CODA

Coordination of Distributed Applications

The Last Phase in Distributed Computing Evolution

Prepared for COMS W4115 by Nirmal K Mukhi

Introduction

In the last two decades, the world has become more connected then ever, through a proliferation of communication and networking technologies that ensure you are never too far away to call home, check a stock quote or even bank online. To support this internetworked world, software applications have interact with each other over computer networks. Distributed communications have progressed from being arcane in the 1970s to commonplace in virtually every non-trivial software system today.

Programming technologies have fortunately made it easier to create distributed applications too, so that developers do not have to manually listen to sockets, sift through formatted binary messages and check 6-inch thick manuals for error code descriptions. Much of this responsibility has been taken over by middleware stacks that incorporate the low level mechanisms needed for distributed communication and associated paraphenalia such as security, transaction management, messaging guarantees, caching and so on. Applications developed on such middleware platforms are thus endowed with complex capabilities, while still allowing the developers to remain focused on the logic being implemented. While much work remains to be done, it is undeniable that distributed programming has evolved considerably. Following work done in the UNIX operating system and its derivatives, DCE and COM were the communication technologies used in the 1980s and early 90s but were still focused on C and C++. CORBA provided language independence in the mid 1990s. The emergence of Java and its easy to use RMI compiler, combined with the World Wide Web's entry on to the computing stage provided impetus for client-server computing. Following those efforts, the popularity of competing, complex middleware stacks (principally the J2EE and Microsoft .NET stacks) and the need to interoperate between them has led to the emergence of Web services.

As data flows across these vast networks between applications, there is a need for "Programming in the Large", i.e. a style of programming where the data being manipulated isn't at the level of bytes and the variables being created do not manipulate a counter. At this level, programming is about coordination of many applications, each of which is available on the network. It is about controlling data flows, managing messages that encompass a set of data, and synchronizing message exchanges between different applications. We will motivate the need for this style of programming with an example.

Consider an interaction between a flower shop and a gift shop. The gift shop allows customers to order flowers, using the flower shop as the supplier. When a customer makes an order, the gift shop checks the availability, makes the order and later receives confirmation of the delivery from the flower shop. Most software architects today, even using the latest language technologies, would begin solving this programming problem by defining contracts for each party through extensive documentation, and then translating those into code. This process is likely to be erroneous, as contracts are after all in a human readable form, and may have ambiguity, may be misinterpreted or may result in incorrect code due to carelessness. This problem is exacerbated when there are more than two parties involved in the interaction (for example: the flower shop uses the courier for delivery, the information about the customer is provided to the courier by the gift shop, via the flower shop, and delivery confirmation arrives at the gift shop from the courier). One of the main reasons this problem is hard is that each of the parties involved *independently* codes their portion of the ditributed interaction. The resulting applications are expected to work correctly if all programmers have followed the contracts faithfully, but this rarely happens and extensive testing is needed. Even then, there are no guarantees of correctness.

What is the most intuitive way to think about this problem? This interaction becomes clear when seen as a series of message exchanges between the different parties. The messages are produced and consumed, there is causality between message exchanges. Additionally, some message exchanges are conditional, some exchange patterns may occur repatedly and so on. We can specify these message exchanges not from any one party's viewpoint, but considering the interaction as a whole. With CODA, we aim to create a language that lets a new kind of programmer, an *application integrator*, to codify formally the coordination of distributed applications by specifying explicitly the message exchanges that are to take place, using the proper data and control flow. Such a CODA program is compiled into executable code for each of the parties involved in the interaction, with the guarantee that the code when executed will comply with the coordination specification described in the CODA program. That is, the messages will be exchanged as prescribed. Additionally, CODA will provide features that are intuitive in when describing such distributed program coordinations.

More on CODA

The first problem is that of how to model the applications that are being coordinated. Here, modeling the applications as services that follow Web services standards seems to be the natural choice. Web services are a set of middleware standards based on XML messaging, extensive use of standardised metadata and the Service-Oriented architectural pattern where dynamic discovery and binding are native capabilities. The popularity of the SOAP messaging format, widespread adoption of the technology and standardization by reputed parties such as the W3C has meant that it is likely to be the method of choice for encapsulating application functionality in the near future. As a result of this choice, we can assume that the applications being coordinated have their function described using WSDL (Web Service Description Language). This description encapsulates the data types, messages and operations offered by the service. Additionally, it includes details on how the services is to be accessed: how data is marshaled or unmarshaled into or from a wire format respectively, and where it is to be sent to or received from.

WSDL generally uses XML schema to define data types. These types are used to define messages. The interface that is implemented by the application is called a *PortType*, and is a collection of operations, each of which specifies what messages are expected as input and produced as output. Following our example, consider the description of the flower shop service below.

```
<?xml version="1.0" ?>

<definitions targetNamespace="flowershop://flower_purchase_service"
             xmlns:tns="flowershop://flower_purchase_service"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <!-- type definitions using XML schema -->

    <!-- tns:flowerset encompasses a set of items       -->
    <!-- available at the florist                       -->

    <!-- tns:order is the order type, which encompasses -->
    <!-- the flowerset being ordered, customer name,    -->
    <!-- delivery address and credit card info          -->

    <!-- tns:confirmation is the order ID               -->

    <!-- tns:status is a status type (encompasses 'ok'  -->
    <!-- and 'fail') states                            -->
  </types>

  <!-- message declns -->
  <message name="FlowerMessage">
    <part name="flower" element="tns:flowerset"/>
  </message>
  <message name="StatusMessage">
    <part name="result" element="tns:status"/>
  </message>
  <message name="OrderMessage">
    <part name="order" element="tns:order"/>
  </message>
  <message name="ConfirmationMessage">
    <part name="id" element="tns:confirmation"/>
  </message>

  <portType name="FlowerPortType">
    <operation name="checkAvailability">
      <input message="tns:FlowerMessage"/>
      <output message="tns:StatusMessage"/>
    </operation>
    <operation name="orderFlowers">
      <input message="tns:OrderMessage"/>
      <output message="tns:ConfirmationMessage"/>
    </operation>
  </portType>

  <binding../>

  <service../>

</definitions>
```

Here, we have left out schema type definitions. We have four message definitions. The first, *FlowerMessage*, encompasses the set of flowers being ordered. It consists of one part, of the appropriate schema type. The second message is a status message, with *ok* and *fail* being the possible data values. These messages are used in the *checkAvailability* operation within the *FlowerPortType* interface. The purpose of this operation is to check whether a possible order can be fulfilled by checking the availability of a set of flowers. The second operation in the interface allows the order to be placed, and returns a confirmation ID. The binding (messaging format and wire protocol details) and service (network endpoint details) elements are left out as these are not relevant to our discussion.

We also present the WSDL for the gift shop service below. This is extremely straightforward, as it offers just one operation, that allows the flower shop to report the delivery of flowers to the customer.

```
<?xml version="1.0" ?>

<definitions targetNamespace="giftshop://flower_shop_order_service"
            xmlns:tns="giftshop://flower_shop_order_service"
     xmlns:flowershop="flowershop://flower_purchase_service"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <!-- type definitions using XML schema -->
    <!-- tns:order is the order type, which encompasses -->
    <!-- the flowers being ordered, customer name,      -->
    <!-- delivery address and credit card info          -->
    <!-- tns:confirmation is the order ID               -->
  </types>

  <portType name="GiftShopPortType">
    <operation name="reportFlowersDelivered">
      <input message="flowershop:ConfirmationMessage"/>
    </operation>
  </portType>

  <binding../>

  <service../>

</definitions>
```

CODA Orchestrations

We sketch the CODA program that orchestrates these services below. The top level language construct is called an *Orchestration*. This may be *public* or *private*; public orchestrations may be extended by other programs. It begins by defining the two services, by specifying the locations of their WSDL definitions. The orcestration is specified as a set of dialogs. Each dialog is between two services. Within the dialog, we specify how messages are produced and exchanged.

Data in CODA

CODA defines and manipulates data only as messages and message fields. Note that since the CODA program itself is not executed, it never initializes any data; data is always produced by a service. We use `<--` to indicate transfer of a message from the right side of the expression (which may be a data source, such as a message, an operation that produces output, or a service) to the left side (a data sink, such as an operation that requires input, or a message). The language additionally allows data inspection (using XPath-like syntax to specify the field being inspected) and has comparison operators. Conditional statements allow decisions to be made so that message routing is not static. All data is scoped to the orchestration, so dialogs can refer to data produced in other dialogs.

Dialog semantics

Each dialog in a CODA program implies a sequential order or message production and consumption. For example, in our dialog below, message *m1* must be produced by the gift shop before it is provided as input for the *checkAvailability* operation of the flower shop service. Note however that an orchestration may consist of multiple dialogs. Each of these dialogs can be executed in parallel. Whenever there is a data dependency between dialogs (for example, we define dialog D2 that begins by inspecting message *m2*), there is an implied synchronization between them. In this case, the thread running dialog D2 would have to wait until message *m2* is available for consumption.

Extending orchestrations

CODA allows orcestrations to be extended, so that an existing orchestration may be augmented with an additional dialog. In our running example, if we decide to extend the flower shop - gift shop scenario by introducing the delivery courier, we can add new dialogs that specify the new message exchanges to be followed, recompile the code to produce new executable code for each party and then begin the orcestration.

```
public Orchestration
{
  // define the services involved
  Service flowerShop('http://flowershop.com/services/FlowerPurchase.wsdl');
  Service giftShop('http://flowers.com/services/wsdls/GiftShop.wsdl');

  public Dialog D1(flowerShop, giftShop)
  {
    // define the messages being exchanged
    Message m1{'flowershop://flower_purchase_service','FlowerMessage'};
    Message m2{'flowershop://flower_purchase_service','StatusMessage'};

    // flower set being checked is produced by the gift shop
    m1 <-- giftShop;
    // check with the flower shop if desired flowers are available
    m2 <-- flowerShop.checkAvailability <-- m1;

    if (m2.result == 'ok')
    {
      // flower are available, proceed with order
      // define messages used in order
      Message m3{'flowershop//flower_purchase_service','OrderMessage'};
      Message m4{'flowershop//flower_purchase_service','ConfirmationMessage'};
      // order message is produced by gift shop
      m3 <-- giftShop;
      // order the flowers and get the confirmatiom message
      m4 <-- flowerShop.orderFlowers <-- m3;

      // flower shop produces confirmation message to report delivery
      Message m5{'flowershop//flower_purchase_service','ConfirmationMessage'};
      // verify the confirmation ID is identical
      if (m4.id == m5.id)
      {
        // flower shop reports that flowers were delviered
        giftShop.reportFlowersDelivered <-- m5;
      }
    }
  }
}
```

Language goals

As we described earlier, CODA is designed for to be written by an *Application Integrator*. This is more likely to be a team of people than a single person, with more knowledge of business operations than computer science skills. Hence the main focus for CODA is on simplicity. The syntax and language constructs are thus designed to be intuitive for novice programmers.

The second goal for CODA is that it must produce executable java code for all parties in an orchestration that is indeed correct with respect to the CODA specification. This will reinforce the view that programming with a global view of the interaction is more sensible than each party locally programming their own applications, which was our motivation for this language.

Finally, CODA is designed to cover the most common use cases for distributed program coordination. Obviously, there may be complicated protocols or message exchange sequences that are not covered, but doing so at the cost of simplicity would defeat the purpose of the language.