the hierarchical nature and flexible naming mechanism of the SOAP data model. To ensure that the desired properties hold, it employs two sets of rules: *interface evolution rules* and *message interpretation rules*. Interface evolution rules define the possible transformations of an interface so that a message formed according to the "new" interface can be correctly understood by a receiver that only knows the "old" interface but follows message interpretation rules.

4. INTERFACE EVOLUTION RULES

A change of an interface can be considered as a set of small individual *actions* applied to the "old" interface to produce the "new" one. A single action affects one type or sub-type of the interface specification. For example, a structure can be expanded by adding a new element; an integer value can be replaced with a real; a single value can be replaced with an array of such values; and so on. Possible actions that may happen during interface evolution are considered below, arranged by the category of types they apply to: structures, arrays, and simple types. Each action is discussed in terms of preserving correctness and compatibility.

4.1 Structures

Add mandatory element. This action breaks both backward and forward compatibility. Backward compatibility is broken because a receiver implementing the new interface will reject messages formed according to the old interface. Forward compatibility is broken because an old receiver cannot meaningfully interpret the new element, and it cannot ignore it either because doing so would contradict the intent with which the element was declared mandatory. To ensure that the element is not ignored by old receivers, all its values must be explicitly marked in messages as mandatory.

Add optional element. Backward and forward compatibility is preserved. A default value for the new element may be supplied in the interface specification. Certain values may be marked in messages as mandatory if their correct understanding by the receiver is more important than forward compatibility.

Delete mandatory element. This breaks forward compatibility. Backward compatibility is preserved, as a new receiver can ignore the value supplied by old senders, unless it is marked in messages as mandatory.

Delete optional element. Backward and forward compatibility is preserved.

Make a mandatory element optional. This action generally breaks forward compatibility. A sender may maintain forward compatibility by always supplying the element's value.

Make an optional element mandatory. This action breaks backward compatibility.

Change default value of an optional element. This action breaks correctness. Correctness may be recovered by breaking backward and forward compatibility (e.g., via versioning). However, forward compatibility may be preserved by explicitly providing the element's value.

Move elements into a sub-structure. This action generally preserves compatibility, provided that the name of the new sub-structure is chosen unambiguously.

Replace structure with its elements. This is the reverse of the previous action. It can only be done if the structure in question is an element of another structure and that no name conflicts occur as the result.

4.2 Arrays

Transform to single instance. This action replaces an array with a single instance of its element. Forward compatibility is preserved here, as an old receiver can convert the single instance from a new message to a single-element array. Backward compatibility can only be ensured by providing a rule that determines which element of the array is to be taken by the new receiver (for example, by selecting the first element).

4.3 Simple types

Change to another simple type. For example, change integer to real. The consequences of this action can be controlled by using explicit type specification via the `xsi:type` attribute.

Change to an array. This action replaces an element with an array of elements of the same type. It preserves compatibility in the presence of an element selection rule (e.g., 'select the first element').

Change to a structure. This action preserves compatibility in the presence of breadth-first search and structure expansion rules (see below).

5. MESSAGE INTERPRETATION RULES

An analysis of the interface evolution actions described above produces the following set of rules which can be adopted by a receiver in order to achieve the best compatibility.

Discarding the unknown. Structure elements not defined by the receiver's interface are to be ignored, unless at least one of them is marked as mandatory, in which case message interpretation must fail.

Breadth-first search. A structure element which is defined by the interface but is missing in the actual message is to be searched for recursively in the undefined structures and arrays in the same structure.

Structure expansion. If a structure element defined by the receiver's interface specification is not found in the message, and this element is itself a structure or an array, the sub-elements of this element are to be matched against the actual structure in the message.

Array element selection. If a breadth-first search for an element finds multiple instances of the element inside an array, the first instance is to be used, unless the instance number is explicitly specified by an attribute of the array's root element.

The recursive nature of these rules brings the ability to resolve interface changes that comprise more than one action. For example, if an element was encapsulated in a structure, which was then encapsulated in another structure, breadth-first search will still find the element even if the receiver only knows the original version of the interface. The relative priorities of these rules are important, as different priorities may cause different interpretations of the same message. Therefore, only the actual implementation of message interpretation rules can define a suitable set of interface evolution rules that ensure a given compatibility level. These interface evolution rules work by placing restrictions on the actions described in Section 4.

A sender may sometimes want to control message interpretation rules invoked by the receiver, for example when use of these rules by a receiver designed for another interface may cause misunderstanding. This control can be exercised by supplying special element attributes. These attributes may selectively inhibit the above rules or provide additional hints, such as array element number for the *array element selection* rule.

6. CONCLUSION AND FUTURE WORK

This paper outlined possible scenarios for independent ad hoc interface evolution in a distributed system that uses SOAP protocol. These scenarios were discussed in terms of actions, each action affecting one aspect of the interface. The impact of these actions on protocol compatibility was explained.

Based on these actions, a number of message interpretation rules were proposed, which, when adopted by the system, can assist smooth interface evolution. Different implementations of these rules can achieve different system behaviour with respect to interface changes. Depending on a particular implementation, action constraints can be defined to achieve a required compatibility level.

A further study can define practical combinations of message interpretation rules, based on case studies of actual interface changes in distributed systems. Development of a suitable formalism for the SOAP data model that could cover interfaces and their dynamics would be useful in future work on this topic. Previous research in the areas of software reuse and database schema evolution can serve as a starting point.

ACKNOWLEDGEMENT

The support of the Informatics Research Initiative of Enterprise Ireland is gratefully acknowledged.

REFERENCES

Christiansen et al, 1997. Type management: A key to software reuse in open distributed systems. *Proceedings of the 1st International Enterprise Distributed Object Computing Workshop (EDOC '97)*. Gold Coast, Australia, pp. 78–89.

Hemer, D. and Lindsay, P., 2002. Supporting component-based reuse in CARE. *Proceedings of the 25th Australasian Conference on Computer Science*. Melbourne, Australia, Vol. 4, pp. 95–104.

Khoussainov et al, 2001. Adaptive distributed search and advertising for WWW. *Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 2001)*. Orlando, USA, Vol. 5, pp. 73–78.

Lerner, B. S., 2000. A model for compound type changes encountered in schema evolution. *In ACM Transactions on Database Systems*, Vol. 25, No. 1, pp. 83–127.

OMG, 2002. *Common Object Request Broker Architecture: Core Specification*. Object Management Group, Needham, MA, USA. Revision 3.0.1.

W3C, 2003. *SOAP Version 1.2*. World Wide Web Consortium. W3C Proposed Recommendation.

Zaremski, A. M. and Wing, J. M., 1995. Signature matching: A tool for using software libraries. *In ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, pp. 146–170.