Language Support for Distributed Proxies

Darpan Saini
School of Computer Science
Carnegie Mellon University
dsaini@andrew.cmu.edu

Joshua Sunshine
School of Computer Science
Carnegie Mellon University
sunshine@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
aldrich@cs.cmu.edu

ABSTRACT

Proxies are ubiquitous in distributed systems. They are mainly used to provide transparent access to server objects, and in some cases for additional functions such as caching, message routing, marshalling, and un-marshalling of data. In this paper, we discuss several software engineering problems associated with using proxies in a distributed system. We believe that proxies in distributed systems suffer from: 1) redundant code 2) multiple data transformations that have to be written before data is marshalled for transfer over the wire, and 3) in the case when proxies are automatically generated, there is no universal or generic mapping (to date) from XML types to Object types and vice versa. We describe these problems using an example, and discuss the partial solutions provided by prior research. A better understanding of these problems may help show the way towards a future solution.

1. INTRODUCTION

Proxies are ubiquitous in distributed systems. The proxy principal as introduced in [4] is an object-oriented way to structure a distributed system. However, this is not limited only to object-oriented designs but applicable to other kinds of systems as well. In its most basic form, a proxy shields a client from the complexities of accessing the server and its internals by providing the same interface as the server. This way the client deals with the proxy as if it was the server object itself and calls the required services. In addition to providing transparent access to the server, a proxy may sometimes perform additional duties such as caching, maintaining a connection pool, and message routing. Nevertheless, there are certain responsibilities that a proxy must fulfill in almost all circumstances. Depending on the underlying technology used, these may come for free or otherwise have to be encoded by developers themselves. These are

1. Creating and closing connections to the server.
2. The marshalling and unmarshalling of method parameters.
3. The actual sending to and receiving messages from the server.

We discuss problems with proxies with respect to a web application that uses several external web services. It is common practice for a server to publish its web services in a WSDL[1] file and for a client to create proxies by looking at this WSDL description. However, our claim is that these proxies contain redundant code since for each method in the WSDL, a corresponding delegate method is written in the proxy. All these delegate methods essentially perform the same duties and contain the same code except for a few parameters. In addition, data that is passed to these methods needs to be transformed so that the proxy can marshal it appropriately for the server. If the same service is provided by different servers then this transformation has to be done differently for each of them. A number of tools exist, mostly IDE plugins such as the Web Tools Platform (WTP)[2] for Eclipse and web services plugins for the NetBeans IDE[3], that automatically generate proxies by looking at the WSDL description of a web service. Not only do these tools suffer from the problems mentioned above but from another called the X/O impedance mismatch[3]. The X/O impedance mismatch is the lack of a generic mapping from XML types to object types and vice versa. Because of this often the generated code does not accurately represent design intent in the XML.

This paper illustrates these problems in more detail through a running example of a web application.

2. EXAMPLES

In this section, we describe a typical web application that uses web services to accomplish the task of verifying and charging a credit card. Our example web application is an online bookstore that offers features such as 1) browsing through a book catalog, 2) creating a shopping cart, 3) adding and removing items from the cart, and 4) checking out items from the cart. For our purposes, we only concentrate on part 4 in which a user can check out items from the shopping cart using their credit card. The bookstore in this case has to first verify the authenticity of the credit

[1] http://www.w3.org/TR/wsdl20/
[2] http://www.eclipse.org/webtools/
[3] http://www.netbeans.org/

```
<xs:element name="chargeCreditCard">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="card" nillable="true" type="ax22:CreditCard"/>
      <xs:element minOccurs="0" name="amount" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 1: WSDL description of the chargeCreditCard service

card, and then make the charge. We implemented our services and the client using the Java programming language. The application server used is Apache Tomcat[4], and the service runtime is Apache Axis 2.0[5]. We chose this platform because of the popularity of these two runtimes.

To accomplish these tasks the bookstore uses three web services provided by an external banking system. It is common practice for a banking server to publish its web services using a WSDL description. The WSDL in this case provides three services. The `chargeCreditCard` service is defined in Figure 1. The WSDL contains similar descriptions for another service called `isValidCreditCard`. All the services mentioned here require a complex type `CreditCard` as a parameter, defined in Figure 2. For brevity, we have reduced the `CreditCard` type to only contain the name and number. In a more realistic example, the `CreditCard` would contain many other attributes such as address, and an expiration date.

2.1 Writing the proxy

Inside the bookstore application, the most obvious way of accessing the bank's web services is through a proxy. This proxy can be hand-written, but the current state of the art (e.g WTP, Netbeans) is to automatically generate proxies from the WSDL descriptions. The exact classes generated by the proxy generator or if written by hand may vary, but the following classes are illustrative:

1. **BankServerStub**. This class contains java methods that are counterparts of services that the server provides. It may contain methods that return service endpoints or ports that are in turn used to call the delegate methods. The stub is responsible for marshalling the parameters and un-marshalling the results. In the case of web services, the marshal operation creates a SOAP[6] message and passes it to the service, and reads the result from the returned SOAP envelope. Generally creating and reading from the SOAP envelope are library functions.

2. **CreditCard**. This is the type of parameter that the methods defined in the WSDL expect. The stub defined above should be capable of marshalling an object of this type into a SOAP message. For our example, the return types of all services are simply booleans and so no other types need to be created. If code is generated, then this type is likely to suffer from the X/O impedance mismatch (described in Section 2.4).

3. **ObjectFactory**. The ObjectFactory contains factory methods that can be used code to create objects of type CreditCard. This is optional, but is good programming practice and tools usually generate it.

In the following sections we describe some of the problems associated with such proxies.

2.2 Code redundancy

Proxies have redundant code because for each method defined in the WSDL there is a corresponding delegating method in the proxy. It turns out that all these delegate methods basically perform the same functions. These are

1. marshal the method parameters for sending them over the wire,

2. send a message to the server and receive the result, and

3. un-marshal the result and return it to the client.

Although the code in our example was written to work with the Axis 2.0 runtime (which takes care of opening and closing connections to the server), the steps for delegation in the proxy would be same across all runtimes. These involve first creating a SOAP message out of the passed parameters, creating an OperationClient[7] to invoke a web service and receive the result, and read the return value from the resulting SOAP envelope. Since these operations are common to all the methods in the proxy, the result is a tremendous amount of redundant code. Writing a proxy using static reflection or class morphing[2] can remove such redundancy.

Huang et al. [2] describe a technique call class morphing that can be used to statically generate type safe implementations of the Proxy design pattern [1]. In their language MorphJ, it is possible to reflect over the methods and fields of a type and generate another type. This generated type delegates all method calls to the original but with some added functionality. Methods and fields of a type can be matched using fairly

[4]http://tomcat.apache.org/
[5]http://ws.apache.org/axis2/
[6]http://www.w3.org/TR/soap12-part1/

[7]This class, in the org.apache.axis2.client package of Axis 2.0, provides an execute() method that is used to invoke a web service. Other runtimes would have a similar mechanism.

```
<xs:complexType name=‘‘CreditCard”>
  <xs:sequence>
    <xs:element minOccurs=‘‘0” name=‘‘name” nillable=‘‘true” type=‘‘xs:string” />
    <xs:element minOccurs=‘‘0” name=‘‘number” type=‘‘xs:long” />
  </xs:sequence>
</xs:complexType>
```

Figure 2: WSDL description of the CreditCard type

```
public class ClientProxy<T> {

  <R,A*>[m] for(public R m(A) : T.methods)
  public R m (A a) {
    OperationClient service = createClient(m.name);
    SOAPEnvelope env = marshal((List<Object>) a.asList());
    service.addMessageContext(
            new MessageContext().setEnvelope(env));
    service.execute(true);
    SOAPEnvelope returnedEnv = // extract returned envelope
    R result = unmarshal(returnedEnv);
    return result;
  }

  //constructor
  //createClient
  //unmarshal, getObjectModelElementFromType, getObjectModelElementFromXML

  public static SOAPEnvelope marshal(List<Object> args) {
    SOAPEnvelope env = new SOAPEnvelope();
    for (Object arg : args) {
      //getObjectModelElementFromType actual marshals the argument
      //into an Axis 2.0 Object Model element that can be packed
      //into a SOAP envelope
      env.add(getObjectModelElementFromType(arg));
    }
    return env;
  }
}
```

Figure 3: Proxy code using static reflection

expressive nested patterns. Although MorphJ supports the Proxy design pattern well in a non-distributed setting, additional engineering is needed to effectively apply MorphJ to typical distributed proxies. Challenges that need to be overcome include:

1. **Reflection over the WSDL.** Our static reflection mechanism must reflect over the WSDL description. Since the WSDL description is in XML, either MorphJ must be extended to reflect over WSDL, or the WSDL must be converted to a Java interface so MorphJ can operate over it. And it is not clear how type safe such reflection would be.

2. **Treating method arguments individually.** The current version of MorphJ does not support treating method arguments individually at run time. To understand this better, consider the code in Figure 3 written in an imaginary extension of MorphJ. The pattern `<R,A*>[m] for(public R m(A) : T.methods)` basically selects all public methods in the type `T`. In the code block that follows we initialize the `OperationClient` using `m`. Currently in MorphJ, it is not possible to iterate over individual method arguments that are contained in `a` at runtime. In order to overcome this we added a utility `asList()` that returns individual method arguments in a `List` for the `marshal()` method. This gives us the ability of writing just one `marshal()` method and not have to overload it to match the signature of every method in type `T`[8].

But what about the type T? This type T is the representation of the WSDL description in our language. In Figure 5 (again the language is an imaginary extension of MorphJ),

[8]The types MessageContext and OperationClient are specific to the Axis 2.0 implementation and don't play any role in what we are trying to achieve here.

```
1   ObjectFactory factory = new ObjectFactory();
2   service = new BankServer();
3   BankServerPortType port =
4                   service.getBankServerHttpSoap12Endpoint();
5   CreditCard cc = factory.createCreditCard();
6   cc.setName(factory.createCreditCardName(card.getName()));
7   cc.setNumber(card.getNumber());
8   if (port.isValidCreditCard(cc)) {
9       port.chargeCreditCard(cc, order.getAmount);
10  } else {
11      // handle invalid credit card
12  }
```

Figure 4: Client code that calls the web service

```
1   type BankServer = load ‘‘http://www.bank.com/services/bankserver?wsdl”;
2   ClientProxy<BankServer> service =
3               new ClientProxy<BankServer>();
4
5   if (service.isValidCreditCard(card)) {
6       service.chargeCreditCard(card, order.getAmount());
7   } else {
8       // handle invalid credit card
9   }
```

Figure 5: Ideal client code to call the web service

line 1 creates a new type `BankServer` from the WSDL description at the given address. This type BankServer is then used to instantiate a ClientProxy.

2.3 Multiple data translations

In addition to code redundancy mentioned above, clients in a distributed system suffer from multiple data translations. Since the bookstore is a comprehensive application, it has its own internal `CreditCard` type. It uses this to persist credit card information in a user profile, so that a user doesn't have to input it each time they login to the system. This means that a translation must be made from the bookstore's `CreditCard` to the `CreditCard` the proxy expects. Figure 4 contains code in which the `ObjectFactory` is used to create an object of type `CreditCard`, but its name and number are set using an object `card`, which is of type `CreditCard` internal to the bookstore.

This translation of data is a problem because it has to be performed multiple times in the system and every time it is different enough that it can't be refactored into a method. For instance, if the bookstore deals with multiple banks then each bank publishes it own WSDL description and has its own type similar to `CreditCard`. To communicate with any of these banks a similar translation of data has to be done. We believe that proxy developers should not have to write translations of data and should be taken care of by the proxy as part of its marshalling mechanism. So ideally the code in Figure 4 should be as simple as described in Figure 5. Here the language takes care of making the translation between the bookstore's internal card object to the one that the proxy expects. One possible but simple implementation strategy is to statically compare the names and types of attributes.

2.4 X/O impedance mismatch

The X/O impedance mismatch has to mainly do with generation of Java code from XML schemas and vice versa. When designing web service interfaces can choose between the Java-to-WSDL or the WSDL-to-Java approach [5]. In the Java-to-WSDL approach, we start by creating Java interfaces for the web service and then the WSDL description by looking at these interfaces. The steps are reversed in the case of WSDL-to-Java. Existing tools (e.g. WTP, Netbeans) generate code in either direction but often due to the lack of a generic mapping between XML and Object types fail to capture design intent accurately.

Eric Meijer et al. [3] in their paper on X/O impedance mismatch describe a number of mismatches between XML and Object types. In our simplistic example, we choose to describe only one but important mismatch that occurs when proxies are automatically generated. The scenario under consideration is the automatic generation of the WSDL description on the server side of the web service. On the server we have a `CreditCard}` type in Java that has only one constructor that takes two parameters and no setters. The design intent here is to always have a well-defined `CreditCard` object. The WSDL in Figure 2 was generated and has both the elements `name` and `number` of the `CreditCard` complex type with attribute `minOccurs=0`. The `minOccurs` attribute specifies if the particular element is optional in valid XML document that conforms to this schema. This means that it is ok to send a message with an ill-formed `CreditCard` type. Checking the validity of the `CreditCard` object that is read from the SOAP message is now up to the service implementation. This is burdensome and can easily be neglected. We can see that the design intent was lost when we generated

the WSDL description from the Java class.

We could of course go the other way round, i.e. first create the WSDL description by hand and from that generate the Java code. This time we make the attribute `minOccurs=1` for both the `name` and `number` and set `nillable=false` for `name`. In this case too the design intent is lost because in the generated Java code this constraint is not enforced.

Both WSDL and Java have multiple ways in which the design constraint can be expressed but there is no obvious method to figure out one from the other. Technologies such as JAXB[9] for Java, and xsd.exe[10] for .NET that are current state of the art in industry, to our knowledge fail to capture this kind of information.

3. PLAUSIBLE LANGUAGE BASED SOLUTION

We feel that native support for proxies from the programming language can help alleviate the problems mentioned above. In this following points, we sketch the high-level requirements of such a language:

1. **Support type safe static reflection as a delegation mechanism**. We feel that static reflection enables us to write succinct and elegant proxies. In the example mentioned above we restricted ourselves to a WSDL description but ideally we should be able to do this over a remote interface described in any language. The language must have in built support for common interface description languages and protocols.

2. **Support for automatic translation of data**. Notice that in Figure 5 the arguments passed on lines 5 and 6 are of type `CreditCard` which is internal to the bookstore and not what the `ClientProxy` expects. In a conventional programming language this code will fail to typecheck. In a realistic setting, the data may not be as simple as the `CreditCard` type described here but a complicated tree data structure. The goal of the new language is to not only support such automatic translations but also guarantee type safety.

3. **Provide better mapping between XML and Object types**. [3] describes many problems that need to be overcome for better mapping between XML and Object types. In our new language we shun the idea of automatically transforming from XML to Objects and vice versa. Instead, we require the user to provide both the XML and Object models but leave the mapping between the two to the language. This approach provides the user with tremendous flexibility and the ability to capture the same design intent in both the data models. However, it requires a great level of sophistication from the language. The mapping algorithm could take a pragmatic approach and initially provide support for common programming idioms like the data integrity constraint we defined in our example.

[9]https://jaxb.dev.java.net/

[10]http://msdn.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx

4. CONCLUSION

In this paper, we have shown that despite the ubiquity of proxies in distributed systems they suffer from a number of software engineering problems. Prior research in the area and commercially available tools alleviate them to an extent but fail to address all issues. Finally, we sketched the high level requirements of a programming language that natively supports proxies for distributed system. This language supports type safe static reflection, automatic translation of data and flexible approach for capturing design intent in XML and Object types. These requirements pose implementation challenges, but when overcome can help us design more elegant proxies.

5. REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Shan Shan Huang and Yannis Smaragdakis. Class morphing: Expressive and safe static reflection. In *Conf. on Programming Language Design and Implementation (PLDI)*. ACM, Jun 2008.

[3] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). 06 June 2007. 80 pages. To appear.

[4] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass, United States, 1986. IEEE.

[5] Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, and Beth Stearns. *Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, XML Services, and Clients*. Pearson Education, 2004.