Chapter 4

Efficient Multicast E-Services over Appcast

In this chapter we show the design and development of multicast applications on the overlay created using the Appcast protocol in chapter 3. While chapter 3 focused on overlay creation, this chapter focuses on multicast application development. We show three kinds of applications: 1. Mass File Push is a static file transfer application that pushes files at regular intervals to the members of the overlay. 2. Database Replication is an XML file transfer application, which carries events and actions in the file and the members after receiving the file take actions at their end on their databases. 3. E-Auctions is an auction application that is interactive with the quotes and products information pushed on to member terminals in multicast mode and the individual bids for products by each member is carried back to e-auction server in unicast mode. We propose to use HTTP, SMTP and FTP as transport protocols; SOAP [18-19] as middleware and XML data exchange language in our design and development of these applications. Section 1 introduces the multicast application development and the middleware. Section 2 gives the generic multicast application development architecture over Appcast. Section 3 describes how the three applications listed earlier are implemented. Section 4 concludes the chapter.

4.1 Introduction

Applications are not able to exploit broadcast media property and reduce redundant packets' movement over network as the applications are written in unicast mode due to much complexity of multicast application programming support. In the paper titled 'Broadcast News' the author John Hunt explained how to write multicast applications. Below is the method of writing multicast applications in which a multicast server is set up that keeps sending data at regular intervals and clients are set up to receive the data sent by server. Both server and clients are set up using native TCP/IP socket programming in Java [76].
The MulticastServer
A multicast server can send data to a group of clients. Following steps show how to set up a multicast server.

1. Get an InetAddress for the group.
2. Create a MulticastSocket object on a specified local port.
3. Create a DatagramPacket object containing the data to be sent, the host and port.
4. Send the packet to the multicast socket to broadcast it.
5. Loop back to step 3.

The ClientListener
The client listens for receiving the multicast server data. Following steps show how a multicast client can be set up.

1. Get an InetAddress for the group.
2. Create a MulticastSocket object on a specified local port.
3. Join the multicast group.
4. Create a DatagramPacket containing a byte array into which the client will receive the data broadcast.
5. Call receive() to wait for a packet.
6. Query received datagram packet for the data sent.
7. Loop back to step 5.

The above-explained IP multicast is a very powerful and efficient way of broadcast information quickly and effectively to a large number of listeners (which may be changing dynamically). It does not tie the server to the clients or the clients to the server. Indeed the server could change and the clients need never know. Also, the clients can broadcast data, which cannot be controlled. It has its limitations, including being not available in every environment.

We are aiming at removing the programming complexities. For example, developer need not be aware of the multicast programming. He just develops the application as a normal unicast application. However, at the time of deployment the application can take advantage of the multicast based on the infrastructure available.
4.1.1 SOAP

To develop applications on any network or system, the developers require some kind of API - Application Programming Interface. Traditionally, sockets have been the interface between transport layer (TCP/IP) of underneath network and the application above the network. The Application Layer Multicast protocols discussed in chapter 2 built their own programming interfaces on top of TCP/IP using sockets. As the protocols like HTTP, SMTP and FTP on top of TCP/IP becoming the ubiquitous standard applications across enterprise, we look for a programming interface that uses these protocols i.e., instead of using low-level TCP/IP sockets, the high level protocol API of HTTP, SMTP etc are preferred. SOAP - Simple Object Access Protocol is one such protocol, which started gaining importance. In this section we describe SOAP and its evolution as middleware.

Introduced in 1999, specifying how to use XML and HTTP as an RPC like infrastructure, SOAP provides a standard object invocation protocol built on Internet standards, using HTTP/SMTP/FTP as the transport and XML for data encoding. SOAP [18,19] evolution into middleware classifies it into the Service Oriented Middleware as described below.

a) **POM - Procedure Oriented Middleware**: Enables the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine. Remote Procedure Call packages from Sun RPC[15], DCE 1.1 RPC[16], Microsoft RPC fall into this category.

b) **MOM - Message Oriented Middleware** [17]: Provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action. IBM MQSeries, Microsoft's MSMQ, JMS – are examples of this type.

c) **OOM - Object Oriented Middleware**: Enables the objects that comprise an application to be distributed and shared across heterogeneous networks. CORBA[10] from OMG,
DCOM[12] based on Microsoft's COM[11], Java RMI [14] fall into this category. A good comparison of all these methods from a developer perspective can be from [13].

d) SOM - Service Oriented Middleware [19,20]: Enable the application services interact with each other. SOAP is fast becoming the standard for this application service oriented architecture.

SOAP is much flexible, giving application developers to choose any application protocol like HTTP, FTP, SMTP as the wire protocol. Developers use SOAP along with other web services protocols like WSDL (Web Services Definition Language) and UDDI (Universal Description, Discovery and Integration).

**Web Services Definition Language (WSDL)**

WSDL standard is an XML format for describing the network services and its access information. It defines a binding mechanism used to attach a protocol, data format, an abstract message, or set of endpoints defining the location of services. Usually a service provider creates Web services by generating WSDL from its exposed business applications. A public/private registry is utilized for storing and publishing the WSDL-based information.

**Universal Description, Discovery and Integration (UDDI)**

UDDI defines the standard interfaces and mechanisms for registries intended for publishing and storing descriptions of network services in terms of XML messages. In web services model, UDDI provides the registry for Web services to function as a service broker enabling the service providers to populate the registry with service descriptions and service types and the service requestors to query the registry to find and locate the services.

**4.2 Design and Implementation**

As stated earlier our approach depends on SOAP, HTTP, FTP and SMTP, which are all standard Internet protocols. Our aim is not to create any more protocols on top of these for setting up broadcast services. We define "Producer Service" as the Server Service that pushes the information, actions etc and "Consumer Service" as the Client Service that keeps listening to receive the information, actions etc from "Producer Service". Apart from the producer server and consumer server, we use one more component called
"Registry", that acts as a directory of producers and consumers. Any interested consumer can search this "Registry" and the consumer can subscribe to the service of his interest and set up his own "Consumer Service" to consume the information that is received from the "Producer Service". The "Registry" itself is a service like UDDI service that allows producer services to advertise their service specifications. While this registry is a global service, every producer service also keeps a registry containing the consumer information.

4.2.1 Producer Service Registry Information

Every "Producer Service" gives the following details to the "Registry" to advertise its service. The "Registry" service assigns a unique "Service Id" to identify the service.

**Service ID:** This ID is automatically generated by "Registry" system to identify the service.

**Name:** Every service has to identify itself with user-friendly service name that can easily be searched by the consumers. In our applications the names are given like Getfile, GetAuction, GetTrans etc.

**Description:** A detailed description of the service in normal plain text describing how the service can be utilized, what can be done with the information that the service delivers to consumer, how to set up the corresponding consumer service to receive the information etc.

**Started On:** The Date from which onwards the service is available.

**Schedule:** Schedule of the service to push the information at regular intervals.

**Broadcast Address:** It is the network Broadcast address of the producer service.

**URI:** The URI (Uniform Resource Identifier) with which the "Consumer Service" would be identified, i.e. consumers have to setup their service as this URI only if they want to receive information in broadcast mode.

**Request Type:** Type of the protocol using which the service pushes the information, like HTTP, SMTP, and FTP etc.

**Response Type:** Type of the protocol using which the consumer service can send response, like HTTP, SMTP, and FTP etc.
WSDL: A file, that contains the Service Description Language written as per the WSDL specifications.

Contact Address: Service Administrator's postal address.

E-Mail Address: E-mail addresses to which queries and responses can be sent.

Other Info: Any other information that the service likes to inform its consumers.

4.2.2 Consumer Service Registry Information
Consumers search for the "Producer Service" in the global "Registry" service, look at its details, set up their corresponding consumer service and subscribe for it with the registry maintained at producer. Following information for each consumer is kept at producer.

Service UniqueID: The Producer Service's ID, the consumer has selected.

Consumer UniqueID: Generated automatically by producer system

Name: Name of the consumer / organization.

Broadcast/Unicast?: Whether subscribing for Broadcast service or Unicast service?

Unicast Address: Unicast address to be used by "Producer Service" if the consumer has subscribed for Unicast mode or if the information is lost while sending through broadcast.

Contact Address: Consumer Service Administrator's contact postal address.

E-Mail Address: E-mail addresses for contact.

Other Info: Any other information that the "Producer Service" likes to know about its consumers.

4.2.3 The Basic Steps
Below we show the steps involved for producer server and consumer server to communicate within an Appcast environment.

1. The producer server publishes/adVERTISEs that it has set up a service - S, that pushes information in XML form at regular intervals. It also advertises the Service Description Language of the corresponding consumer service using which the consumers can set up their service.
2. It also advertises that it is on a broadcast medium and those who want to consume this service on broadcast should set up service with the name URI - Http://broadcastadress/servicename.

3. The clients (consumers) who want to consume that service now can set up a service - C such that it can receive the content sent by the server and in response can send a mail or call an acknowledgement service hosted at producer.

4. The clients now can register with the service - S, giving their uni-cast IP address, so that the server can resend the content in point-to-point model in case the client could not send the acknowledgment.

5. If C has many clients further down on its broadcast network, it can simply make itself as proxy service - P to S as shown below.

![Proxy Chaining](image)

**Figure 4.1 Proxy Chaining**

6. P advertises the same and clients down P can set up their own services. Like this the chain can go further down.

7. Those who are not on broadcast medium at any level of the chain can register with the service as point-to-point communications.
The figure 4.2 shows the flow of events. The events are as described below.

1. Producer advertises the service.
2. Consumers (1,2,3) set up their consuming services and register with the "Producer Service".
3. Producer broadcasts the content to 1,2,3.
4. Consumers 1,2,3 send acknowledgements as a mail using standard SMTP protocol or call "acknowledge service" hosted at producer.
5. Consumer 4 registers itself as unicast, and receives data using point-to-point communication.
6. Consumer 4 sends acknowledgement as mail using SMTP or calls "acknowledge service" hosted at producer.

4.3 Implementation

In all the examples of our implementation, we used the producer as the original information provider. Consumers can subscribe to producer for their own consumption or for distributing the information further down. We call these kinds of consumers as distributors. The distributors are almost equal to producers except that they themselves
cannot produce any content. Once distributors receive content from the original producer, then only can they distribute further down. The following figure 4.3 depicts this scenario.

4.3.1 Mass Information Push
The consumer can set up the “GetFile” consumer service from the information advertised by the "Producer Server" such as the corresponding GetFile WSDL, the broadcast address, Service's e-mail address and the URI. The figure 4.4 depicts the design and flow of "GetFile" service. The consumer contacts the registry of the producer, and registers himself. Once it registers, a virtual directory will be created at the consumer and a dynamic web service will be created at the consumer as per the producer's specifications like the URI, input parameters, output parameters etc and configures the web service to receive the file at scheduled intervals. It allows the consumer to modify his registration or unregistered also. The figure 4.5 shows the sequence of events for registration.
The consumer can register either as distributor or just as a consumer. In case he registers as a distributor, then the web service created at the consumer will be capable of redistributing the file to the consumers down it.
The producer gets the data file that has to be pushed, by calling the local “GetData” procedure and gets all the URI of the consumers to whom it should push the data, by calling “GetIP” local procedure. Then it calls the remote “FileTrans” procedure, a web service hosted at consumer, which in turn calls its local “Getfile” procedure that takes action (like storing) on the received file. The “Getfile” procedure checks whether its distributor and if so, it repeats all the steps as the producer. The figure 4.6 shows the sequence diagram show the file push events.
Setting up consumer service

To set up a consumer service, the consumer has to download the corresponding WSDL file hosted at the producer. From the WSDL file, the consumer computer can display the following wizard. Once the required fields are specified, a web service at the consumer side is created automatically.

Figure 4.7 Getfile Wizard
GetFile Consumer Service:

```csharp
<%@ WebService Language="c#" Class="GetFile" %>
using System;
using System.IO;
using System.Web;
using System.Web.Mail;
using System.Web.Services;

[WebService]
public class GetFileWebService {
    [WebMethod]
    public void getfile(string fileType, string content) {
        /** Type of the file and content are parameters to this web method **/

        /** Storing the File **/
        string FILE_NAME = "C:\WS\upload\getFile\";fileType;" + fileType;
        StreamWriter sw = File.CreateText(FILE_NAME);
        sw.WriteLine (content);
        sw.Close();

        /** Sending E-Mail **/
        MailMessage msg = new MailMessage();
        SmtpMail SmtpServer = "mailidbt";
        msg.From = "consumer1@idrbt.ac.in";
        msg.To = getFileservice@idrbt.ac.in;
        msg.Subject = "Acknowledgement";
        msg.Body = content;
        msg.BodyFormat = MailFormat.Html;
        SmtpMail.Send(msg);
        Server.ClearError();
    }
}
```

The received content is stored as file.

Prepare the acknowledgement and send the mail to the server.

GetFile WSDL file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns:soap="http://schemas.xmlsoap.org/soap/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://tempuri.org/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
  xmlns:tns="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:mime="http://schemas.xmlsoap.org/soap/mime/"
  targetNamespace="http://tempuri.org/"
  xmlns:schema="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <s:element elementFormDefault="qualified" targetNamespace="http://tempuri.org/"
      name="getfile">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="emailID" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="fileType" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="content" type="s:string"/>
          </s:sequence>
        </s:complexType>
    </s:element>
    <s:element name="getFileResponse">
      <s:complexType>
      </s:element>
    </s:element>  
  </types>
  <message name="getfile">
    <parts>
      <s:part name="emailID" element="emailID" use="optional"/>
      <s:part name="fileType" element="fileType" use="optional"/>
      <s:part name="content" element="content" use="optional"/>
    </parts>
    <soap:body use="literal"/>
  </message>
  <binding name="getfileSoap" type="http://tempuri.org/GetFileService">
    <soap:address location="http://tempuri.org/GetFileService"/>
  </binding>
  <service name="GetFileService">
    <port name="GetFileServicePort" binding="http://tempuri.org/GetFileService">
      <soap:address location="http://tempuri.org/GetFileService"/>
    </port>
  </service>
</definitions>
```
6.4.2 Database Replication

Just like the previous application, in this application, the consumer can set up a "GetTrans" consumer service with the information advertised by the "Producer Server". In our example, we just used a simple XML file as database transaction file that includes database actions like 'add', 'update', and 'delete' along with data. Once the XML file with all data and actions is received, the consumer acts upon the data using the actions specified against its own database. Lot of work on standardizing the XML with database events embedded is going on [19].

Figure 4.8: Database Replication Wizard

![Database Replication Wizard](image)
GetTrans Consumer Service:
<%@ WebService Language="c#" Class="Transactions" %>
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web;
using System.Web.Mail;
using System.IO;
using System.Web.Services;
public struct AddTrans
{
    public String AccountNumber;
    public String TransactionID;
    public String TransactionType;
    public String Amount;
}
public struct AddCust
{
    public String AccountNumber;
    public String CustomerName;
    public String Address;
    public String Balance;
}

{
    public String strToParse;
    public AddTrans at;
    public AddCust ac;
    public String TableName;

    [WebMethod]
    public void GetTrans(String strSource)
    {
        /** Storing the file **/
        String FILE_NAME = "C:\WS\upload\trans.xml";
        StreamWriter sw = File.CreateText(FILE_NAME);
        sw.WriteLine (strSource);
        sw.Close();
        try{
            String strAdd;
            String strDelete;
            String transID;
            String strModify;
            int iPos; /* Initial position */
            int ePos; /* End Position */
            strToParse=strSource;
            TableName=getValue("<Table>");

            /** Add operation **/
            iPos=strSource.IndexOf("<Add=").
            strAdd=strSource.Substring(iPos,strToParse.Length-iPos);
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            at=new AddTrans();
            at.AccountNumber=string.Replace(strAdd,strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            at.TransactionID=string.Replace(strAdd,strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            at.TransactionType=string.Replace(strAdd,strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            at.Amount=string.Replace(strAdd,strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust.Add(at);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strDelete=GetDelete();
            transID=strDelete.Substring(strDelete.IndexOf("=")+1,strDelete.Length-strDelete.IndexOf("="));
            delCust=GetCust(strDelete);
            delCust.Remove((AddCust)at);
            delCust.AddDel(TableName);
            TableName=getValue("<Table>");
            strModify=GetModify();
            transID=strModify.Substring(strModify.IndexOf("=")+1,strModify.Length-strModify.IndexOf("="));
            modifyCust=GetCust(strModify);
            modifyCust.Modify(transID,delCust,TableName);
            TableName=getValue("<Table>");
            strAdd=GetAdd();
            transID=strAdd.Substring(strAdd.IndexOf("=")+1,strAdd.Length-strAdd.IndexOf("="));
            addCust=GetCust(strAdd);
            addCust.AddDel(TableName);
            TableName=getValue("<Table>");
        }
    }
};
ePos = strSource.IndexOf("</Add>");
strAdd = strSource;
while(iPos != -1)
{
    strToParse = strAdd.Substring(iPos, ePos - iPos);
    if(TableName == "Trans Table")
    {
        AddTrans ad = new AddTrans();
        at.AccountNumber = getValue("<AccountNumber>");
        at.TransactionID = getValue("<TransactionID>");
        at.TransactionType = getValue("<TransactionType>");
        at.Amount = getValue("<Amount>");
        AddDetails();
    }
    if(TableName == "Cust Table")
    {
        ac = new AddCust();
        ac.AccountNumber = getValue("<AccountNumber>");
        ac.CustomerName = getValue("<CustomerName>");
        ac.Address = getValue("<Address>");
        ac.Balance = getValue("<Balance>");
        AddDetails();
    }
    strAdd = strAdd.Substring(ePos + 5, strAdd.Length - ePos - 5);
    iPos = strAdd.IndexOf("</Add>");
    ePos = strAdd.IndexOf("</Add>");
}

/** Delete Operation **/
ifPos = strSource.IndexOf("<Delete>");
ePos = strSource.IndexOf("</Delete>");
strDelete = strSource;
String keyToDelete;
String str = "";
while(iPos != -1)
{
    strToParse = strDelete.Substring(iPos, ePos - iPos);
    if(TableName == "Trans Table")
    {
        keyToDelete = getValue("<TransactionID>");
    }
    else
    {
        keyToDelete = getValue("<AccountNumber>");
        Delete(keyToDelete);
        strDelete = strDelete.Substring(ePos + 8, strDelete.Length - ePos - 8);
        iPos = strDelete.IndexOf("</Delete>");
        ePos = strDelete.IndexOf("</Delete>");
    }
}

/** Sending Mail **/
//Now send the mail.

GetTrans WSDL file:
<?xml version="1.0" encoding="utf-8"?>
    xmlns:xsi=http://www.w3.org/2001/XMLSchema" xmlns:s0="http://tempuri.org" xmlns:soapenc =
6.4.3 Open Auctions

Participating in auctions is an interactive activity. The consumer has to establish 2 services, one through which he sets up his information receiving "Consumer Service", and the other through which he views the information and sends the response.

Figure 4.9: Auction Service Wizard
GetAuction Consumer Service:

```csharp
<%@ WebService Language="c#" Class="Auction" %>
using System;
using System.IO;
using System.Web;
using System.Web.Mail;
using System.Web.Services;

[WebService]
public class Auction:WebService {

[WebMethod]
public String GetAuction(string content) {

    /** Storing the File **/
    String FILE_NAME = "C:\WS\upload\Auction.xml";
    StreamWriter sw = File.CreateText(FILE_NAME);
    sw.WriteLine (content);
    sw.Close();

    Display content;
    try{
        /** Sending E-Mail **/
        MailMessage msg=new MailMessage();
        Smtph.SmtpServer="mailidrbtl";
        msg.From = " acausevice@idrbtl.ac.in";
        msg.Subject = "Auction Ack:"
        msg.To=consumer1 @idrbtl.ac.in;
        msg.Body=content;
        msg.BodyFormat = MailFormat.Text;
        Smtph.Send(msg);
        Server.ClearError();

        Store the auction.xml for audit purpose
        Show the bids
        Send the mail as acknowledgement
    }
}
```
6.5 Advantages and Disadvantages of the model

Since our approach is at application level, we compare our method with other application level broadcast approaches. Figure 4.11 clarifies the simplicity of our approach. While protocols like Overcast [5] and Scattercast [3-4] had built some more new protocols - Gossamer, Up/Down protocols etc on top of TCP/IP; Appcast relies on its Appcast overlay on which SOAP is used as wire protocol.

- The receivers need not write applications especially for broadcast as in traditional IP Multicast.
- No special infrastructure deployment is required as in Overcast [5] or Scattercast [3-4].
- Customization wizards can be set up by commercial vendors such that consumers can very easily set up consumer service without need to write a code as all applications follow the standard protocols like SOAP, HTTP,
SMTP etc. Using Proxy chaining is exclusively in the hands of network administrators.

✓ Controlling is very easy.

✓ No need to look into router level network outages etc, as all data loss and resending is controlled by application level utilities like HTTP and SMTP.

✓ Appcast environment can be build incrementally.

✗ The model is taking advantage of "TCP Spoofing" which may not be acceptable to all. But in a closed environment and between a parent and just it's next level client node, this can be enabled without any problem.

✗ The model assumes that the communication between sender and receiver is reliable. We are taking advantage of broadcast media between sender and receiver of same network. On LANs and Satellite based WANs communication media is reliable. If the sender and receiver are on different networks, with so many networks in between, the communication is just like point-to-point. And for this, the sender need not write application separately.

✗ Though the source sends the information, it receives acknowledgements from all receivers unlike in other multicast protocols. Since we use application level features, unlike others, who control information-flow at packet level and hence no application level acknowledgement.
4.4 Conclusion
Multicast push-services increase network efficiency. In our model, though we could make the same application push the data in both unicast and multicast modes, we could not actually install E-Service on consumer machine dynamically. We did the whole experiments in our LAN environment and we wrote the consumer services separately for each application as installable executables. We discuss below some issues in setting up consumer services.

How to Set Up Consumer Service?
Normally setting up an E-service in a web-service paradigm is to receive a request and send a response. Even the web sites behave this way. Majority of the times, the request is just like a query, which is few bytes from the initiator and response is like sending a list of results (may be huge data), which is from the service. The initiator need not be connected always to Internet/network. However, the service must be always up listening for the requests from consumers. In multicast, it is totally in contrast. The initiator (producer) pumps large amounts of data. The consumers must always be up and connected to network, listening for the service (producer) to push the data and the consumers just send an acknowledgement as response.

Dynamic E-Service
How the consumers can set up the listening service? What do they do with the information they receive? Unless there is some natural language, it is difficult to specify these things. In general, the WSDL specifies only the input, output parameters, the remote service names, the ports at which these services listen etc. But it cannot specify how to act upon those parameters. But the service provider is aware of how to process the input parameters and arrive at the result to send the response. So, here the service provider informs the consumers, how to call its service. In multicast, it is different in that the consumer should already be having the consumer service to which the producer can push the information. Unless, all consumers set up their receiving services in same way, the producer cannot push the information. So, he advertises the service definition in terms of input and output parameters, just like in WSDL. However, the difference is that, in normal course the consumers call the producer service and in multicast, the producer calls
the consumer's service. The consumer has to set up its service dynamically by looking at
the WSDL.

E-Service Languages

As discussed earlier, WSDL is just like an Interface Definition Language. It cannot
describe the E-Service in totality. In a multicast environment, one needs a language that
is common on all platforms, such that once described, it can be implemented on any
platform. Work in this direction of describing a web service in XML and executing on
any platform has been described in [52]. It details on program statements, expressions,
variables etc. In our case of multicasting, this work may help if every consumer would
like to use the received information in similar fashion. This can be applied in case of
distributors, where in producer can program how the information be distributed. If so, the
producer can place the service in XL language [52], which the distributor can download.
Digital signatures can be used to authenticate the code. However, if the consumers would
like to use the received information in different fashions, then this will not help. The
producer can dictate what the consumer can receive, but not how to consume the same.
So, we feel that, the programming languages for the web services paradigm must vary
from simple to complex. For example, it should allow simple statements like below to
receive the office order, store it and display or mail it.

```
{ 
Receive Office Order From http://www.organization.com/orders/ 
Store Office Order In d:\folder
Display Office Order 
}
```

```
{ 
Receive Office Order From http://www.organization.com/orders/ 
Mail Office Order admin@myoffice.com
}
```