e-speak

Service Framework Guide

Developer Release 3.01
June 2000

HEWLETT® PACKARD

Contents

Chapter 1 Service Bus Java Programmer's Guide 1

Introduction 1

Programming Model 3

Negotiation Library 18

Chapter 2 An End to End Example. .41

About The Example Application 41

Installing & Running The Example Application 46

Chapter 1 Service Bus Java Programmer's Guide

This chapter describes the E-speak *Service Bus package* for Java programmers. The e-speak service bus package `(net.espeak.servicebus)` is a Java implementation of the E-speak Service Framework Specification (SFS).

This document contains the following sections:

- Section 1 : Introduction: Describes the E-speak Service Framework Specification (SFS).
- Section 2 : Programming Model: Describes the interfaces provided by the net.espeak.servicebus package and how it allows development of SFS-compliant services in Java.
- Section 3: Matchmaker Service: Describes how a service-provider can advertise its service-offerings at a matchmaker and how service-consumers can look up service-offerings.
- Section 4: Negotiation Service: Describes how e-services can negotiate with each other.
- Section 5: End-to-End Example: Describes an example in which a paper buyer discovers a paper suplier and completes a business transaction over the Service Bus.

Introduction

The Internet has evolved from a platform for serving documents to a platform on which services, such as book-purchase or bill-payment are offered. Today, these services are largely implemented as standalone "backend" components and do not leverage other services offered over the Internet. Our vision is that the Internet will develop into an "e-service" platform where software services can directly interact with each other to provide richer functionality than each service can provide by itself. For example, businesses can achieve tighter supply chain integration by allowing their ERP systems to speak to each other.

The E-speak Service Framework

Unlike traditional client-server systems, e-service consumers are not directly under the service-provider's administrative control. Hence, service-providers cannot assume that service-consumers will use a common programming language, runtime platform, or that they can be trusted to behave well. In fact, the service-provider may not even know its consumer's identity until it is used. The E-speak framework allows services written in different languages and running on different platforms to interact with each other. It does this by using a document-exchange model of interaction where each message is an XML document. E-services are programmed to receive XML documents and to respond with XML documents and can be written without any dependence on a specific communication protocol. The communication protocol is configured at deployment-time; hence it is not a development-time decision.

The E-speak service framework uses the following terminology :

- Service : A service is an end-point that is connected to the service-bus and indicates a single entity from a trust perspective. Each service has a unqiue identifier (URI) that distinguishes it from other services and is addresible at a (protocol-dependent) specific URL.
- Conversation: Each service can support multiple conversations. A conversation is a logical group of document exchanges. These interactions may have been grouped together because they form a logical chain of interactions or because they all need the same trust assumptions. Conversations are analogous to interfaces in the object-oriented paradigm.
- Interaction: Each interaction represents a document that is sent across from one service to another and any responses that may be sent back. An interaction is similar to the notion of a method call in traditional object-oriented languages.

The E-speak service framework consists of two sets of specifications:

- Basic specifications : These are the basic set of specifications that are required for services to interact with each other.
    - The E-speak Message Schema specifies the low-level tags used to encode routing and contextual information in documents that are exchanged between services.
    - The E-speak Service Definition Schema specifies how the interface of a service can be represented in terms of the types of conversations it can participate in.
- Conversation Specifications: These are a set of standard conversations that are widely-used and hence provided along with the base set of specifications.
    - Matchmaking Conversation : Service-providers enter into this conversation with a Matchmaker service to advertise their service-offerings, while service-consumers enter into this conversation with a Matchmaker to lookup service-offerings that match their interest.
    - Negotiation Conversation : This conversation is supported by any two services that want to negotiate the values of any tag in an XML document. The negotiation conversation specifies the structure of each counter-offer document.
    - Management Conversation : This conversation takes place between a management station and the services that it manages.

Please refer to the E-speak Service Framework Specification document (http://www.e-speak.net) for the details on the above.

Programming Model

The primary goal of the E-speak service framework specification is to allow dynamic interaction and management of e-services over the Internet. The specification itself only defines the XML schemas for E-speak Messages and for a set of commonly used Conversations. The actual application logic for the services that carry out these Conversations may be written in any programming language. It is to be expected that for each programming language, there will be libraries that help the developer create and communicate XML documents. The `net.espeak.servicebus` package, henceforth referred to as the `servicebus` package, is such a class library for Java. This section describes the functionality provided by the ServiceBus package and illustrates its various aspects with a running example.

The ServiceBus package

The `servicebus` package is a set of Java classes that eases development and deployment of E-speak services. The functionality provided by the library can be categorized as follows :

- **Transport :** of XML documents to designated services using a variety of transport protocols.
- **Dispatching :** of incoming XML documents to the appropriate application logic.
- **State Management :** of conversations so that documents are interpreted in the right context.
- **XML utilities :** for converting Java objects into XML documents and vice-versa.

Besides the basic infrastracture, the service-bus also provides a set of conversation libraries that implement commonly-used conversations. These are :

- **Introspection Library :** to query other services about what conversations they support.
- **Negotiation Library :** that allows services to negotiate the fields of any arbitrary XML document.
- **Matchmaking Library :** to talk to a matchmaker that matches service-providers with consumers.
- **Management Library :** that allows a management station to manage a service and monitor its performance. *(Note: The Management library is not included in the current release.)*

Basic Concepts

ServiceBus

The `ServiceBus` is a singleton object that represents a connection to the underlying communication infrastructure. The `ServiceBus` has a set of transport adapters through which it sends and receives Messages. Services connect to the `ServiceBus` by binding a callback `MessageHandler`. The ServiceBus is instantiated by invoking its `init` method.

Message

`Messages` are documents that are exchanged between services. Each `Message` has two parts - its `Content` and its `Context`. The `Content` of each Message arriving at the ServiceBus is a Mime-encoded XML document.

Context

The `Context` of a message contains all information related to delivery and dispatch of Messages. To initate a conversation, a client must create a `Context` in which the first Message will be sent. Message recepients can look at the `Context` to determine the sender's identity. The `Context` class also provides functionality needed to send and receive Messages.

MessageHandler

`MessageHandlers` are callback objects that are bound to the `ServiceBus` and are called when an incoming message arrives for a specific service. A `MessageHandler` must implement a `handleMessage` method which is called by the ServiceBus whenever a message arrives. A `MessageHandler` may implement the actual application logic or may in turn dispatch the Message ot other higher-level `MessageHandlers`.

Container

`Containers` are `MessageHandlers` that receive Messages from lower layers and dispatch them to higher-level `MessageHandlers`. The `ServiceBus` is one example of a Container that dispatches Messages based on the name of the Service to which it is addressed. Besides dispatching, Containers also contain a `ContextFactory` that can be used to create `Contexts`. This feature is used by clients that want to initiate a conversation.

ServiceContainer

A `ServiceContainer` is another type of Container - it receives all Messages destined to a particular service, and dispatches them to different Handlers based on the Conversation type.

A Simple Example: Echo

This section uses a simple Echo program as its running example. The example consists of two parts - the EchoService that echoes back all incoming messages and an EchoClient that sends messages to it. In the first version of this example, described below, we assume that the EchoClient knows the URL of the EchoService.

EchoService Implementation

```
import net.espeak.servicebus.*;

/**
* Echo Service simply echoes back the incoming documents
*/
public class EchoService implements MessageHandler {

/**
* Simple message handler displays message and echoes it back
*/
public void handleMessage(Message msg) {
  try {
      System.out.println(" Received Message :" + msg.getBodyAsString());

      // Echo back message to the client
      Context context = msg.getContext();
      context.send(msg);
  } catch (Exception e) { System.out.println(e); }
}

/** Usage : java EchoService */
public static void main(String[] argv) throws Exception {
      ServiceBus serviceBus = ServiceBus.init();

      // Bind a EchoService object as the service : "EchoService".
```

```
          Binding b = serviceBus.bind("EchoService", new EchoService())

          // Print EchoService URL. Will be used by EchoClient.
          System.out.println("EchoService URL is : " + b.getLocalAddress());
     }

     }
```

EchoClient Implementation

```
import net.espeak.servicebus.*;
import java.net.URL;
import java.io.IOException;

/** EchoClient pings a "Hello World" Message from EchoService */
public class EchoClient {

/** Usage : java EchoClient <URL of EchoService> */
public static void main(String[] argv) throws Exception {
     if (argv < 1) {
        System.out.println("Usage: java EchoClient <URL>");
        System.exit(1);
     }
     URL echoServiceURL = new URL(argv[0]);
     ServiceBus serviceBus = ServiceBus.init();

     // Bind the "EchoClient" service with a ServiceContainer as the
     // MessageHandler. The ServiceContainer can create Contexts.
     ServiceContainer sc = new ServiceContainer();
     serviceBus.bind("EchoClient", sc);

     // Open a communication context with EchoService
     Context context = sc.createContext( echoServiceURL );

     // Construct Message
     MessageImpl mesg = new MessageImpl();
     mesg.setBody("<hello>Hello World</hello>");

     //Send the Message and block until a Message arrives.
     try {
          context.send(mesg);
          mesg = context.receive(); // blocking receive
     } catch (IOException e) {
        System.out.println("Transport Error"); throw e;
```

```
            }
            // Print out the Echoed Message.
            System.out.println("Echoed Message :" + mesg.getBodyAsString());
        }
    }
```

Conversations

The EchoService example shown above assumes EchoService responds in exactly the same way for every incoming document. In practice, a service may support mulitple types of conversations and do something different with incoming documents in each conversation. For example, EchoService may support an "EchoConversation" as well as a "ManagementConversation" simultaneously. In such cases, it is often easier to associate a different MessageHandler for each Conversation type.

The following classes in the Servicebus package are used to support Conversations :

ServiceContainer

`ServiceContainers` receive all Messages directed to a particular service and dispatch them to the appropriate MessageHandlers based on the Conversation name contained in the Message. This ensures that each MessageHandler will only be sent documents that belong to the Conversation type that it supports.

The ServiceContainer also allows binding a MessageHandler to a specific `MessageId`. This is used when the MessageHandler for a particular Conversation desires to delegate each Conversation instance to a different MessageHandler.

ConversationContainer

A `ConversationContainer` is a special type of MessageHandler that can be bound to a ServiceContainer. It receives all documents that belong to a specific Conversation and dispatches them to different Document handlers. Like other Containers, a ConversationContainer can be used to initiate Conversations by creating Contexts. ConversationContainers create `ConversationContexts` which are specialized to carry Messages that belong to a specific Conversation type.

Echo Example Revisited

The following code snippets show how the EchoService and EchoClient programs can be modified to support an EchoConversation.

New EchoService Implementation

The main method of the previous EchoService example must be changed to support an EchoConversation. A ServiceContainer bound to the ServiceBus receives all Messages that are sent to "EchoService". A ConversationContainer bound to the ServiceContainer receives all Messages that belong to the "EchoConversation". The handleMessage method remains the same.

```
public static void main(String[] argv) throws Exception {
     ServiceBus sbus = ServiceBus.init();

    // Bind a service container to receive messages for "EchoService"
    ServiceContainer sc = new ServiceContainer();
    sbus.bind("EchoService", serviceContainer);

    // Bind a conversation container to get messages for "EchoConversation"
    ConversationContainer cc = new ConversationContainer();
    sc.bind("EchoConversation", cc);

    // Set message handler to receive messages for EchoConversation
    cc.setDefaultMessageHandler(new EchoService());
}
```

New EchoClient Implementation

The EchoClient implementation is also changed. It now uses a ConversationContainer to create a ConversationContext. All Messages sent out in this Context will be tagged as belonging to an "EchoConversation". The following code snippet shows what the new implementation looks like:

```
// Create a serviceContainer and bind it to the ServiceBus, as before.
servicebus.bind("EchoClient", sc);

// Create container to receive conversation level messages
// Bind it to the serviceContainer
ConversationContainer cc = new ConversationContainer();
sc.bind("EchoConversation", conversationcontainer);
```

```
// Create context to start conversation with EchoService
Context context = cc.createContext(echoServiceURL);

// Construct a Message and send it, as before..
MessageImpl mesg = new MessageImpl();
mesg.setBody("<hello>Hello World</hello>");
context.send(mesg);
```

Conversation Executors : State Management

The executor layer allows services to associate a state-machine with their conversations and associate a specific document handler for each state. There are two XML files that need to be specified in order to conduct a conversation using an Executor. These are:

1. A state-machine file that defines all the possible states in a given conversation, and

2. A configuration file that specifies the handlers for each incoming documents in every state.

Essentially, conversation executors are sophisticated ConversationContainers. They are bound to a ServiceContainer and receive all Messages that belong to a specific Conversation. The conversation executor also keeps track of the current state of the conversation. When a new message comes in, the conversation executor updates its conversation state passes the incoming document to the handler that is appropriate in the current state.The new state of a conversation depends its previous state, the incoming Message and the ConversationContext of the incoming Message. If the incoming document is invalid in its current state, the Executor sends back an Error document to the sender.

When a new instance of a conversation executor is created, it is either the initiator of a conversation or it is responsible for responding to a particular conversation. Clients can initiate a conversation by invoking the Conversation Executor's `execute` operation.

The following classes are relevant :

ConversationHome

This class provides the mechanisms for creating executors. It accepts properties that contain the state diagram, the configuration descriptor, the name of the service and the name of the conversation. It then returns a `ConversationExecutor` object that is capable of executing the specified conversation. Like ServiceBus, the `ConversationHome` is a singleton object that is responsible for creating all `ConversationExecutors`.

ConversationExecutor

ConversationExecutors are created by the ConversationHome to handle a specific Conversation instance and are configured with a state-machine and with a set of document-handlers. The Conversation Executor automatically keeps track of the current state of an ongoing conversation and dispatches Messages to the appropriate Document Handler. To initiate a Conversation, programmers must call the `execute` method on a Conversation Executor.

The Introspection Library and Negotiation Library described next, illustrate the use of ConversationExecutors.

Introspection Library

Every E-speak service should support the Introspection conversation defined in the E-speak Service Framework specification. The introspection conversation is very simple - it specified that any service can be queried for its Properties by sending it a `getServiceProperty` document. Further details about the Service can be obtained by sending it a `getProcessDefinition` document.

The finite state machine for the introspection conversation has only one state that accepts both the `getServiceProperty` request as well as the `getProcessDefinition` request. When the service being introspection receives the `getServiceProperty` request, it responds with the a `ServiceProperty` document. When it receives a `getProcessDefinition` document, it responds with a `ProcessDefinition` document.

The following handler, uses a Conversation Executor to conduct the introspection conversation.

```
import net.espeak.servicebus.executor.*;
import java.util.Properties;
```

```
/** Illustrates a simple Server that can be introspected */
public class IntrospectionServer  {
    public static void main(String[] args) throws Exception {

     ConversationHome ch = ConversationHome.getConversationHome();

     Properties p = new Properties();
     p.put("FSM_FILE_NAME", "ProcessDefinition.xml");
     p.put("CONFIG_FILE_NAME", "ProcessConfig.xml");
     p.put("SERVICE_NAME", "introspection");
     p.put("CONV_TYPE", "introspection");

     ConversationExecutor ce = (ConversationExecutor) ch.bind(p);
    }
}
```

In the example above, the server essentially waits for introspection requests from the clients of this service. The finite state machine for the introspection conversation has only one state that accepts both the getServiceProperty request as well as the getProcessDefinition request. When the introspections server receives the GetServiceProperty request, it responds with the a ServiceProperty document. When the introspection server receives a GetProcessDefinition document, it responds with a ProcessDefinition document. This is formalized in a an XML file that looks as follows:

```
<ProcessDefinition uri="" type="introspection">
   <!-- document type declarations -->
  <DocumentType name="GetServiceProperty" id="GetServiceProperty"/>
  <DocumentType name="ServiceProperty" id="ServicePropertySheet"/>
  <DocumentType name="GetProcessDefinition"
id="GetProcessDefinition"/>
  <DocumentType name="ProcessDefinition"  id="ProcessDefinition"/>

  <!-- Interaction declarations -->
  <Interaction id="Interaction-GetServiceProperty"
xsi:type="RequestResponse">
```

```
    <Input> <DocumentType idref="GetServiceProperty"/> </Input>
    <Output> <DocumentType idref="ServiceProperty"/> </Output>
  </Interaction>
  <Interaction id="Interaction-GetProcessDefinition"
xsi:type="RequestResponse">
    <Input> <DocumentType idref="GetProcessDefinition"/> </Input>
    <Output> <DocumentType idref="ProcessDefinition"/> </Output>
  </Interaction>


  <StateMachine type="XMI">
    <!-- State machine definition in XMI -->
    <XMI version="1.1" >
      <XMI.header>
        <XMI.metamodel xmi.name="UML" href="some url"/>
      </XMI.header>
      <XMI.content>
        <UML:StateMachine>
          <UML:StateMachine.top>
            <UML:CompositeState isConcurrent="false">
              <UML:CompositeState.substate>
                <UML:PseudoState name="Start" kind="initial"
xmi.id="STATE-START">
                  <UML:StateVertex.outgoing>
                   <UML:Transition xmi.idref="TRANSITION-PROCDEF"/>
                  <UML:Transition xmi.idref="TRANSITION-SERVPROP"/>
                  </UML:StateVertex.outgoing>
                  <UML:StateVertex.incoming>
                   <UML:Transition xmi.idref="TRANSITION-PROCDEF"/>
                  <UML:Transition xmi.idref="TRANSITION-SERVPROP"/>
                  </UML:StateVertex.incoming>
                </UML:PseudoState>
              </UML:CompositeState.substate>
            </UML:CompositeState>
          </UML:StateMachine.top>
          <UML:StateMachine.transitions>
         <UML:transition name="Search" xmi.id="TRANSITION-PROCDEF">
```

```
              <XMI.extension xmi.extender="e-speak">
                <ES:Interaction
idref="Interaction-GetProcessDefinition"/>
              </XMI.extension>
            </UML:transition>
            <UML:transition name="Search"
xmi.id="TRANSITION-SERVPROP">
              <XMI.extension xmi.extender="e-speak">
                <ES:Interaction
idref="Interaction-GetServiceProperty"/>
              </XMI.extension>
            </UML:transition>
          </UML:StateMachine.transitions>
        </UML:StateMachine>
      </XMI.content>
    </XMI>
  </StateMachine>

</ProcessDefinition>
```

In addition to the definition of the finite state machine, the introspection server also has to set up the config file that determines the handler for each kind of document in any state. For the introspection request, the config file essentially states that a new instance of the introspection request handler can be spawned off to handle the introspection request. The XML for the config looks as follows:

```
<Config uri="http://www.books.com/config">
  <State name="Start">
    <config-list>
    <config>
    <Interaction id="Interaction-GetServiceProperty"
xsi:type="RequestResponse">
      <Input> <DocumentType idref="GetServiceProperty"/> </Input>
    </Interaction>
    <Handler-List>
    <Handler>
    <Detail language="java"
classname="net.espeak.essf.application.IntrospectionRequestHandler
" newinstance="yes" lifetime="call"/>
    </Handler>
    </Handler-List>
    </config>
    <config>
      <Interaction id="Interaction-GetProcessDefinition"
xsi:type="RequestResponse">
      <Input> <DocumentType idref="GetProcessDefinition"/> </Input>
  </Interaction>
    <Handler-List>
    <Handler>
    <Detail language="java"
classname="net.espeak.essf.application.IntrospectionRequestHandler
" newinstance="yes" lifetime="fsm"/>
    </Handler>
    </Handler-List>
    </config>
    </config-list>
  </State>
</Config>
```

The creator of the introspection service also implements the introspection request handler. The servicebus comes with a simple default implementation of the introspection service handler. This service handler responds to two XML messages, the GetServiceProperty message and the GetProcessDefinition message. It then replies with the ServiceProprerty document or the ProcessDefinition document. In the default implementation, the service property and process definition documents are expected to be in the directory where the service is running. In fact, the actual implementation of the introspection request handler looks as follows:

```
package net.espeak.servicebus.appl.introspection;

import net.espeak.util.xml.*;
import net.espeak.servicebus.*;
import net.espeak.servicebus.executor.*;
import java.io.FileInputStream;
import org.w3c.dom.Document;

public class IntrospectionRequestHandler implements MessageHandler {

    public IntrospectionRequestHandler()
    {
    }

    public void handleMessage(Message m) {
   try {
        Context context = m.getContext();
        System.out.println("Received introspection request");
        String fileName = null;
        Document d = m.getBodyAsDocument();
        String docType = d.getDocumentElement().getTagName();
        if(docType.equals(ConversationConstants.serviceProp)) {
      System.out.println("request for service property");
      fileName = "ServiceProperty.xml";
        }
        else {
      if(docType.equals(ConversationConstants.processDef)) {
          System.out.println("request for process definition");
          fileName = "ProcessDefinition.xml";
      }
        }
        AccessXml axml = new AccessXml(fileName);
        Document doc = axml.getDocument();
        Message repl = (Message) new MessageImpl();
        repl.setBody(doc);
        repl.setContext(context);
            context.send(repl);
         } catch (Exception e) { }
    }
}
```

A client can invoke the introspection service using either the simple servicebus level interfaces. For example:

```
import java.io.FileInputStream;
import java.net.URL;
import net.espeak.servicebus.*;
import net.espeak.util.xml.*;

import org.w3c.dom.*;

public class IntrospectionClient implements MessageHandler {
    private Context context = null;
   private IntrospectionClient(URL url) throws Exception {
   ServiceBus servicebus = ServiceBus.getServiceBus();
   ServiceContainer sc = new ServiceContainer();
   servicebus.bind("introspection", sc);
   ConversationContainer cc = new
ConversationContainer("introspection");
   sc.bind("introspection", cc);
   this.context = cc.getContextFactory().createContext(url, this);
    }

    public void send(String s)  {
   Message m = (Message) new MessageImpl();
   m.setBody(s);
   m.setContext(this.context);
        try {
            this.context.send(m);
        } catch (Exception e) { e.printStackTrace(); }
        System.out.println("IntrospectionClient does a send");
    }

    public void handleMessage(Message message) {
        this.context = message.getContext();
       AccessXml axml = new AccessXml(message.getBodyAsDocument());
        String s = axml.getText();
        System.out.println("Client receives message " + s);
    }

    public static void main(String[] args) throws Exception {
   if(args.length < 1) {
       System.out.println("provide name of get property file");
       return;
   }
   String port = "9000";
   URL toAddress = new URL("http://127.0.0.1:" + port
           + "/servlet/ServletAdapter?service=introspection");

   System.out.println("Connecting " + toAddress);
        IntrospectionClient client = new
IntrospectionClient(toAddress);
   AccessXml axml = new AccessXml(args[0]);
   Document doc = axml.getDocument();
   Element elem = doc.getDocumentElement();
   String text = axml.getText(elem, true);
   System.out.println("Sending: ");
   System.out.println(text);
        client.send(text);
    }
}
```

Negotiation Library

The service framework provides a means for services to attach any negotiators for the documents that the service may want to negotiate. Each negotiation conversation conducted by the service is conducted by a negotiator that keeps essentially acts like a conversation executor for the negotiation conversation. However, the negotiator can be set up so that it can it can negotiate the contents of any type of XML document. At this point in time, one can associate with each negotiator, a counter offer generator that is capable of generating counter offers. For example, suppose a book selling service wants to set up a negotiator for negotiating the prices of books. The counter offer generator may look as follows:

```
import org.w3c.dom.*;

import net.espeak.util.xml.*;
import net.espeak.servicebus.appl.negotiation.*;
public class Sellerco implements CounterOfferIntf {
    public Sellerco() {
   this.numRounds = 0;
   this.initPrice = 100.00;
    }
    public Document generateCounterOffer(Document offer) {
   if(this.acked) {
       System.out.println("Negotiation Over. Final doc is:");
       System.out.println(new AccessXml(offer).getText());
       System.exit(0);
   }
   try {
   this.numRounds++;
   String docType = offer.getDocumentElement().getTagName();
   if(docType.equals("NegotiationOffer")) {
       if(numRounds > 11) {
      // if negotiation goes on for too many rounds, we quit
      AccessXml axml = new AccessXml("nDisagree.xml");
      Document d = axml.getDocument();
      return d;
       }
       AccessXml bxml = new AccessXml(offer);
       String bxmlStr = bxml.getText();
       String priceStr =
"NegotiationOffer/NegotiationBody/books/price";
       String numberStr =
"NegotiationOffer/NegotiationBody/books/number";
       Node priceNode = bxml.getNode(priceStr, false);
       Node numberNode = bxml.getNode(numberStr, false);
       String priceVal = null;
       if(priceNode instanceof Element)
      priceVal = ((Element)priceNode).getAttribute("value");
       String numberVal = null;
```

```
          if(numberNode instanceof Element)
         numberVal = ((Element)numberNode).getAttribute("value");
          if(Double.valueOf(priceVal).doubleValue() > this.initPrice) {
         AccessXml cxml = new AccessXml("nAgree.xml");
         Document d = cxml.getDocument();
         return d;
          }
          else {
         ((Element)priceNode).setAttribute("value", "100.00");
         return offer;
          }
      }
      if(docType.equals("NegotiationAgreement")) {

          System.out.println("Negotiation successful");
          System.exit(0);
          // if other party agrees, we ack
      }
      if(docType.equals("NegotiationDisagreement")) {
          System.out.println("Negotiation successful");
          System.exit(0);
          // if other party disagrees, we ack.
      }
      }
      catch (java.io.IOException ioe) {
          ioe.printStackTrace();
      }
      return null;
       }
       private int numRounds;
      private double initPrice;
       private boolean acked = false;

}
```

Essentially, the counter offer generating service attempts to get at least 100 units for the book. It drops the negotiating when the number of rounds has exceeded 11.

The server for the negotiation looks as follows:

```
import net.espeak.servicebus.appl.negotiation.*;
import java.util.Hashtable;
import java.util.Properties;
public class Seller {
    public static void main(String [] args) {
   Properties sysProps = System.getProperties();
   sysProps.put("net.espeak.servicebus.httpservlet.portnum",
          "9000");
   Hashtable h = new Hashtable();
   h.put("books", "Sellerco");
   NegotiatorFactory.setCounterTable(h);
   Negotiator n = new Negotiator();
   n.setCounterOfferGenerator("books", (CounterOfferIntf) new
Sellerco());
    }
}
```

Now, a client can invoke the negotiation service and negotiate the contents of a books document. It can do this by either programming to the negotiation interface, or it can program directly to the service bus interface. For instance, the client below interacts with the negotiation server described above, sends offers and responds to counter offers.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.net.URL;
import net.espeak.servicebus.*;
import net.espeak.util.xml.*;

import org.w3c.dom.*;


public class NegotiationClient implements MessageHandler {
    private Context context = null;
    private NegotiationClient(URL url) throws Exception {
   ServiceBus servicebus = ServiceBus.getServiceBus();
   ServiceContainer sc = new ServiceContainer();
   servicebus.bind("negotiation", sc);
   ConversationContainer cc = new
ConversationContainer("negotiation");
   sc.bind("negotiation", cc);
   this.context = cc.getContextFactory().createContext(url, this);
   this.numRounds = 0;
   this.initPrice = 50.00;
   this.maxPrice = 110.00;
   this.increment = 35.00;

    }

    public void send(String s)  {
   Message m = (Message) new MessageImpl();
   m.setBody(s);
   m.setContext(this.context);
         try {
             this.context.send(m);
         } catch (Exception e) { e.printStackTrace(); }
         System.out.println("Negotiation does a send");
    }

    public void handleMessage(Message message) {
         this.context = message.getContext();
   try {
       Document d =
generateCounterOffer(message.getBodyAsDocument());

   Message m = (Message) new MessageImpl();
   m.setContext(this.context);
   m.setBody(d);
   this.context.send(m);
   }
   catch(IOException ioe) {}
```

```
    }

    public Document generateCounterOffer(Document offer) throws
java.io.IOException {
   this.numRounds++;
   String docType = offer.getDocumentElement().getTagName();
   if(docType.equals("NegotiationOffer")) {
        if(numRounds > 11) {
      // if negotiation goes on for too many rounds, we quit
      AccessXml axml = new AccessXml("nDisagree.xml");
      Document d = axml.getDocument();
      return d;
        }
        AccessXml bxml = new AccessXml(offer);
        String priceStr =
"NegotiationOffer/NegotiationBody/books/price";
        String numberStr =
"NegotiationOffer/NegotiationBody/books/number";
        Node priceNode = bxml.getNode(priceStr, false);
        Node numberNode = bxml.getNode(numberStr, false);
        String priceVal = null;
        if(priceNode instanceof Element)
      priceVal = ((Element)priceNode).getAttribute("value");
        String numberVal = null;
        if(numberNode instanceof Element)
      numberVal = ((Element)numberNode).getAttribute("value");
        if(Double.valueOf(priceVal).doubleValue() > this.maxPrice) {
      AccessXml cxml = new AccessXml("nDisagree.xml");
      Document d = cxml.getDocument();
      return d;
        }
        else {
      double nextPrice = this.initPrice +
(numRounds*this.increment);
      Double nextP = new Double(nextPrice);
      ((Element)priceNode).setAttribute("value", nextP.toString());
      return offer;
    }
}
if(docType.equals("NegotiationAgreement")) {
    System.out.println("Negotiation successful");
    System.exit(0);
    // if other party agrees, we ack
}
if(docType.equals("NegotiationDisagreement")) {
    System.out.println("Negotiation unsuccessful");
    System.exit(0);
    // if other party disagrees, we ack.
}
return null;
  }

public static void main(String[] args) throws Exception {
if(args.length < 1) {
    System.out.println("provide name of get property file");
    return;
}
String port = "9000";
URL toAddress = new URL("http://127.0.0.1:" + port
        + "/servlet/ServletAdapter?service=negotiation");
System.out.println("Connecting " + toAddress);
     NegotiationClient client = new NegotiationClient(toAddress);
AccessXml axml = new AccessXml(args[0]);
Document doc = axml.getDocument();
Element elem = doc.getDocumentElement();
String text = axml.getText(elem, true);
System.out.println("Sending: ");
System.out.println(text);
      client.send(text);
  }
 private int numRounds;
private double initPrice;
private double maxPrice;
private double increment;
```

```
}
```

The MatchMaker service is a registry where service-providers can describe their service offerings and service-consumers can lookup them up. In particular the Matchmaker service provides a means for:

- Creating vocabularies that specify the schema for offer descriptions.
- Accepting service offers in different vocabularies.
- Revoking existing offers.
- Looking up service-offers.

Creating Vocabularies

A vocabulary specifies a set of attributes that can be used to describe an offer. Each attribute can be refered to by its attribute-name and can contain data of some specific type. The MatchMakerService comes pre-loaded with a vocabulary called **"MatchMakerVocab"**, that defines only attribute for offer descriptions: **"Service-Name"** which is of type "String". In order to register a service description in the MatchMakerVocab, the service has to send an XML document with two elements, the offer description element ,which would specify the offer in terms of the Match-MakerVocab and the value for the attribute "ServiceName" , and the owner element which would specify the contact information .The clients that do a lookup would try to contact the service at the address specified in this element.

To create a new vocabulary, a client can send a <CreateVocabulary> document to the Matchmaker. The schema of the <CreateVocabulary> document is specified in Appendix X.0 . The following example shows the document sent to the Matchmaker requesting it to create a car-dealer vocabulary. The vocabulary has attributes such as manufacturer, model and price that refer to the appropriate attributes of the cars that the car-dealer sells.

```
<create-vocabulary>
<resource
xmlns="http://www.e-speak.net/Schema/E-speak.register.xsd" >
   <resourceDes>
```

```
<vocabulary>http://www.e-speak.net/Schema/E-speak.base.xsd</vocabu
lary>
        <attr name="Name">
            <value>car-dealer </value>
        </attr>
        <attr name="Type">
             <value>Vocabulary</value>
        </attr>
    </resourceDes>
    <attrGroup name="Simple Car dealer vocabulary"

xmlns="http://www.e-speak.net/Schema/E-speak.vocab.xsd">
         <attrDecl name="Manufacturer" required="true">
               <datatypeRef name="string"/>
         </attrDecl>
         <attrDecl name="Model" required="true">
               <datatypeRef name="string"/>
         </attrDecl>
           <attrDecl name="Price" required="false">
               <datatypeRef name="float">
                  <default>0.0</default>
                  <minInclusive>0.0</minInclusive>
                  <maxInclusive>100000.0</maxInclusive>
               </datatypeRef>
         </attrDecl>
    </attrGroup>
</resource>
</create-vocabulary>
```

Suppose the above XML was in a file named createvocab.xml. The call in a java program would look as follows:

```
MMDiscoveryIntf agent = null;
String matchMakerLocator =
new String("http://rgelpc063.rgv.hp.com:9099/servlet” +
        "/ServletAdapter?service=MatchMakerService");
agent = new MMDiscoveryIntf(matchMakerLocator);
Document reply = agent.createVocabulary("createvocab.xml");

If the request Document was created in memory by the client
application (an object of type org.w3c.dom.Document) the
createVocabulary call would be

Document createVocabRequestDocument ;//in-memory document created by
the client application
Document reply =
agent.createVocabulary(createVocabRequestDocument);
```

For the BankVocabulary example shown above, the response document would look as follows :

```
<create-vocab-reply>
<status>
success
</status>
</create-vocab-reply>
```

Submitting Offers to a MatchMaker

A Matchmaker service accepts offers with descriptions that correspond to any vocabulary that has been registered with the matchmaker. An offer consists of information that can be used by clients of the matchmaker to search for the offer. Once an offer is discovered, the public portions of the offer can be returned to the entity performing the lookup.

For example, if a car-dealer wants to make an offer in both these vocabularies (both the vocabularies should be present in the MatchMaker repository) the offer-description would look like this

```
<offer>
<offer-description>
```

```
      <Car-Dealer>
             <Manufacturer>Honda</Manufacturer>
             <Model>Accord</Model>
             <Price>20000</Price>
      </Car-Dealer>
      <Automobile-Seller>
              <TypeOfAutomobile>CAR</TypeOfAutomobile>
             <MakeOfAutomobile>Honda</MakeOfAutomobile>
             <Model>Accord</Model>
             <Price>20000<Price>
     </Automobile-Seller>
</offer-description>
<owner>
      <url>”http://www.myservice.com:8080”</url>
</owner>
</offer>
```

Each child element of offer-description refers to a vocabulary name and the contents of the element have to conform to the schema laid out by the vocabulary. In addition to the offer description, the offer must also have the owner element. This element describes the contact point for an entity that is making the offer. Essentially, it is a URL or ESURL that can be used to send messages to the offerer.

The java code for such an invocation looks as follows:

```
MMDiscoveryIntf agent = null;
String matchMakerLocator =
new String("http://rgelpc063.rgv.hp.com:9099/servlet” +
        “/ServletAdapter?service=MatchMakerService");
agent = new MMDiscoveryIntf(matchMakerLocator);
Document reply = agent.makeOffer("offer.xml");
```

Structure of lookup

The matchmaker also supports the notion of lookups. Clients can submit queries that are matched up against the offers that get submitted to it. The queries have to be in the vocabulary that is registered with the matchmaker. For example, suppose a client wants to look for an offer using car-dealer vocabulary for Honda Accord cars that are priced less that 25000. The XML for the query looks as follows:

```
<lookup>
<esquery>
 <from src='url://testclient:88'/>
<where>
   <vocabulary name="car-dealer"/>
   <condition>ManufacturerName=&apos;Honda&apos;and
Model=&apos;Accord&apos; and Price &lt; 25000.00</condition>
</where>
</esquery>
</lookup>
```

Currently, the result of lookup is a list of owners of the offers that match the query. Therefore, the XML document that the client qill receive as a result of sending the above lookup request looks as follows:

```
<lookup-reply>
<status>
success
</status>
<url>"http://www.myservice.com:8080"</url>
</lookup-reply>
```

The sequence of java calls that achieves the above document exchange looks as follows:

```
String matchMakerLocator =
new String("http://rgelpc063.rgv.hp.com:9099/servlet" +
         "/ServletAdapter?service=MatchMakerService");
agent = new MMDiscoveryIntf(matchMakerLocator);
Document reply = agent.lookup("lookup.xml");
```

Structure of revoke-offer

The matchmaker accepts requests for revoke the offers created bythe services. These requests must contain the “offer-id” that was returned as a reply while creating the offer. A typical revoke-offer request look like this :

```
<revoke-offer>
  <offer-ID>
tcp://rgelpc063.rgv.hp.com:12346/proc/resource/NameFrame/75/home41
1bed8ee1502d37
  </offer-ID>
</revoke-offer>
```

The above request results in this reply if the request was executed successfully

```
<revoke-offer-reply>
  <offer>
    <offer-ID>
tcp://rgelpc063.rgv.hp.com:12346/proc/resource/NameFrame/75/home/4
11bed8ee1502d37
    </offer-ID>
    <status>success</status>
  </offer>
</revoke-offer-reply>
```

The sequence of java calls that achieves the above document exchange looks as follows:

```
String matchMakerLocator =
new String("http://rgelpc063.rgv.hp.com:9099/servlet” +
          “/ServletAdapter?service=MatchMakerService");
agent = new MMDiscoveryIntf(matchMakerLocator);
//if the request document is stored in a file called revokeoffer.xml
Document reply = agent.lookup("revokeoffer.xml");
//or if the request is in memory
Document revokeOfferRequest;//constructed in the memory
Document reply = agent.lookup(revokeOfferRequest);
```

Class MMDiscoveryIntf

MMDiscoveryIntf is the interface class for the MatchMaker which is used by all services to create offers or to do lookups for other services.

The services can use one of the following two constructors for instantiating it

- MMDiscoveryIntf( ) - the default constructor which uses the value of the system property

net.espeak.servicebus.matchmaker.url as the location URL for the MatchMakerService . This property is loaded by default by the ServiceBus infrastucture when **ServiceBus.getServiceBus( )** is invoked.

- MMDiscoveryIntf (String matchMakerLocation) - uses the parameter passed as the

MatchMakerService location.

Once this object has been instantiated , methods can be invoked upon it to talk to the MatchMakerService ,as shown in the examples above.

Class SampleMatchMakerRequest

This is a reference class that provides sample methods for creating request documents and parsing the reply documents that are interexchanged with the MatchMakerService. This class uses the MatchMakerVocab as the vocabulary for all requests.

Echo example revisited

Consider a simple example of a service that echoes the messages that are sent to it. This example just illustrates how a service is supposed to be set up. This simple service receives XML documents, converts them into objects and sends the object back in a reply message. On its startup , this service first creates an offer for itself with the MatchMakerSerivice . This offer is based upon the vocabulary called MatchMakerVocab and the attribute **ServiceName** is set to **EchoService**. The client (called EchoClient) does a lookup at the MatchMaker quering for offers under MatchMakerVocab where the attribute **ServiceName==EchoService** .The contact information (for the owner element) is obtained from the call `Binding.getLocalAddress().toString()`, binding is returned when a serivce does a bind() with the

`ServiceBus`.The MatchMakerService returns the lookup results , which is a list of contact URLs of the matched services, to the client. The client then tries to send a message to the URLs returned.

```
package matchmaker.echo;
import net.espeak.servicebus.*;
import net.espeak.servicebus.mminterface.MMDiscoveryIntf;
import net.espeak.servicebus.mminterface.SampleMatchMakerRequest;
import sun.servlet.http.HttpServer;
import org.w3c.dom.*;


public class EchoService implements MessageHandler {

    /**
    * The MatchMaker agent  this should be instantited after loading 
the properties.
     */
    private static MMDiscoveryIntf theMatchMakerAgent;

    /**
     * initializes the MMDiscoveryIntf object
     * This method should be called after loading the 
servicebus.properties
     */
    public static void initMatchMakerAgent()
    throws Exception{
           theMatchMakerAgent =new MMDiscoveryIntf();
    }

    /**
     * Message handler to process messages send to echoservice.
    * This simple message handler displays message on the screen and
     * echos the request back to the client.
     * @param msg
     */
    public void handleMessage(Message msg) {
        try {
            Context context = msg.getContext();
            // Echo back message to the client
            context.send(msg);
            } catch (Exception e) {
                System.out.println(e);
        }
    }

    /**
    * @param argv - This param if specified would override the System
property servicebus.propertyfile and would be used by the servicebus
infrastructure to load the properties.
*/
    public static void main(String[] argv) {
        try{

            // Get service bus
            ServiceBus serviceBus =null;
            if (argv.length > 0){
              ServiceBus.init(argv[0]);
            }

            serviceBus= ServiceBus.getServiceBus();
            initMatchMakerAgent();

            // Register EchoService with message handler
            Binding binding = serviceBus.bind("EchoService", new
EchoService());

            //create an offer document , the contact address (for
the owner element) is obtained by the
binding.getLocalAddress().toString() call.
            Document
offerDocument=SampleMatchMakerRequest.createMakeOfferRequest(bindi
ng.getLocalAddress().toString(),"EchoService");
```

```
            //send a makeOffer request
            Document
reply=theMatchMakerAgent.makeOffer(offerDocument);

            //parse the reply document
            String
offerid=SampleMatchMakerRequest.parseMakeOfferReply(reply);

            if (offerid !=null && !(offerid.equals("-1"))){
                System.out.println("Successfully created an offer
for EchoService with an offer -id of");
                System.out.println(offerid);
            }
            else {
             System.out.println("Can't create an Offer ..exiting");
                System.exit(0);

               }
        }catch(Exception e){
             System.out.println("Caught an exception
..."+e.getMessage());
             System.exit(0);
            }
       } //end of main
}//end of EchoService
```

The Client side code looks like ::

```
package matchmaker.echo;
import net.espeak.servicebus.*;
import net.espeak.servicebus.util.*;
import net.espeak.servicebus.mminterface.MMDiscoveryIntf;
import net.espeak.servicebus.mminterface.SampleMatchMakerRequest;

import java.util.Hashtable;
import java.util.Vector;
import java.net.URL;
import java.io.IOException;
import org.w3c.dom.*;

public class EchoClient {

   /**
    * The MatchMaker agent  this should be instantited after loading
the properties.
     */
    private static MMDiscoveryIntf theMatchMakerAgent;

    /**
     * initializes the MMDiscoveryIntf object
     * This method should be called after loading the
servicebus.properties  as it needs the matchmaker url .
     */
    public static void initMatchMakerAgent()
    throws Exception{
           theMatchMakerAgent =new MMDiscoveryIntf();
    }

    /**
     * the main method for EchoClient
     *
     * @param argv - optional param - The property file which would
override the System Property servicebus.propertyfile
     *                              The property file must contain
entries for these 2 properties
     *                                 -
net.espeak.servicebus.httpservlet.portnum = the listner port for
this application
    *                       - net.espeak.servicebus.matchmaker.url
= the url of the MatchMakerService
     */
    public static void main(String[] argv) {
        try {
            // Get service bus
            ServiceBus servicebus =null;

           //ServiceBus.init loads the propertyfile - if specified
            if (argv.length > 0)
             ServiceBus.init(argv[0]);

            servicebus= ServiceBus.getServiceBus();

            //initializes the MMDiscoveryIntf that talks to the
MatchMaker Service
            initMatchMakerAgent();

            // Create service container
            ServiceContainer sc = new ServiceContainer();

            // Register EchoClient with service container.
            Binding binding = servicebus.bind("EchoClient", sc);

            //create the lookup request
            Document
request=SampleMatchMakerRequest.createLookupRequest("EchoService")
;

            //send a lookup request
            Document reply=theMatchMakerAgent.lookup(request);

           //parse the reply sent by the MatchMaker , exit if there
is no match found.
            Vector
listOfUrls=SampleMatchMakerRequest.parseLookupReply(reply);
            String echoService=null;
            if (listOfUrls.size() <=0){
              System.out.println("No Match Found for EchoService");
                System.out.println("End of echo sample !");
                System.out.println("Press Ctrl-C to end program");
                return;
            }
            else{
                //try to talk to the first URL of the list returned
                echoService=(String)listOfUrls.elementAt(0);
            }
            // Convert echoservice string to URL
            URL toAddress = null;

            try {
                toAddress = new URL(echoService);
            } catch (java.net.MalformedURLException e) {
              System.out.println("Echo service address is not valid
URL");
            }


          // Note:: Use binding.getLocalAddress to get your address.
            // Create new context to connect with echo service
            Context context =
sc.getContextFactory().createContext(toAddress);
            MessageImpl mesg = new MessageImpl();

            // Create hello world message
            mesg.setBody("<hello>Hello World</hello>");
            try {
                context.send(mesg);            // send
            } catch (IOException e) {
                System.out.println("Could not send message");
            }
            Message m = context.receive();    // blocking receive

           System.out.println("Received message in Echo Client - ");
            System.out.println(m.getBodyAsString());
            System.out.println("End of echo sample !");
            System.out.println("Press Ctrl-C to end program");
        } catch (Exception e) {
            e.printStackTrace();
            }
        }
    }
```

Running the MatchMakerService

Setting the Classpath

The following items need to be in the classpath for running the MatchMakerService

- xerces.jar
- jsdk.jar
- cryptix.jar (This file is currently under $ESPEAK_HOME$/extern/cryptix)
- $ESPEAK_HOME$/lib
- $ESPEAK_HOME$/contrib/lib

It is important to keep xerces.jar at the beginning of your CLASSPATH to avoid any conflicts with other DOM parsers that may be in your CLASSPATH.

Setting the Path

The $ESPEAK_HOME$/lib should be in the $PATH . This is needed for the eccStubs.dll.

Setting the Configuration Parmeters

The following needs to be set for the ESpeak-core instance :

In the config file **espeak.cfg ,** under the Security Properties section

- set **net.espeak.security.activate=off**

The MatchMakerService uses the following two system properties

net.espeak.servicebus.htppservlet.portnum -to start the listener to listen to requests

net.espeak.servicebus.matchmaker.backendagent - the backend repository(agent) , currently the only value supported for this property is **"escore"** which represents an espeak-core.

net.espeak.servicebus.matchmaker.backendhost - the host where the backend repository (espeak-core) is running.This attribute needs to be the full name of the host (name as well as domain) even if the core is running on the same machine (e.g. instead of localhost , specify

myMachineName.myDomain.com)

net.espeak.servicebus.matchmaker.backendport - the port on which the backend

repository(espeak -core) is listening.

net.espeak.servicebus.matchmaker.backendprotocol - protocol used to talk to the backendagent currently the only value supported for this property is **"tcp".**

these properties are defined in a file called "**matchmaker.properties**" which the MatchMakerSerivce tries to load **from the current directory** ,if no argument is specified while running it. This file has to be in the current directory from where the MatchMakerService is started , otherwise the full path of the propertyfile has to be passed as an argument. This file is currently located in <espeak_home>/contrib/lib/samples/matchmaker

Running the Program

Before starting the MatchMakerService make sure the ESpeak core is running with security turned off and the matchmake property file reflects the correct settings for the same.

The MatchMakerService canbe started by the following command

<JRE> net.espeak.servicebus.matchmaker.MatchMakerService [property-file-path]

where property-file-path is an optional parameter which specifies the path of the propertyfile that overrides the default property-file "matchmaker.properties".

Chapter 2 An End to End Example

This chapter shows an example application that utilizes the E-speak Service Bus Interface (S-Bus). The chapter contains the following sections:

- *About The Example Application:* Its design and the business workflow it performs
- *Installing & Running The Example Application:* What software components you need in order to run the example and how to configure them.
- *Where To Go To Download The Example:* The URL and instructions for downloading the example application.

About The Example Application

The example application is a web-based demonstration of a "Buy Paper" conversation. There are three participants in the conversation: the Paper Buyer, the Paper Seller, and the MatchMaker.

Both the Paper Buyer and the Paper Seller participants are full-fledged web applications. They each are implemented using Java Web Server technologies -- Java Server Pages, HTML, JavaBeans, and Servlets. The MatchMaker is an external service that these two applications use to discover each other.

The Business Process

The example application demonstrates a complete business transaction. This transaction involves a number of steps -- each of which involves an exchange of XML documents. The individual steps in the business transaction can be grouped into 4 phases:

Phase 1 “Discovery” – During this phase, the Paper Buyer posts an RFQ (request to purchase paper) to the MatchMaker. The MatchMaker then remembers that the Paper Buyer would like to purchase paper. After the Paper Buyer has posted its RFQ, the Paper Seller application queries the MatchMaker for any potential customers who have posted RFQ’s for paper. In response to the query, the MatchMaker sends the Paper Seller the URL for the Paper Buyer. (From this point forward, the MatchMaker drops out of the conversation and the Paper Seller and Paper Buyer can communicate directly.)

Phase 2 “Price Negotiation” -- During this phase the Paper Buyer and the Paper Seller negotiate a price for the paper. The first step involves the Paper Seller sending the Paper Buyer a “quote.” The Paper Buyer then responds with a “proposed purchase order” based upon the quote and any discounts the Paper Buyer is attempting to obtain. The Paper Seller then either agrees to the Paper Buyer’s proposal or sends a new quote to the Paper Buyer. These steps repeat until the Paper Seller accepts a “proposed purchase order” sent by the Paper Buyer.

Phase 3 “Forming A Contract” -- Once the Paper Buyer and Paper Seller have agreed on a price, the Paper Seller sends the Paper Buyer a digitally signed “contract.” The Buyer then acknowledges the contract.

Phase 4 “Invoicing, Payment & Fulfillment” -- After the contract has been formed between the Paper Buyer and the Paper Seller, The Paper Seller fulfills the order and ships the paper and sends shipping documentation on to the Paper Buyer. Then the Paper Seller sends an “invoice” to the Paper Buyer. The Paper Buyer processes the invoice and sends a “Payment” document to the Paper Seller. The Paper Seller then processes the payment and finally sends a “receipt” document back to the Paper Buyer.

Application Design

Both the Paper Buyer and Paper Seller applications are constructed using the same layered architecture. The architecture consists of these parts:

ServiceBus – This “bus” provides the mechanism by which the Paper Buyer and Paper Seller applications communicate with one another as well as with the MatchMaker. (The ServiceBus is discussed in detail in Chapter 1.*)*

MessageHandler (called "DocumentAdapter" in these applications) -- Each of the applications have a single instance of this component. The DocumentAdapter receives all incoming messages that were received from the ServiceBus. It then uses Java "reflection" and a mapping between XML Element names and document handler functions (there is one document handler function for each different XML document expected by the application) to invoke the appropriate function on the Application Object -- passing the received message as a parameter to the function.

Application Object -- Each of the applications has a single instance of this component. The Application Object contains a reference to a Workflow Object and maintains the current "state" of the application. It also provides a document handler function for each XML document it expects to receive. The Application Object also contains an "action" function for each web FORM that it expects to receive from the browser. The document handler functions and the "action" functions are all "pass through" calls into the Workflow Object -- they let the Workflow Object decide what is the "current" step and then invoke that step's business logic.

Workflow Object -- Each Application Object (and thus, each application) contains one of these components. This component implements the Finite State Machine that describes the application's business workflow. Each state in the workflow is represented by a WorkflowStep object. Each WorkflowStep object contains the business logic for a single step in the workflow.

Workflow Step -- The Workflow Object contains of a collection of WorkflowStep objects -- one for each StateVertex in the Finite State Machine for the business transaction. Each of these steps contains the business logic for a single step in the workflow.

JavaServerPages & HTML -- The user interface portion of each application is comprised of primarily Java Server Pages and a few HTML pages.

Diagram: XML Document Being Delivered To An Application

This diagram shows the flow of data from the ServiceBus into the application and finally to the web browser.

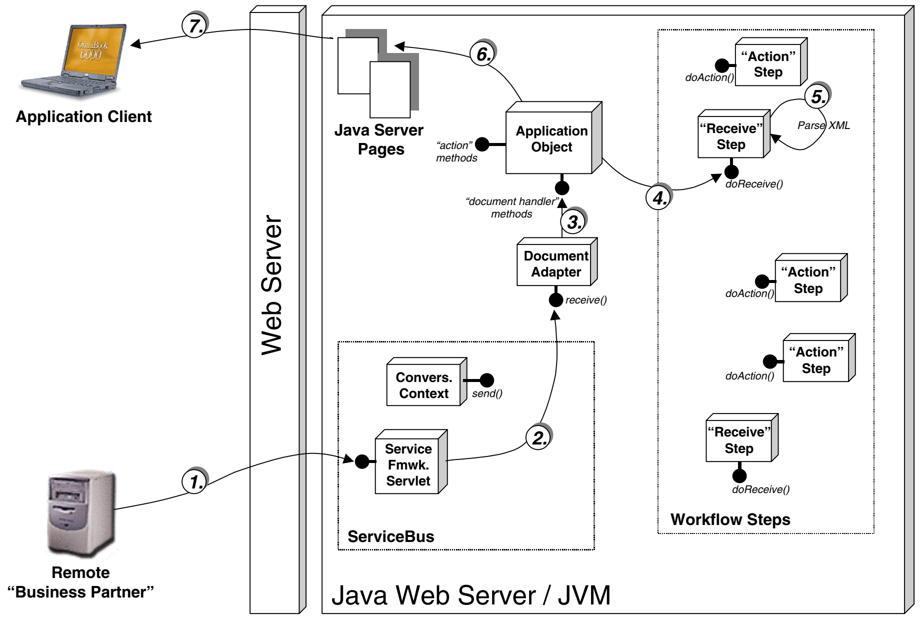
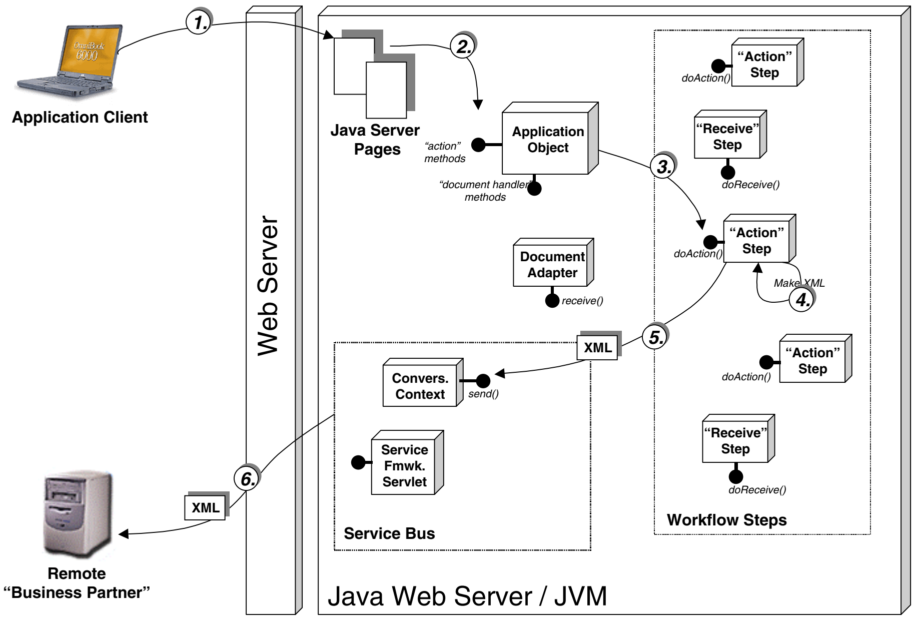


Diagram: Application Sends An XML Document

This diagram shows the flow of data from the web browser, into the application, and then onto the ServiceBus.



Installing & Running The Example Application

This Example has been tested on two web server platforms running under Windows NT 4.0. The two platforms are:

- Javasoft's reference implementation of the Java 2 Enterprise Edition environment
- Apache and the Tomcat 3.0

This section discusses installing and configuring the Example under both environments.

Compiling The Example

If you have the ".java" files for the application (these will be placed into the java web server's "classes" subdirectory) but not the ".class" files, compile them using the JDK 1.2.2 or later.

When you compile the source, make sure you have included in your class path

- jar file(s) containing the "javax.servlet" and "javax.servlet.http" packages (Servlet API).
- essf.jar file and/or classpath containg net.espeak.essf.* pcakages i.e., Service Framework (ESSF) implementation
- webmacro.jar file (stable release dated 22 December 1999 or later from www.webmacro.org).
- xerces.jar (Apache XML parser from xml.apache.org).

Installing Under Java 2 Enterprise Edition

The steps below assume you have an environmental variable %J2EE_PATH% pointing to where you have J2EE installed.

1 Copy following jar files and/or classes to %J2EE_PATH%\lib\system\ subdirectory.

- webmacro.jar from WebMacro (www.webmacro.org)
  NOTE: Use stable release dated 22 December 1999 or later from WebMacro
- essf.jar and/or it's classes
- jsdk.jar
  (NOTE: ESSF needs jsdk.jar (JSDK 2.0))
- xerces.jar (Apache XML Parser) from Apache (xml.apache.org)

2 Copy all the ".class" files into the %J2EE_PATH%\lib\classes subdirectory.

3 Copy the application's WebMacro template (.wm) files to the "%J2EE_PATH%\lib\classes \endtoend_wmfiles" subdirectory.

4 Copy the "CurrentScreenApplet.class" file into the "%J2EE_PATH%\public_html" subdirectory.

5 Copy all the ".html", ".jsp", ".jpg", and ".gif" files into the "%J2EE_PATH%\public_html" subdirectory.

6 Copy the WebMacros jar file (webmacro.jar) and the WebMacro.properties file into the "%J2EE_PATH%\lib\system" subdirectory. After Copying WebMacro.properties file edit the file and append to "TemplatePath" the path to WebMacro teampate (.wm) files the application i.e., "%J2EE_PATH%\lib\classes\endtoend_wmfiles". Read the comments in the file for appropriate pathseparators, delimiters.

7 Modify the "SYSTEMJARS" entry in the "setenv.bat" file in the %J2EE_PATH%\bin subdirectory.

- Prepend this to the start of SYSTEMJARS:
  %J2EE_PATH%\lib\system\lib\xerces.jar

- Append this to the end of SYSTEMJARS: %J2EE_PATH%\lib\system\webmacro.jar
- Append this to the end of SYSTEMJARS: %J2EE_PATH%\lib\system\essf.jar
- Append this to the end of SYSTEMJARS: %J2EE_PATH%\lib\system\jsdk.jar

8 Start the J2EE server by entering this command from the “%J2EE_PATH%\bin” directory:

  j2ee -verbose <Enter>. (This opens the server up in "verbose" mode - so you can see any trace statements and exceptions that may occur.)

NOTE: NOTE: Each time before you restart the server, look into the "c:\j2sdkee1.2\repository" directory for a subdirectory for your machine. Delete all files from the "web" subdirectory (these are the compiled JSP pages). If you don't "clean out the cache" like this, you may encounter a server crash when you start up the J2EE server.

Installing Under Apache & Tomcat

The steps below assume you have an environmental variable %TOMCAT_HOME% pointing to where you have Tomcat is installed.

1 Copy following jar file and/or classes to the %TOMCAT_HOME%\lib subdirectory.

- webmacro.jar from WebMacro (www.webmacro.org) NOTE: Use stable release dated 22 December 1999 or later from WebMacro
- essf.jar and/or it's classes, jsdk.jar NOTE: ESSF needs jsdk.jar (JSDK 2.0)
- xerces.jar from Apache (xml.apache.org)

2 Create a subdirectory called "endtoend" under the subdirectory %TOMCAT_HOME%\webapps.

3 Create a subdirectory called "Web-inf" underneath the %TOMCAT_HOME%\webapps\endtoend subdirectory you just created.

4 Create a subdirectory called "classes" underneath the %TOMCAT_HOME%\webapps\endtoend\Web-inf subdirectory you just created.

5 Modify the Tomcat server's deployment descriptor file "server.xml". This file is in the following directory: %TOMCAT_HOME%\conf. You should add a new "Context" element into "server.xml" that represents the End-to-End Example. To do this add the following to the file:

```
<!-- End to End Example Entry for Tomcat  -->
<Context path="/endtoend" docBase="webapps/endtoend" debug="0"
    reloadable="false" >
</Context>

<Context path="/servlets" docBase="webapps/endtoend " debug="0"
reloadable="false" >
 </Context>

<Context path="/servlet" docBase="webapps/endtoend " debug="0"
reloadable="false" >
</Context>
```