CHAPTER 3

ENHANCED DISTRIBUTED SERVICES FOR VIDEO ON DEMAND

3.1 INTRODUCTION

The demand for secure, integrated and distributed video service is increasing significantly from the last decade. The development of distributed applications has been evolved with many different platforms, approaches, specifications and relevant APIs from the open standards like RMI, OMG, XML-RPC, OASIS, Apache and many open groups/forums. As a result, there are many standards and specifications that are formulated by researchers and technical committees to achieve the requirements of distributed application/middleware development.

Many factors like ease of use, maintainability and performance are to be considered while choosing one of these standards or specifications for distributed application development (Jagannatham et al. 2007). There is always a high dilemma to decide a suitable environment while developing multimedia applications over Web and providing multimedia contents using different platforms. This chapter is mainly dealt with design and development of scalable and enhanced distributed models for Video on Demand. A SOAP communication model for effective VoD using multicasting is proposed and further investigated on the deployment of VoD services on cloud environment. The Web Services platform is evolved from the past distributed component technologies like Remote Procedure Calls (RPC), messaging and
Object Remote Procedure Calls (ORPC) such as DCOM, CORBA and Java RMI. The conventional client-server computing provides an environment for distributed application development that partitions the tasks or workloads between service providers, called servers and service requesters, called clients. An important aspect of implementing a client-server model for Video on Demand application is to distribute the video files between the client and the server and to make this implementation transparent to the video requester who is located geographically apart.

The main advantage of the client-server model is its better reuse in which the video files are placed solely in the server and can be fetched by the clients. In the client-server model for Video on Demand, each instance of client will send a request for a video file to one or more connected server. In turn, the servers can accept this request, process it and return the requested video file to the client, thereby providing easy access and robustness for Video on Demand model. The video requesters will suffer when the server gets congested or overloaded, and the whole Video on Demand system fails when the server fails.

The Remote Method Invocation is a distributed client-server model that allows an object running in one virtual machine (VM) to invoke methods on an object running in another Java VM. It provides the mechanism for the server and client to communicate and to pass information back and forth. The RMI server creates remote service objects, makes a reference accessible for the clients and waits for them to invoke the methods on those remote objects. An RMI client gets a remote reference to one or more remote objects in the server and then invokes methods on them. An RMI based distributed application uses the registry to obtain a reference to a remote object. The RMI server calls the registry to associate a name with a remote object. The client looks up the remote object by its name in the server’s registry and then
invokes a method on it. The Java RMI based model for remotely accessing the media files has been reviewed in the previous chapter. RMI based implementation of VoD eliminates the overheads associated with conventional approaches. The major problem is that, the traditional VoD models have self-contained monolithic programs, which are having limited access to one another’s procedure. This approach is not suited for real-time, as they are usually cumbersome to build and expensive to maintain because simple changes in the procedure or data may cause the entire application to be rewritten, recompiled and retested.

The implementation procedure of RMI and Web services are more similar as they both use stubs on the client side and skeletons on the server side to mask the complexity of remote communication from the developer (Matjaz et al 2005). Inspite of this similarity, there are several significant differences exist between Java RMI and Web services. The main difference between the various types of service-oriented systems is in the communication protocol. In RMI, the messages are serialized Java objects and transmitted over Java Remote Method Protocol (JRMP).

The CORBA and COM are using very efficient binary data representation for message transfer. In case of Web services, the communication is using XML data over the Web protocols such as Hyper Text Transport Protocol (HTTP), Simple Object Access Protocol (SOAP), etc. Thus, to pass through a proxy or firewall, Web services traverse seamlessly as they rely on HTTP port 80/ HTTPS port 443 which are normally not blocked at firewalls, whereas RMI requires a separate port to be allocated and opened for the Video on Demand model, which is quite a complicated task.

In a comparative study carried by William et al (2006), it is found that RMI application requires multiple round trips to access even a single document. The RTT for both RMI and Web Service implementations, scale
linearly with the number of files being retrieved. However, the slope of the RMI implementation is much greater. The key advantage of Web services is that the deployed services are intended to solve the key issues like interoperability, complexity and firewall traversal.

3.2 **XMLISED REPRESENTATION OF VIDEO DATA**

Internet provides a heterogeneous environment for Web applications development. Number of applications, which are hosted in the Web, is countless and it is a growing phenomenon. Many vendors on the Web provide the various categories of applications that include social media, consumer applications, industrial real time applications, learning and entertainment applications etc. These applications have been developed in different platforms using different programming and scripting paradigms. The data needed for those applications are in different formats and are scattered geographically apart. Interoperability becomes a major issue in most of the Web enabled applications.

Data sharing between Web enabled multimedia applications should be reliable, consistent, and platform-independent especially with respect to audio and video files. The major problem with most of the relational databases is their incompatible formats while accessing by various applications of heterogeneous nature. XML is both hardware and software independent paradigm and has flexible integration capability with any type of platform or programming paradigm. Thus, to enhance the interoperability between multimedia applications in a distributed environment, it becomes very essential to convert or transform the multimedia files stored in the relational databases into an XML document for flexible data sharing. As XML is text based, data sharing becomes very easy and faster when compared to the multimedia files, thereby increasing the Internet computing performance. By providing proper authentication details, the blob data stored
in the relational databases can be converted to XML documents that can be used independently for multiple purposes.

An efficient, reliable and secure method is required for transforming the data and for transferring the same in a distributed environment, especially when the size of data is large as in case of binary large objects i.e., blob data. The major problem with blob data is that, the universal database support for this data type is very minimal. Unlike RDBMS, which is in a machine readable format, XML document is human readable and is self-explanatory. It is also very simple to add or remove data to a node in an XML document, but still keeping the file size in manageable state. An XML based data conversion model is proposed to convert the blob data into XML document and it is implemented as a Web service “XMLisedBlobDataGeneration”, using JAX-WS environment. The XMLised blob data representation is not only better to share the contents but it is also easier for querying the required data.

3.2.1 Data Conversion Model

A Video on Demand system needs an efficient and faster method for blob data sharing in a distributed environment. In a 3-tier model, usually the video data are stored on to the relational database as a blob entity. Exchanging binary information in a Web service environment will lead to interoperability issues. To make use of the complete capability of the Video on Demand services over Web, it would have been better that the blob data should be in XML form, which ensures the interoperability of VoD services in the heterogeneous environment. The conversion of blob data into text based XML document is preferred in SOAP messages, both in request and response payloads rather than including the blob data as attachments. If the video data are stored in disk files, the same conversion procedure is applicable as the video data can be easily converted into sequence of bytes.
Using JDBC API, the video data can be inserted and retrieved as BLOB entity, but the procedure differs from one relational database to another. As the first step of the proposed VoD system, a data conversion model is implemented as a Web Service using the Java API for XML (JAX-WS). The JAX-WS based XMLised blob data generation service is created as a message-oriented Web service, which is built upon the AXIS2 engine and supports the SOAP 1.2 specification. Figure 3.1 shows the complete deployment procedure for XMLised blob data generation as a Web service.

![Diagram](image)

**Figure 3.1 Deployment of XMLised Blob Data Generation Web Service**

The inputs required for the data conversion model are the service endpoint implementation, an XML document that designates the URL for the service endpoint implementation and a service deployment descriptor file
using XML to provide a URL to the appropriate Web Service Descriptor file through which the client can access the XMLised blob data generation service. In case of JAX-WS based applications, the Web service implementation class itself implicitly defines a Service Endpoint Interface.

The JAX-WS runtime provides an Annotation Processing Tool (APT), which has a list of annotations to be used with the endpoint implementation to generate the endpoint interface and other components and also provides wsimport command to generate portable client side artifacts in order to enable the client to communicate with the Web service. The input, XMLBlobGeneration.java represents the service implementation. This endpoint implementation class is annotated with javax.jws.WebService and hence the Service Endpoint Interface is implicitly defined. The methods that are to be exposed as Web services to the clients must be annotated with javax.jws.WebMethod. This implementation class contains the required method XMLGeneration(), which converts the video data as sequence of bytes if it is being stored in a disk file or extracts the BLOB data if it is stored in the relational database and converts them into XML documents using Base64Encoding scheme.

The portable artifacts, which are generated using wsimport command are actually Java Bean classes and are named as XMLGeneration and XMLGenerationResponse respectively. These files are responsible for marshalling / unmarshalling of method invocations and responses. Marshaling and unmarshaling are the transformation from XML to Java objects and vice versa. The configuration file sun-jaxws.xml defines the service endpoint address, using which the video service deployment descriptor is created.

The service implementation class namely XMLBlobGeneration along with the configuration files and the generated portable artifacts are
bundled into a standard WAR file called as XMLBlobGeneration.war. The WAR file is then deployed on to the servlet container. After successful deployment, the service can be accessed from the URL http://localhost:8000/XMLBlobGenerationWSAppln/XMLBlobGeneration, which is the endpoint. The video requesters can use this endpoint address for accessing the XMLised blob data generation services. The video service descriptor file, which is generated using WSDL is used to publish and discover the XMLBlobGeneration service.

Thus, when the client gives a request for a particular video file to the remote XMLGeneration() method using the XMLGeneration bean, the video data stored as blob has been converted into an XML document using the XMLBlobGeneration implementation class and the response is accessed through the XMLGenerationResponse bean.

3.2.2 Implementation of XMLised Blob Data Generation Service

The proposed XMLised blob data generation model dynamically fetches the requested video data from the database and converts it into XML, which lowers the engineering efforts that are necessary to integrate the video data in the Web services environment. The various stages involved in the implementation of proposed JAX-WS model for the XMLised blob data generation are service implementation, service configuration, service deployment and invocation.

3.2.2.1 XML Schema for Blob Data Representation

An attempt has been made to convert the video data that is stored in the relational database as blob entity into an XML document, instead of sending the whole video as such to the video requester. The XML document has to be validated for its correctness and should be error free. The XML
schema or the Data Type Declaration (DTD) is used to format an XML document. The schema defines the exact structure in a descriptive form to store the entire video blob and its metadata. The DTD for the XML document, which stores the video data, is shown below:

```xml
<!DOCTYPE videodbxml [ 
<!ELEMENT videodbxml(videodetail+) >
<!ELEMENT videodetail(videoname?, video?)>
<!ELEMENT vidoename (#PCDATA)>
<!ELEMENT video (#PCDATA)>
]> 
```

The root element of the XML document is “videodbxml” that contains child element “videodetail”. The element videodetail is mandatory and appears either once or can be repeated indefinitely. It contains two child elements namely “videoname” and “video”, which may appear once or never. The data inside the elements “videoname” and “video” are the name of the video file and the contents of the video in XML form respectively.

### 3.2.2.2 Service Implementation

The Web service implementation class (XMLBlobGeneration) defines the methods that a client can invoke for accessing the service and encapsulates all aspects of the network protocol used for communication between the clients and the service provider. An explicit Service Endpoint Interface is not required when building a JAX-WS based Web service.

The implementation class, XMLBlobGeneration, is annotated as a Web service endpoint using the @WebService annotation and it defines a method named XMLGeneration(), annotated with the @WebMethod annotation, which exposes the method to Web service clients. An
XMLGeneration() method returns the XMLised video data to the requester by accepting a video filename as its parameter. At the time of deployment, the servlet container maps the implementation class to WSDL using the annotations specified. The following code shows the implementation of the XMLBlobGeneration Web service.

```java
@WebService()
public class XMLBlobGeneration {
    ... ... ...
    @WebMethod(operationName = "XMLGeneration")
    public String XMLGeneration(@WebParam(name = "xmlFileName") String xmlFileName) {
        ... ... ...
        ps.println("<xml version="1.0" encoding="UTF-8">");
        ... ... while(!data.equals("end"))
        {
            ps.println(data);
            data = br.readLine();
            success = success + data;
        }
        ... ... ...
        return success;
    }
}
```

The XMLGeneration() method uses the middleware component named as “webservlet”, which is deployed in the server to communicate with
the database and to fetch the required video blob data. The following code segment shows the fetching of required blob data from the database and converts the blob data into string using Base64Encoding scheme.

```java
public class WebServlet extends HttpServlet{
    Blob aBlob = rs.getBlob("Video_Data");
    byte[] allBytesInBlob = aBlob.getBytes(1, (int) aBlob.length());
    byte[] byteblob64 = new byte[]{0x12,0x23};
    byteblob64 = allBytesInBlob;
    String s = new sun.misc.BASE64Encoder().encode(byteblob64);
}
```

The above service implementation is deployed in the server for the clients to access it from a remote location.

### 3.2.2.3 Endpoint Configuration and Deployment Descriptor

The standard web.xml file, shown below is configured to deploy the implementation class into the servlet container and to describe the URL, which is listening, by XMLised blob data generation service and the classes that will handle this listening process.

```xml
<listener>
    <listener-class>
        com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
</listener>
<servlet>
    <servlet-name>XMLBlobGeneration</servlet-name>
    <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>XMLBlobGeneration</servlet-name>
    <url-pattern>/XMLBlobGeneration</url-pattern>
</servlet-mapping>
```
Additional configuration file named as “sun-jaxws.xml” is required to define the service implementation class “XMLBlobGeneration”, which will be used by the listener to execute the process by accessing the endpoint address. The endpoint element contains all information about the implementation class “XMLBlobGeneration” and the url-pattern to access the same. The endpoints are configured as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<endpoints version="2.0"
xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime">
<endpoint implementation="org.me.BlobGeneration.XMLBlobGeneration"
name="XMLBlobGeneration"
url-pattern="/XMLBlobGeneration"/>
</endpoints>
```

The clients are aware of the service endpoint through which the XML representation of the video file can be accessed. In the proposed JAX-WS model for XML representation of video file, the WSDL is used as the metadata language for describing the blob data generation services. It describes how the service endpoint and the clients communicate with each other. WSDL is capable of describing services that are implemented using any language and deployed in any platform.

WSDL represents information about the interface and semantics of how to invoke the service. It contains the information about the port, binding and address information for invoking the services from the service provider. The video data is attached with SOAP message and can be communicated to the various video requesters through Service Descriptors, which is created using the XMLBlobGenerationService and is represented in the form of XML. The WSDL file of the deployed service can be viewed from the URL
http://localhost:8000/XMLBlobGenerationWSAppln/XMLBlobGeneration?wsdl. This URL shows the definition for the XMLGeneration message.

```
<definitions targetNamespace="http://BlobGeneration.me.org/
    name="XMLBlobGenerationService">
  <message name="XMLGeneration">
    <part name="parameters" element="tns:XMLGeneration" />
  </message>
</definitions>
```

The return type and the datatype of the parameters required for method invocation for the XMLBlobGeneration service is given in a separate XSD document. The XML Schema binds the XML name to unique Java identifier in a Java package. The complex types and some of the simple types defined for the XMLBlobGeneration service are given below. The URL, http://localhost:8000/XMLBlobGenerationWSAppln/XMLBlobGeneration?xsd=1", shows the XMLGeneration() method and it’s parameters.

```
<xs:element name="XMLGeneration" type="tns:XMLGeneration" />
<xs:element name="XMLGenerationResponse"
    type="tns:XMLGenerationResponse"/>
<xs:complexType name="XMLGeneration">
  <xs:sequence>
    <xs:element name="xmlFileName" type="xs:string"
      minOccurs="1" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="XMLGenerationResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:string" minOccurs="1" />
  </xs:sequence>
</xs:complexType>
```
The XML request for the XMLGeneration() takes xmlFileName as parameter whose data type is a string. Similarly, the data type of the returned response is also a string.

3.2.2.4 Invoking the Service

The proposed XMLised Blob Data Generation Service is successfully deployed that acquires the required video contents dynamically from a database server and creates necessary XML documents. The XMLBlobGeneration service endpoint and the port in which the service is available are known to all the clients, regardless of the language or platform used. When the video requester invokes the remote XMLGeneration() method on the port, the client performs the following steps:

- The client uses the generated XMLBlobGenerationService class, which identifies the URI of the XMLBlobGenerationService that is extracted from the WSDL
- The client retrieves a proxy to the XMLBlobGenerationService service.
- The proxy which is also known as a port, is obtained by invoking the method getXMLBlobGenerationPort() on the service
- The port implements the XMLBlobGenerationService Endpoint Interface, which is defined by the service
- Finally invokes the port’s XMLGeneration() method by passing the file name of the requested video as parameter
The following code delineates how the remote client invokes the XMLGeneration() method that is deployed in the remote server.

```java
org.me.blobgeneration.XMLBlobGenerationService service = new 
org.me.blobgeneration.XMLBlobGenerationService();
org.me.blobgeneration.XMLBlobGeneration port = 
service.getXMLBlobGenerationPort();
java.lang.String xmlFileName = filename;
java.lang.String result = port.xmlGeneration(xmlFileName);
```

The proposed XMLised Blob Data Generation service is tested by fetching video files of various formats. The client sends the file name of the required video as parameter to the remote XMLGeneration() method, which in turn fetches the video from the database and converts the video into XML format. The XML document generated for a particular video file named as “WildLife.wmv” is shown below:

```xml
<?xml version="1.0" encoding="UTF-8"?><videodbxml>
<videodetail>
<videoname>WildLife</videoname>
<video>AAAAHGZ0eXBtcDQyAAAAAGlzvb21hdMxbXA0MgA
AOCttb292AAAAAbG12aGQAAAAAxaJN+sWiTfoAAAJY
AABi5wABAAAAABAAAAAAAAAAAAAAAAAAQAAAA
AAAAAABAAAAAAAAAEBAAAAAAAAAAAAAAA
EFWtvZHMAAAAAEAcATKRXAAAfgXRy ...... ....
...... .... ....ZfkAwZwAMPgA==
</video>
</videodetail>
</videodbxml>
```
The XML document received by the clients as response can be converted to any format like html, pdf or word document as required by the user using the XML Stylesheet Language (XSL) and can also be converted back to video format to watch the video.

3.3 CONCEPTUAL WEB SERVICE ARCHITECTURAL MODEL FOR VoD SERVICES

Current Web service standards enable publishing of video service descriptions and finding of video services based on method signatures. The video providers upload the videos to the server, which can be accessed through distributed video retrieval services from any geographical location, thus made available to requesters. A conceptual Web service architectural model, which will link the various components of the VoD system, is proposed as suggested by the World Wide Web Consortium. This model provides clear understanding of the lifecycle behavior of video service interactions and enumerates the relationship between the key modules of the Video on Demand system.

The proposed Web service architectural model for the VoD system consists of four modules namely, policies, resources, messages and services. The policy module focuses on the security issues such as permission, authentication, and authorization that are created by the video distributors, which are subjected to the client actions on a video file. The policies or the guidelines are created for accessing the video contents as well as the video service. The resource module focuses on the hardware and network resources and especially on the data resources like video / audio contents. The resource component of the proposed Web service model with regard to VoD system maintains all the details regarding the video resources, including the ownership and the policies created for those resources.
The message module explains the way in which the messages or video contents are transported within the VoD system. It also concentrates on the structure of the request and response messages and the relationship between video providers and the video requesters. Any communication to the video service is in the form of SOAP request and response and the messages are processed in accordance with the policies. The service module is on the top of the message module and it extends the same in order to explain the fundamental concepts involved in a service i.e., the purpose of the communication within the VoD system and it examines the different actions performed by the clients in order to utilize the VoD services. The concept map shown in the Figure 3.2, depicts the relationship between the various key components of VoD service model.

![Figure 3.2 Concept Map of the VoD Service](image)

The arrows represent that each component in the VoD system is partially layered on top of the other components.

### 3.3.1 Policy Module

The policy module focuses on various aspects related to the video files such as security and QoS in order to provide the VoD services. The distributor creates the policies regarding the Video on Demand services. The
security of the VoD services is maintained by the audit and permission guards. The audit guard in the policy module is used to continuously track and monitor the clients, video resources and the video services are in consistent with the policies created by the distributors. If there is any violation, the permission guard enforces the corresponding action on behalf of the video distributors. For instance, when the client requests for a video file, the audit guard checks whether the client is a registered user or not and the permission guard depending on the status of the client, specifies the appropriate permission grant whether to access or deny service. The Figure 3.3 shows the Policy Module of the VoD services.

![Policy Module for the VoD system](image)

**Figure 3.3 Policy Module for the VoD system**

The security fundamentally emphasizes on the various constraints based on the client’s action while accessing the video contents or while utilizing any service in the VoD system. Similarly, the Quality of Service is about the constraints on the video service. The QoS is associated with the delivery of messages/video contents by the SOAP mechanism. The delivery policy ensures reliable transmission of video data.

### 3.3.2 Resource Module

The Resource module shows the relationship between the video providers/distributors and the video resources. It maintains the details of all
the video resources that are uploaded to the server and the information about the video provider. It also ensures that whenever the resources are accessed, either by the video providers or video requesters, the corresponding tasks / events are carried out according to the predefined policies assigned by the distributors. The Resource Module of the VoD system is given in Figure 3.4.

![Figure 3.4 Resource module for the VoD system](image)

The video requester identifies the resource location, by requesting the discovery service, which has been catalogued in the Resource Description. The resource description refers the policies that apply to the requested video resource and depending on which, it returns the corresponding URL of the resource to the client for accessing the video. In dynamic discovery, the video requester may interact directly with the discovery service to find the appropriate resource.

Discovery is the act of locating a machine-processable description of Web service-related resources. The Video requester entities will find the
service description during development for static binding or during the execution for dynamic binding. A discovery service is a service that identifies the required resource for the client and also used to publish the service descriptors. The video provider uses this module to publish the video service descriptors. The uploaded resources are owned by the distributors. The client request for uploading or downloading a video resource can be either through HTTP GET or through HTTP POST method, which is a standard action associated with Web resources.

3.3.3 Service Module

The ‘Service’ module focuses on the action that can be performed by the users like either requesting for a video or playing the video. The user can either be a video provider or video requester. An action is typically expressed in terms of executing some fragment of a program. Service is a set of actions that form a coherent group from the point of view of service providers and service requesters. The Service as shown in Figure 3.5 provides the video files (resource) to the client.

![Figure 3.5 Service module for the VoD system](image)
The video provider uploads the video and the video requester uses the video service. Services are mediated by means of the messages exchanged between video requester and video provider. When the user requests for a service, it looks at the Video Service Description that describes all the information regarding the Service Interface. The service interface declares the method signature depending on which the client can request the service. An endpoint is the realization of a video service interface. A VoD Endpoint is a network location at which an implementation of a service interface is made.

The messages are created according to the choreography, which defines the sequence and condition through which the clients can exchange the messages in order to perform a task. For the user to complete the registration task, the sequence of operation would start with the request by the client for “signing up” to the service. The different form elements are sent as response to the user. The user fills up all the form elements and sends the filled in form. The services checks for the validation and if satisfied, the user is successfully registered with the service.

3.3.4 Message Module

The message module focuses on the aspects of architecture that relates to processing of messages. A message represents the data structure passed from the sender to its recipients. The structure of the message is described in the video service descriptors. This module concentrates on the relationship between the message sender and the message receiver and also deals with the way in which the messages are transmitted. The message module does not much emphasize on the semantics of the message. The messages are transported by means of SOAP, which represents the information needed to invoke a service or reflect the results of a service invocation, and contains the information specified in the service interface.
description. For instance, to invoke a video service, the client has to provide the filename of the video, which is attached within the SOAP body of the request message.

Distributed video service communicates via message exchanges. A Message Exchange Pattern (MEP) is a template, devoid of application semantics, that describes a generic pattern for the exchange of messages between clients and servers. The MEP describes the relationship of multiple messages exchanged in conformance with the pattern, as well as the normal and abnormal termination of any message exchange. The message module for VoD system is shown in the Figure 3.6.

![Diagram of Message Module for the VoD System]

Figure 3.6 Message module for the VoD system

The message module defines the structure of a message that mainly contains an envelope, which encapsulates both message header and body. The header holds the information regarding the message and the message body contains the content and the URL of the requested video data. All these
modules form an integral part of the Web Service architecture for Video on Demand system.

### 3.4 WEB SERVICE MODEL FOR VIDEO ON DEMAND

RMI supports only synchronous invocation bound to the Java programming paradigm, making interoperability a major issue. It is proved that an RMI approach serves videos of smaller size better. But RMI suffers from connectivity issues when the service is to be invoked across firewall. A flexible Video on Demand system is required that will provide services in a heterogeneous environment.

A Video on Demand system is designed and developed using Java API for XML based Remote Procedure Call (JAX-RPC), which uses inherent SOAP (Simple Object Access Protocol) communication for request and response. SOAP is an XML based protocol for exchange of information in a decentralized and distributed environment. The built-in API for JAX-RPC allows sharing and exchange of video blobs in the form of XML.

The VideoClient application sets the service endpoint address to the stub, which is a client side artifact that is generated using the configuration of the Web service descriptor. The video service implementation which is the actual remote object is exported to an anonymous port when it is getting instantiated. The invocation of getVideoIntPort() method of VideoService_Impl class associates the anonymous port to the stub. The stub knows everything about a service, while the client does not know anything about the service endpoint. The client invokes the service through the stub.

The architecture of the proposed JAX-RPC model for Video on Demand service is given in Figure 3.7. The server side contains a JAX-RPC server runtime environment and the implementation of the video service
endpoint. The client side contains a JAX-RPC client runtime environment and the VideoClient application.

The client side JAX-RPC Runtime Environment invokes the client Request Handler to process the user’s request for a particular video file. It ensures whether all the data required to fetch the video are available. After confirming the availability of the required information such as video filename, username and password, the request message is sent to the video implementation class on the server as a SOAP request for further processing. The stub in the client side has all the required details about the Video Service Endpoint. Thus the video requester through the VideoClient invokes the method in the Video Service Endpoint. The Web service model for Video on Demand enables Java applications to invoke Web services that execute on non-Java platforms and non-Java applications to invoke Web services that execute on Java platforms.

Figure 3.7 JAX – RPC Architectural Model for Video Service
The client or the video requester needs only the Web service descriptor to access the video service through a Video Service Endpoint Interface. The Web service descriptor has a reference to the Video Service Endpoint Interface, which is a remote interface that declares the remote methods through which the client interacts with the video service in the JAX-RPC environment. The client side runtime contains components to convert the video files into XML and generates SOAP messages and a parser to convert the XML response into a video file.

The developed JAX-RPC model for Video on Demand service uses the static proxy named as stub, which is to be set with the end point address and the port in order to invoke the service. At the server side, the JAX-RPC run time environment invokes the server request handler to handle the received SOAP messages. This checks for the attachment inside the SOAP message and if available, it repackages with the SOAP message context and forwards it to the service endpoint. The response from the Web service endpoint is sent to the client in the same manner. The developed JAX-RPC model for Video on Demand service provides a guaranteed Quality of Service.

3.4.1 Implementation of JAX-RPC Model for Video on Demand

The Video on Demand service is implemented as a remote entity. The VideoImpl class extends the VideoInt interface and provides the definition of the methods declared in the interface. The definition for the getVideo() method explicitly converts the video data in to sequence of bytes stored in a byte array, which has memory restrictions and is time consuming. These issues are addressed in the proposed SOAP communication model, which uses implicit conversion that is faster and does not pose any restrictions
on the memory usage. The video service implementation class is defined as follows:

```java
public class VideoImpl implements VideoInt {
    byte[] buffer;
    public byte[] getVideo(String fname) throws RemoteException {
        File f = new File(fname);
        FileInputStream fis = new FileInputStream(f);
        buffer = new byte[(int) f.length()];
        fis.read(buffer, 0, buffer.length);
        fis.close();
        return buffer;
    }
}
```

If the requested file size is large, the byte array cannot store all the bytes in the local segment and it needs to use the extended memory. Hence the video file of large size more than the extended memory in the server cannot be downloaded to the requested client at one stretch rather divided into segments, which is time consuming. This getVideo() method is being triggered by the clients through the VideoInt Interface to fetch the required video.

### 3.4.2 Configuration of Video Service

The JAX-RPC runtime requires certain meta information regarding the video service in order to generate the client and server side artifacts to handle network connectivity and SOAP communication. JAX-RPC does not
support all the data types as used in the service endpoint interface, for instance, the JAX-RPC understands the data type, byte [] (byte array) as base64Binary. A mapping file has to be generated which maps the standard Java types to JAX-RPC supported types and vice versa. The mapping file consists of all the data required to associate the VideoInt Interface to the definition of Web service descriptor file. A configuration file to describe the video service endpoint interface has to be created in XML and usually named as “config-interface.xml” and it is defined as follows:

```xml
<configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="VideoService"
           targetNamespace="urn:video"
           typeNamespace="urn:video"
           packageName="video">
    <interface name="video.VideoInt"/>
  </service>
</configuration>
```

The definition of the configuration file includes the name of the service, various namespaces, the package name and the details of the video service interface. The Java Web Service Developer Pack provides a tool named as “wscompile”, which is to be used to generate the Web service descriptors and the necessary mapping.xml file. The wscompile tool requires the config-interface.xml file as its input and generates the necessary WSDL and mapping files. These two files are required while deploying the video service in the Web server. During deployment of the VoD service, the server side ties will be generated and bundled with the Web archive file. The WSDL file contains all the necessary details including the location of the Web service and method name. In detail, it consists of the datatype of the messages, the
port number at which the service is operating, the details of the input, output and the fault, the necessary binding information like which protocol will be used to carry the message. This generated “VideoService.wsdl” file acts as the link between the Web service endpoint and the JAX-RPC Runtime.

The Web service descriptor provides information such as package, method, data type, port, part and the endpoint, which are specified in the form of XML. The getVideo() method of the video service returns the byte array and hence the generated WSDL file denotes the response data type as xsd:base64Binary. The following code shows a portion of the generated WSDL file which denotes that the data type of the input is a string and the output is the byte array.

```xml
<message name="VideoInt_getVideo">
  <part name="String_1" type="xsd:string"/>
</message>

<message name="VideoInt_getVideoResponse">
  <part name="result" type="xsd:base64Binary"/>
</message>
```

The video service endpoint interface and its implementation and the generated WSDL and mapping files are packaged into Web archive file and the same is deployed in a Web server. After deployment of the service, the remote object can be accessed through the information contained in the WSDL file. The tie class required to communicate with the clients is generated by the server during the deployment process. The Web service descriptor file can be viewed by requesting the following URL: “http://localhost:8080/myvideo/VideoStream?WSDL”.
3.4.3 Generation of Client Side Artifacts

The WSDL file will provide all information for stub creation and appropriate SOAP request and response serializers, which are termed as client side artifacts. The “wscompile” tool requires “config-wsdl.xml” file as its input, which actually provides the location of the WSDL file to generate these artifacts. The location of the WSDL file is configured as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
<wsdl location="http://localhost:8080/myvideo/VideoStream?WSDL">
   packageName="staticstub"/
</configuration>
```

The configuration file contains the location of the WSDL file deployed in the Web server. The “wscompile” tool generates the artifacts and stores them in the sub-directory named as “staticstub”. One of the important class files in this directory is VideoService_Impl.class which consists of the getVideoIntPort() method that provides the port where the service is listening. The client side stub is acquainted with the service offered by the remote interface. The client will be able to invoke the getVideo() method using this stub. The following code shows how the client fetches the video through the interface.

```java
Stub stub = (Stub)(new VideoService_Impl().getVideoIntPort());
VideoInt client = (VideoInt) stub;
byte [] buffer = client.getVideo("song.mpg");
```
The received sequence of bytes in the form of byte array is copied into a disk file. The platform dependent media player is to be invoked to play the video. The client application will invoke the media player as plug-in using Runtime reference as shown below:

```java
Runtime rt = Runtime.getRuntime();
Process p = rt.exec("c:\program files\windows\wmplayer.exe " +
                   videoFilename);
```

The videos of different file formats and of different sizes can be invoked through this service with memory constraints. As the proposed VoD model in JAX-RPC uses XML-based SOAP protocol for exchange of messages, it enables interoperability with other Web services and the client need not be necessarily in Java platform.

3.5 SOAP COMMUNICATION MODEL FOR VIDEO ON DEMAND

The SOAP communication is emerging as a standard way to send XML documents over the Internet from different platforms and thus enhancing the interoperability of Web services. XML Web services are already used as middleware components for interacting Internet applications. The SOAP envelope is structured in XML as well and will be delivered by an arbitrary protocol, by default HTTP.

The SOAP with Attachment API for Java (SAAJ) conforms to the Simple Object Access Protocol specification and the SOAP with Attachments specification. To increase the security of the video contents and to permit different distributed services to run on various platforms, a SOAP communication model for the Video on Demand has been proposed with
security enhancements. The block diagram representation of the proposed SOAP communication model is given in Figure 3.8.

![Figure 3.8 SOAP Communication Model for VoD Service](image)

The XMLised Blob data is encapsulated within the SOAP envelope and also can be included in the SOAP message as attachment in the proposed model. In the SOAP communication model with regular message exchange pattern, the XMLised Blob data is included as an attachment with the SOAP message and in the proposed multicast message exchange pattern the data is kept within the SOAP envelope. As video piracy is the major concern, the proposed SOAP communication model for Video on Demand system ensures authentication of the client while delivering the video contents.

### 3.5.1 Video Service Provider

The video service provider is designed in such a way that it will deliver video contents only to the authenticated SOAP clients. But at the same time, it is decoupled from the authentication code. The authentication is earmarked to an Authentication Service Provider, which uses an extended RPCMessageReceiver that will decide the type of the SOAP clients i.e., a video provider or a video requester. Based on the type of the client, the read or write permission will be given in the form of a Message ID. The video
service provider uses this Message ID to allow the client either to upload or download the video contents. Both authentication and video service providers use request-response message exchange pattern, in which both request and response messages are XML based SOAP envelopes created using AXIS based Object Model (AXIOM) i.e., OMEElement references.

The video service provider has two methods whose signatures are as follows:

```java
public OMEElement getVideo(OMElement request);
public OMEElement uploadVideo(OMElement request);
```

The parameter to the getVideo() method contains the name of the video file to be retrieved and the response contains the Video ID of the SOAP attachment. In case of uploadVideo() method, the input parameter contains the Video ID of the video data to be uploaded onto the server and the response contains the status of upload.

### 3.5.2 Modified Message Exchange Pattern for Video Service Provider

The RawXMLInOutMessageReceiver associated with the video service provider invokes the getVideo() or uploadVideo() method based on the Message ID as sent by the client. The predefined invokeBusinessLogic() behaviour of RawXMLInOutMessageReceiver invokes findOperation() behaviour based on the information contained in the Message ID.

The invokeBusinessLogic() method has two MessageContext parameters, namely, msgContext and newmsgContext, where msgContext represents the SOAP request from the client and newmsgContext represents the SOAP response. The Message ID as provided by client is parsed within
the invokeBusinessLogic() in order to find the appropriate service to be invoked. If the operation to be performed is defined within the video service provider, the invokeBusinessLogic() invokes the method using reflective middleware and receives the response as OMElement reference from the method and uses this response to formulate the SOAP envelope of the newmsgContext, which is its response parameter.

The behaviour of invokeBusinessLogic() is suitably modified, which is given as below and the same is incorporated in the RawXMLINOutMessageReceiver.java source code. The modified version of RawXMLINOutMessageReceiver class file is copied into axis2-kernel-1.4.1.jar file, which is included in the axis2.war. The modified war file is deployed in the Tomcat Web Server and hence an exclusive Message Exchange Pattern for Video on Demand System is achieved.

```java
SOAPBody body = msgContext.getEnvelope().getBody();
String messageID = body.getFirstElement().getText();
...
String fileName = decrypt(messageID).nextToken();
if(decrypt(messageID).nextToken().equals("read"))
Method m = findOperation(op, VideoService, "read");
// op is the AxisOperation and VideoService is the actual video
// service implementation class
// The method reference ‘m’ here refers to getVideo() method.
```

The authentication service provider is associated with RPCMessageReceiver of org.apache.axis2.rpc.receivers package, which is bundled within the axis2-kernel-1.4.1.jar and there is no change in the
message exchange pattern. The SOAP response of authentication service provider is the SOAP request for the video service provider. The authentication service provider has encapsulated with only one behaviour whose signature is given as follows:

```java
public OMElemfent authenticate(OMElement request);  
```

The SOAP request to the authentication service provider is as follows:

```xml
<video:authenticate xmlns="http://video">  
<video:videofile>song.mpg</video:videofile>  
<video:servicetype>retrieve</video:servicetype>  
<video:password>u5m9v8d4</video:password>  
<!--password is encrypted-->  
</video:authenticate>
```

If the authentication is success, the SOAP response which will be sent to client is as follows:

```xml
<video:authenticateResponse xmlns="http://video">  
<video:messageID>x7d5m8v6lcfadbfa</video:messageID>  
<!--the messageID has two parts: name of the data source and permission-->  
</video:authenticate>
```
The authenticate() method of authentication service provider is defined as follows:

```java
SOAPBody body = msgContext.getEnvelope().getBody();
OMElement element = body.getFirstChildWithName(new QName("http://video", "authenticate"));
OMElement videoElement = element.getFirstChildWithName(new QName("http://video", "videofile"));
String fileName = videoElement.getText();
OMElement service = element.getFirstChildWithName(new QName("http://video", "servicetype"));
String serviceType = service.getText();
OMElement key = element.getFirstChildWithName(new QName("http://video", "password"));
String password = decrypt(key.getText());
PasswordControl pc = new PasswordControl(password);
```

The built-in security manager calls checkRead() or checkWrite() method implicitly while trying to perform input or output operation on the specified file and then assigns a file permission by creating a FilePermission reference. The PasswordControl class, which is extended from SecurityManager class redefine the methods of checkRead() and checkWrite() methods originally defined in its base class. This security manager will check the user-defined permission for the requested file with the overall system permissions, which are stored in “.java.policy” file. Even if the given password by the client is authenticated, the security manager fails to approve the authentication when the system policy does not permit read or write access to a specific file.
Based on the “service type” and authentication status, the authentication service provider will create a messageID with three attributes: read, write and fail. The “messageID” is an encrypted string with two components, the name of the video file and the permission attribute concatenated with the pipeline symbol. The corresponding SOAP response is sent back to the client in order to access or to upload the video contents. If the authentication fails, the video service provider will send an “access denied” message.

3.5.3 Implementation of the Video Service Provider

The FileDataSource class of javax.activation package will provide data typing services. Using this instance, the MIME type of the requested data is obtained. Creating a FileDataSource instance does not open the file. A FileDataSource reference implies an abstraction of collection of data and it encapsulates a file of any content type. A DataHandler object can be created in association with the FileDataSource. The FileDataSource provides an InputStream to the DataHandler in order to access the data and the data available are in byte stream form. While FileDataSource does the data typing services, the DataHandler is used to copy the contents to another disk file or to make the byte stream readily available onto the associated OutputStream.

The DataHandler associated with a FileDataSource provides a faster response to handle multimedia data as the data is readily available in byte stream form in the implicit InputStream provided to the DataHandler by the FileDataSource. When compared to the explicit conversion of multimedia data to sequence of bytes using regular file handling procedure and then transferred to the desired location using OutputStream, an average of 40 percent time is saved while transferring multimedia data using DataHandler. Another major advantage is that there is no need to use the extended memory as is the case with respect to regular file handling procedure, as the design of
String fileName = element.getFirstElement().getText();
FileDataSource fds = new FileDataSource(fileName);
DataHandler dh = new DataHandler(fds);

The getVideo() method accepts the video filename as request in the form of OMElement that fetches the requested video and sends it back to the client as OMElement. OMElement is an AXIS2 representation of XML node, it provide a tree model like the Document Object Model (DOM). The code segment which will make the video contents in a readily deliverable form is given below. It is designed in a way that the SOAP response of the authentication service provider is given as SOAP request to video service provider, which contains an encrypted messageID. The received messageID is parsed within the message exchange pattern, i.e., within the invokeBusinessLogic() of RawXMLINOutMessageReceiver which in turn invokes the appropriate method using findOperation() and reflective middleware.

For each and every request sent to the server, a MessageContext is created which is bound to the OperationContext, which represents the running ‘instance’ of an operation. It allows the messages to be grouped into operations and when these messages are being exchanged, the OperationContext remembers the state of where the messages are in the message exchange pattern. The “inMessageContext” object of MessageContext is created to keep track of the client from whom the request has come. In order to send the requested video to the same client, an OperationContext object is created using the inMessageContext, and then an
outgoing MessageContext “outMessageContext” is created by passing a WSDL constant MESSAGE_LABEL_OUT_VALUE, which refers to “out” parameter, as shown in the code below:

```java
MessageContext inMessageContext =
    MessageContext.getCurrentMessageContext();
OperationContext operationContext =
    inMessageContext.getOperationContext();
MessageContext outMessageContext =
    operationContext.getMessageContext(WSDLConstants.MESSAGE_LABEL_OUT_VALUE);
```

The video file reference has to be attached to the message and a response has to be formulated in the form of XML. The requested video is in the form of byte stream and it is readily available with the DataHandler. Using the outMessageContext reference, the DataHandler Object is passed as parameter to the addAttachment() method, which returns a unique ID as a String reference to refer the attached byte stream. The fetched video ID is sent to the client in the form of XML which has been created using OMFObject. Finally the generated XML document “wrapperElement” is sent to the requested client through the message receiver. The definition of getVideo() method is as follows:

```java
String videoID = outMessageContext.addAttachment(dh);
OMFactory factory = OMAbstractFactory.getOMFactory();
OMNamespace omNs = factory.createOMNamespace("http://video", "swa");
OMElement wrapperElement =
    factory.createOMElement("getVideoResp", omNs);
OMElement videoElement =
    factory.createOMElement("video", omNs, wrapperElement);
videoElement.addAttribute("href", videoID, null);
```
### 3.5.4 Service Descriptor

The video service has been created and the description of service is specified in services.xml as shown below:

```xml
<serviceGroup>
  <service name="VideoService">
    <description> Video Service – SOAP with Attachment Model </description>
    <parameter name="enableSwA">true</parameter>
    <parameter name="ServiceClass" locked="false">video.VideoService</parameter>
    <operation name="getVideo">
      <actionMapping>urn:getVideo</actionMapping>
      <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
    </operation>
  </service>
</serviceGroup>
```

The name of the service is “VideoService”, which will be the name of the archive file. Two Parameters have been specified which will be transformed into service properties in the corresponding AXIS Service. The above transformation is carried out by the “ServiceClass” which specifies the service implementation. The specified Java class “video.VideoService” is loaded by the MessageReceiver when required. The <operation> tag specifies the public method “getVideo()” to be invoked based on the client request.
3.5.5 Invoking the Video Service from the Client

The “VideoService” is deployed in the Web server in order to enable the clients to access the required Videos on demand. The corresponding WSDL file or the EndpointReference can be located using the following URL: “http://localhost:8000/axis2/services/VideoService”. The client sends a SOAP message to the server requesting for the required video file name, “videofilename” which is attached to an OMElement. The SOAP Envelope for the SOAP Request Message is created using SOAPFactory, as shown below:

```java
SOAPFactory fac = OMAbstractFactory.getSOAP11Factory();
SOAPEnvelope env = fac.getDefaultEnvelope();
OMNamespace omNs = fac.createOMNamespace("http://video","video");
OMElement method = fac.createOMElement("getVideo", omNs);
OMElement param1 = fac.createOMElement("videoName", omNs);
param1.setText(videofilename);
method.addChild(param1);
env.getBody().addChild(method);
```

The OperationClient object provides direct access to the MessageContext object, which sends the SOAP request to the VideoService, as shown in the code below:

```java
OperationClient mepClient =
client.createClient(ServiceClient.ANON_OUT_IN_OP);
MessageContext mc = new MessageContext();
mc.setEnvelope(env);
mepClient.addMessageContext(mc);
mepClient.execute(true);
```
The SOAP Request is sent to the getVideo() method of the VideoService, which retrieves the video filename from the message and retrieves the corresponding video file from the server and sends the reference i.e., video ID as a part of SOAP response to the client. The client can retrieve the requested video file using the video ID in association with the DataHandler. After the execution, the incoming message can be accessed using MessageContext that is bound to the OperationClient. The WSDL constant MESSAGE_LABEL_IN_VALUE represents the “in” parameter, which is used to retrieve the contents from the InputStream associated with the MessageContext as shown below.

```java
MessageContext response =
    mepClient.getMessageContext(WSDLConstants.MESSAGE_LABEL_IN_VALUE);
```

The video file in the form of binary content is available as SOAP attachment. Using the video ID, which has been sent by the VideoService, the DataHandler retrieves the attachment and provides the requested video file to the client and the code used to retrieve the video file is given below:

```java
SOAPBody body = response.getEnvelope().getBody();
OMElement element = body.getFirstChildWithName(new QName("http://video","getVideoResp"));
OMElement videoElement = element.getFirstChildWithName(new QName("http://video","video"));
String videoID = videoElement.getAttributeValue(new QName("href"));
DataHandler dataHandler = response.getAttachment(videoID);
if (dataHandler!=null){
    // The Video File is downloaded and played to the client.
}
```
When the client requests for the video file “song.mpg”, the SOAP request is generated and sent to the server, which is shown below. The SOAP request contains the video filename and the method to be invoked.

```xml
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
     <swa:getVideo xmlns:swa="http://video">
       <swa:videoName>song.mpg</swa:videoName>
     </swa:getVideo>
   </soapenv:Body>
</soapenv:Envelope>
```

The following XML document shows the SOAP response obtained based on the SOAP request. The SOAP response contains the reference video ID, using which the client can retrieve the video from the DataHandler.

```xml
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
     <swa:getVideoResponse xmlns:swa="http://video">
     </swa:getVideoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```
In this SOAP communication model, the SOAP request is transferred to the message receiver using the OperationClient’s execute() method. The OperationClient will get the target endpoint reference using the options provided by the requesters. The execute() method will take the request to the RawXMLINOutMessageReceiver and then calls the executeImpl() on the server side. The whole responsibility of executeImpl() is to implement in-out message exchange by appropriately creating the ‘in’ and ‘out’ MessageContexts. The executeImpl() method in turn calls the invokeBusinessLogic() of the message receiver by passing the ‘in’ and ‘out’ message contexts as parameters. The ‘in’ message context actually contains the request and the ‘out’ message context holds only the socket reference initially. It is the responsibility of the invokeBusinessLogic() method to call the required service that will return the response as OMElement. The SOAP response is observed within the invokeBusinessLogic() and the same is encapsulated within a SOAP envelope of ‘out’ message context. The client receives the SOAP response and extracts the video data from the attachment as referred by video ID and will play it in a video player, which is plugged-in onto the client application.

3.6 SOAP WITH MULTICASTING VIDEO ON DEMAND

The recent growth in use of the World Wide Web especially for multimedia applications in the Internet has caused a significant increase in the demand placed on Web servers. This increased load results in noticeably not only longer response times for users but also increased use of network bandwidth. The key problem in deploying large-scale Video on Demand applications is economy rather than technology. In order to have an efficient and cost effective Video on Demand system it is important to reduce the bandwidth requirements as much as possible. A typical Video on Demand system uses dedicated channels to provide the service to every user request.
This increases the bandwidth requirement as more and more streams are requested. Multicasting is an approach whereby various customers can share a single movie stream resulting in reduced system cost per customer and improved system scalability. Multicasting has some limitations associated with it. For example, in order to share a single channel with a group of customers, all the customers have to request the video at the same time and watch it without any interaction (no pause, fast forward, rewind, etc). This defeats the purpose of a true and interactive Video on Demand system. There are certain multicasting techniques which can be used to provide interactive Video on Demand services.

Batching is one of the multicasting techniques in which the requested video is intentionally delayed by some amount of time, called a batching interval, so that subsequent request for the same video arriving during the current batching interval may be serviced using a single I/O stream. This trade off reduced I/O stream requirements for increased latency. Therefore, large batching intervals would seem to be incompatible with the notion of a true VoD system. Batching can only be used with popular videos since unpopular videos are unlikely to receive multiple requests during the short delay interval (Steve Spicer and Santosh Kulkarni 2001).

Patching is another multicasting technique in which the multicasting stream is not static but can expand dynamically to accommodate new requests. For example, if there is a video being streamed through a local video server and there is a new request for the same video, then the video server starts caching the video data in its local disk. The server therefore needs an initial patching stream to transmit the leading portion of the video. When the leading portion of the video is transmitted, the rest of the video is transmitted from the data already buffered in the local disk (Kien 1998).
Using the patching technique, channels are used only briefly to broadcast the first few minutes of the video, instead of being held up for the entire duration of the playback. Patching can be seen as a multicasting technique for interactive Video on Demand systems where a single channel can be shared with multiple users. Patching also saves the storage costs as only a small amount of the video stream has to be buffered at the local disk depending on the requests. If there are new requests for the same video title then patching is used for that time period. Patching is very effective in reducing the bandwidth and storage requirements if the number of interactive requests from the users is within a certain limit. Beyond that, patching loses its efficiency as it results in starting multiple patches of the same video and increases the bandwidth requirements.

In piggybacking approach, the playback rates are altered for the ongoing video streams by 5% in order to merge the respective video streams into a single stream, which can then serve the entire group of requests. The idea is to display the leading stream at a slower rate, and the trailing stream at a faster rate. Then, assuming this interval is sufficiently small, the faster stream will eventually catch up with the slower stream. At this point the streams can be piggybacked or merged (Golubchik 1995).

There are different piggybacking policies that can be used to achieve maximum savings in bandwidth. The three basic types of piggybacking policies are simple merging policy, the odd-even policy and the greedy policy. The first two are elementary as they involve at most a single change of speed for each video stream. Therefore they can achieve a maximum of 50% improvement in the number of streams saved, because the subsequent streams are paired. The greedy policy changes the speed of the different streams in such a way that the streams are merged at the earliest possible time in order to achieve maximum savings in bandwidth.
In volatile storage approach, every single video title that is transferred to a video server is cached in the server’s local disk completely. Apart from its local storage, each video server has a volatile storage where the video content is stored for a short period of time. This procedure enables any future requests to be served from the local server without any additional video streams required from the distant server (the only exception is when a future request is for a part of the video, which is still not present in the local volatile storage). There are no restrictions in the number of interactive requests as all the requests are satisfied from the local video server. The only drawback with this approach is that it increases the storage cost as additional volatile storage is required at each video server connected to a network.

The current Web services standards are not sufficient to cater for multimedia streaming, the commonly used delivery method for large multimedia objects. Lam et al (2008) have proposed SOAP based streaming content delivery framework for multimedia Web services. The framework uses an extension of existing SOAP standards, which allows streaming content to be transmitted between two SOAP nodes over HTTP protocol. The extension includes a new SOAP streaming message exchange pattern and its corresponding SOAP HTTP binding.

The proposed framework is capable of delivering quality adaptive multimedia content to users with heterogeneous configurations and requirements. When a multimedia file is requested, the current Request-Response message exchange pattern is not sufficient to handle one request and multiple responses. The authors also proposed a service oriented architecture for the delivery of streaming multimedia, which follows the standards of Web services. The purpose of this model is to unify the e-learning platform together with streaming multimedia in the Web services domain.
Hence, in order to reduce the load on the servers and to minimize the bandwidth requirements, the current research in this thesis is focused on the use of multicasting in the delivery of video contents by enhancing the conventional MessageContext of SOAP communication in Apache eXtensible Interaction System (AXIS). The existing “in-out” message exchange pattern is extended to “multicast in-out” message exchange pattern within the RawXMLINOutMessageReceiver of the SOAP communication system in addition to the regular “in-out” scheme. The current MessageContext creates a new connection for every SOAP request and the latency associated with SOAP communications is huge while handling multimedia data. A multicast SOAP communication would have been a better solution to reduce the load on the server and to minimize the bandwidth requirements.

In the proposed Video on Demand model using SOAP communication, the requests arriving for a particular video file are grouped but with restriction in the size of the group and the response is transmitted using a multicast connection from the server to the requested clients. The architecture of the proposed Multicast Video on Demand System using SOAP Communication model is shown in Figure 3.9.

![Figure 3.9 Proposed Multicast SOAP Communication model for VoD system](image_url)
The multicast connection is an additional behaviour that has been included in the MessageContext, which has already been encapsulating a regular socket connection. This is achieved by introducing a new class MulticastMessageContext, which extends MessageContext. Thus, the MulticastMessageContext provides regular HTTP response within the SOAP envelope if the request is for other than multimedia data. The requests are always sent to the server via an HTTP connection. The server classifies the requests as “requests for multimedia data” and “requests for non-multimedia data” using data typing services provided by FileDataSource. Based on the classification, the response is decided either to use Socket connection or MulticastSocket connection.

The proposed model is designed in such a way that during a specific period of time, if there is only one request for the multimedia data, the response is sent back using the same HTTP connection that had carried the request. A vector of video/audio file names are maintained in the server and if there are requests for these data, the requests are grouped and a multicast connection is established for each group. If there are multiple requests simultaneously within a specific period of time from various users to the same multimedia data, whose file name is included in the vector, the multicast connection is established. The proposed system also supplements the multicast service to the registered clients for a specific type of multimedia data. The overhead while making the decision of using regular socket connection or multicast socket connection is negligible while compared to bandwidth requirement due to single HTTP connection for every request and the load on the server while transferring large amount of same data repeatedly. The server is forming a multicast group for those clients requesting the same multimedia data. A single multicast IP address is used for a multicast group and all the clients need to explicitly join that group. The entire request URLs are mapped to that multicast IP address. The major
advantage of the proposed model is that the client can have its own option by sending an indication to the server that it will support a multicast response for its request. The server listens for connection requests and receives HTTP requests once a connection is established. When the requests are for the same multimedia data then a new connection is established with the multicast IP address in order to satisfy those requests. If the server receives a request other than multimedia data, the request is satisfied using regular way of unicasting over the request connection.

3.6.1 Implementation of Multicast Message Exchange Pattern

Web Services use SOAP for messaging. SOAP is an XML protocol that provides an envelope, which encapsulates XML data for transfer through the web infrastructure. SOAP-over-UDP decouples SOAP from its underlying layers and defines a binding for SOAP envelopes to UDP. It uses Video Service Endpoint Address to support the message exchange patterns as SOAP-over-HTTP. In the current Web service, the exchange of information is of single request and single response only. The SOAP Message Exchange Pattern (MEP) defines properties for exchange of SOAP messages required by a communication protocol HTTP. The two message exchange patterns namely Request-Response MEP and Response MEP are suitable for RPC representation of Web services where for each request there is only one response. This is not suitable for multicasting the video files as the video contents are divided into multiple small segments and there should be multiple responses. The requirements for implementing the multicast MEP are given below:

- The request is given only for joining the group and not for fetching the video. Once the client joins the group, the video has to start
playing from the same scene that the other users in the group are currently viewing.

- As the messages are sent through UDP, the video files are divided into small chunks of data that the UDP packet can hold. The message exchange should support multiple responses for a single request.

A new SOAP message exchange pattern is required for video multicast in order to send multiple SOAP responses in a single HTTP response. This can be achieved by adopting the multipart structure in the Multipurpose Internet Mail Extensions (MIME). Hence, the proposed video multicast uses SOAP for message exchange between the video service provider and video requester by adopting multipart/related MIME type. The video file is divided into small chunks of 48KB, less than the size an UDP packet can hold. These packets are transmitted one after the other from the server to the clients continuously. The client receives these packets and starts playing the video file. The workflow of the proposed SOAP MEP for video multicast is shown in the Figure 3.10.

![Figure 3.10 WorkFlow of the proposed SOAP MEP for video multicast](image-url)
The proposed multicast video service provider has two methods whose signatures are given below:

```java
def getVideo(element request):

def multicastVideo(element request):
```

The input parameter of both methods has `<responseType>` tag, which has the value either “regular” or “multicast”. The RawXMLINOutMessageReceiver handles the request if the client does not support multicast response. The customized in-out message exchange pattern associated with RawXMLINOutMessageReceiver in order to decide unicast or multicast communication is coded as follows:

```java
SOAPBody body = msgContext.getEnvelope().getBody();
OMElement element = body.getFirstChildWithName(new QName("http://video", "responseType");
String resp = element.getText();
if (resp.equals("regular"))
Method m = findOperation(op, VideoService,"regular");
```

If the response type of the request given by user is “multicast”, the request is transferred to MulticastMessageContext, which is being associated with MulticastMessageReceiver, a new class with the same behaviours as available in the RPCMessageReceiver namely, findOperation() and invokeBusinessLogic(). The modified method signature of invokeBusinessLogic() is given as below:

```java
public void invokeBusinessLogic(MessageContext msgContext,
MulticastMessageContext newmsgContext) throws AxisFault;
```
The MulticastMessageContext class encapsulates MulticastSocket along with regular Socket as it is extends from SOAP MessageContext. In this case, there will be no change in the in-out message exchange pattern. In case of “multicast” response type, i.e., if the client supports multicasting, the request to the MulticastMessageReceiver is to join the multicast group. The model triggers the multicastVideo() service once the specified number of clients joined the group.

The server starts the video multicast through the Multicast IP address and a specific port. The clients who wish to view the video have to join the above multicast IP address to receive the video. Once the server starts streaming the video, the video data is divided into multiple datagrams and sent using SOAP with Attachment to the client, who receive the datagram and start playing the video. Thus, the four main operations of the MulticastMessageContext of the video multicast are as follows:

- Clients has to Join the multicast group using JoinGroup().
- The Server sends the video to the members of the group.
- The clients receives the video from the group.
- After the video multicast stops, the client leaves the multicast group.

SOAP MessageContext is extended by integrating multicast features for handling viewers of similar interest in a distributed environment. Multicast extends the traditional unicasting with efficient multipoint communications in which data can be sent to a set of destinations simultaneously belonging to a particular group. By sending same video content to multiple users, the better bandwidth utilization can be achieved with less host processing.
The MulticastSocket is used to design multicast communication for the video service. The DatagramSocket class is used to create UDP socket connections that sends and receives the video data using DatagramPackets. The video client and server communicate over a UDP connected to a port on their machines. The client requests for a video file through a SOAP message and the corresponding video data is sent over DatagramSocket using DatagramPacket and each DatagramSocket contains a data buffer, the address of the remote host to send the data, and the port number the server is listening on.

The client request is implemented through a HTTP operation where the required video file name is sent to the server and the transmission of this operation will be in request-response message exchange pattern. The SOAP message is sent to the multicast address. For the multicast transmission of video, the multicast service provider sends the SOAP message with the same attachment to the multicast address, thus all clients will receive the same packet. The SOAP message over UDP restricts the size of the message to be transmitted over the network. Hence, the video data is divided into multiple small packets before transmitted.

3.7 CLOUD SERVICES FOR VIDEO ON DEMAND

To provide an efficient and reliable Video on Demand system, which should be easy to use, easily accessible from heterogeneous environments without any geographical restriction, a cloud based Video on Demand service is proposed.

The Web Server that holds the Video on Demand service has a fixed IP address, which is known to all the devices connected to the Internet and hence the server can be identified easily. In contrast, when the VoD
service is deployed in the cloud environment, the cloud server does not have a fixed IP address. The video data shall be stored in a single server or multiple servers depending on the number of users accessing the video service. When a client requests for a specific video file, the cloud identifies the server that contains the VoD service to serve the client with best Round Trip Time.

When the video service is requested by many clients, the video service is deployed in many servers dynamically and the client can access the video service from the nearest server. In order to provide a scalable Video on Demand service in a platform independent manner, the top two PaaS (Platform as a Service) providers namely Google App Engine (GAE) and Azure are selected to deploy the VoD Service.

### 3.7.1 The proposed VoD model in Google App Engine

The proposed Video on Demand system is designed and deployed in the Google App Engine (GAE) that utilizes the Google infrastructure. GAE provides a complete cloud based integrated development environment that enables to design cost effective and scalable Video on Demand Services. The main characteristic of the proposed model is that it can scale up and down without any changes in the infrastructure or in the application for any number of users at a particular instance.

The GAE allows dynamic allocation of system resources for the VoD service based on the actual demand. When the video requester accesses the proposed cloud based VoD service, the request has been forwarded to the load balancer that will identify the appropriate server and allocate the system resources dynamically based on the number of the requests. The implementation of the Video on Demand Service in GAE is given in Figure 3.11.
The App Engine Software Development Kit for Java is used to create and deploy the Video on Demand service in the cloud environment. Before deploying the VoD service in the cloud, an application identifier “cloudcinemas” has to be reserved within the cloud platform and also necessary storage and computational power are to be allocated. The VoD service implementation and its configuration and all the other necessary files are packaged and deployed in the allocated cloud’s store with reference to the identifier. Thus the video requesters can access the VoD service through the URL “http://cloudcinemas.appspot.com”.

The policies regarding accessing of VoD service is created by the service provider. The authorized clients can download and play the required video from the VoD service provider. Two configuration files are required for deploying the service in the cloud environment namely, the service deployment descriptor and an appengine based configuration file “appengine-
web.xml”. The service deployment descriptor contains the name of the index file that is to be loaded to the client when the request is triggered and it is defined as follows:

```xml
<web-app
xmlns="http://java.sun.com/xml/ns/javae"version="2.5">
<welcome-file-list>
   <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```

When the video requester accesses the VoD service from the URL “http://cloudcinemas.appspot.com”, the home page lists the video catalogue from which the client can choose the video they want on demand. The appengine configuration file “appengine-web.xml” contains the reserved application identifier “cloudcinemas”, the namespace and the version number of the VoD service. During deployment of the VoD service to the cloud, the specified application ID instructs the GAE to upload the packaged files to the cloud that is pointing to the VoD hosting service. The appengine configuration file for the VoD service is shown below:

```xml
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app
xmlns="http://appengine.google.com/ns/1.0">
<application>cloudcinemas</application>
<version>1</version>
</appengine-web-app>
```

The <application> element contains the application identifier “cloudcinemas”, that is reserved to deploy the VoD Service. The Video on Demand service is tested locally using the “dev_appserver” (development
server), which is a simulator for the GAE cloud Environment and services, including sandbox restrictions and the datastore. The appcfg tool is used to upload and deploy the VoD service to the real Google cloud infrastructure and makes the service available to the client globally. When the video requester accesses the URL “http://cloudcinemas.appspot.com”, the appengine server responds to the Web request, by identifying the domain name “cloudcinemas”. The Video on Demand service is fetched to the client from the server that provides the service anytime and anywhere.

### 3.7.2 VoD Model using Azure Cloud

Windows Azure provides developers with on-demand compute and storage to host, scale, and manages Web applications on the Internet through Microsoft datacenters. Unlike Google App Engine that supports only Java and Python, Windows Azure is a flexible platform that supports multiple languages and also supports integrating with the existing on-premises environment. In addition, Windows Azure supports popular standards, protocols and languages including SOAP, REST, XML, Java, PHP and Ruby and is now commercially available in 40 countries. Azure is an operating system for the cloud services that serves as the platform for developing, hosting and managing the services, through which it is proposed to develop Video on Demand services.

The Azure Software Development Kit is used for designing, developing, configuring and deploying the VoD service to the Azure Cloud. The VoD service is developed in .NET framework and is simulated locally in the Development Fabric, which is then deployed to the Azure Fabric. The implementation of the Video on Demand service along with the configuration files are coalesced with a WebRole named as ‘VoDRole’, which makes the connections to the Video on Demand service through HTTP. The role refers
to the Virtual Machine (VM) image, where the Video on Demand service is hosted and hence it is the endpoint reference to the clients accessing the VoD service. The role has been configured to run on top of two VMs and all the requests from the clients are equally distributed to these instances through the Load Balancer. These virtual machines, which run 64-bit Windows Server 2008, are responsible for allocating CPU time and memory on thousands of physically separated servers and it is being monitored by the “Hyper-V” hypervisor. The implementation and the configuration files are bundled together and using the ‘publish’ command which in turn calls the ‘cspack’ tool for creating the Cloud Service (CS) package that is used for deploying the Video on Demand service to the cloud environment. The deployed Video on Demand service can be accessed from the URL “http://cloudcinemas.cloudapp.net/index.aspx”. The identifier ‘cloudcinemas’ in the URL has already been registered with Azure to host the developed VoD service. The architecture of the proposed Video on Demand service in Azure Cloud is shown in Figure 3.12. The Video on Demand service using download and play approach is implemented using the Azure in order to provide a secure and cost effective service.

Figure 3.12 Cloud Architecture for Video on Demand in Azure Environment
The two configuration files namely, service definition and service configuration are required, which specify few parameters like the endpoint address and the number of instances to be created. The service definition, which cannot be changed during runtime, defines the WebRole and the communication endpoint for the VoD service. The service configuration is configured at runtime and comprises of the complete configuration details needed for the WebRole instances on which the VoD service would be running. The service definition for local deployment of the VoD Service is given below:

```xml
<ServiceDefinition
    name="videoservice" xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/ServiceDefinition">
    <WebRole name="VoDRole">
        <InputEndpoints>
            <InputEndpoint name="HttpIn" protocol="http" port="80" />
            <InputEndpoint name="HttpsIn" protocol="https" port="8000" />
        </InputEndpoints>
        <ConfigurationSettings>
            <Setting name="DiagnosticsConnectionString" />
        </ConfigurationSettings>
    </WebRole>
</ServiceDefinition>
```

The Video on Demand service allows the VoDRole only to receive requests from two endpoints namely HTTP (port 80) and HTTPS (port 8000), thereby filtering out the other requests reaching the service. Thus, the Video on Demand service is protected from the port attacks and the distributed denial of service attacks. In addition to the service definition file,
a service configuration file is required to determine the configuration of the VoD service and the details pertaining to the deployment of the service. The configuration includes information like the number of instances in a VoDRole and other operational characteristics, which is shown below:

```xml
<ServiceConfigurations>
  <Role name="VoDRole"><Instances count="2" />
    <ConfigurationSettings>
      <Setting name="DiagnosticsConnectionString" value="UseDevelopmentStorage=true" />
    </ConfigurationSettings>
  </Role>
</ServiceConfigurations>
```

The Service Management API is used to programmatically deploy, upgrade and configure the Video on Demand Model to the Azure Cloud. In the developed Web services and SAAJ model, it is necessary to manage various components such as servers, networking, storage and the VoD service, where as in the cloud environment, there is no necessity to maintain the hardware resources. As Azure environment incurs charges for the hosted VoD Service, it is temporarily removed from the cloud platform.

3.8 PERFORMANCE ANALYSIS

The distributed services for the VoD system is designed to support download and play approach of the video contents from the server. It is proposed to estimate the Round Trip Time (RTT) for the developed Video on Demand models using RMI, JAX-RPC, SOAP communication and Cloud
Environments by requesting the video files of various sizes in local and remote environments. Even though RMI based VoD model is not reported, it is being taken into account since RMI is the base for all proposed distributed services.

RTT is the time that elapses between the initiation of a video request by the clients till the specified video is played to them. The RTT includes network time, Web server time, application server time and database server time. To analyze the performance exactly, the services are invoked for specified number of times for each video file of different size, and the average of the obtained measures are considered for analyzing the performance of the proposed services.

3.8.1 Local Invocation Performance

Initially the performance analysis is being carried out for the distributed VoD services by considering both client and server are being executed in the same system. All local tests are being carried out on the system configured with Dual Core Processor at 2.00GHz, and 2GB RAM running the Windows Vista Ultimate Operating System. The Java Developer Kit 1.5 is used to develop the VoD service in RMI and JAX-RPC models. The Apache AXIS 2.1.4is used to develop the VoD Service in SOAP communication Model. The development fabric and development app server are used to deploy the VoD Services locally to simulate the performance in the Azure cloud and the GAE cloud environments.

The Figure 3.13 shows the average Round Trip Time in milliseconds for the download and play approach of VoD services in different distributed environments. RTT is estimated by triggering the VoD services
locally to retrieve the video files of varying sizes starting from 1MB to 30MB.

![Graph showing RTT for different environments](image)

**Figure 3.13 Local Invocation Performance of VoD Service in different environments**

The performance analysis of the distributed services, while invoking the VoD service locally shows that the RTT increases as the size of the video file increases. It is observed that the Remote Method Invocation, which supports only Java to Java communication, gives better performance when compared to all the other distributed environments locally. Both JAX-RPC and SOAP communication models use XML representation for message transfer, whereas the RMI uses binary encoding for the video files, which increases its performance. The VoD service deployed in Cloud Environments shows faster response from the server, when compared to the JAX-RPC and SOAP communication model. As interoperability is the major concern, the overheads associated with proposed distributed VoD services can be neglected.

### 3.8.2 Remote Invocation Performance

The performance measure, Round Trip Time (RTT) is estimated for the developed Video on Demand models using RMI, JAX-RPC, SOAP communication and cloud environments by requesting the video files of
various sizes from remote system. Computers connected in a LAN via 100BaseT Ethernet with a normal network traffic load and each system in the LAN having a hardware configuration of Dual Core Processor at 2.00GHz, and 2GB RAM are used for testing the performance. The VoD service is deployed to GAE and Azure cloud environments to test the remote invocation performance. The Figure 3.14 shows the Round Trip Time obtained for different sizes of video files starting from 1MB to 30MB deployed in different distributed environments. The Figure 3.15 shows the Local and Remote Invocation Performance for VoD Service in JAX-RPC model.

![Figure 3.14 Remote Invocation Performance of VoD Service in different environments](image)

![Figure 3.15 Comparison of Performance of Web Service Model for Video on Demand in both local and remote scenarios](image)
The interoperability issue is a key factor in Remote Method Invocation whereas the JAX-RPC and SOAP Communication models support cross-platform. When the video services are invoked remotely using JAX-RPC and SOAP communication models, the connectivity and the interoperability are not at all issues, as these models use the XML based SOAP communication between remote services. In the remote invocation of the Video on Demand service developed using JAX-RPC, the RTT is very high. The overheads of inherent XML encoding of response video data now becomes more apparent and thus the performance of JAX-RPC lags. The performance measure RTT for the remote invocation of the download and play approach of VoD service is better in cloud environments when compared to the other distributed environments.

In order to avoid piracy issues, commercial video providers prefer allowing the users to view the video files on-line on demand instead of allowing the video files to be downloaded to the end users. In order to achieve this, either streaming or progressive download would be the only option. Since, the streaming approach requires a dedicated streaming server, which is very costly, the progressive download has been considered which is economical and also streams the video files in a similar way as that of a streaming approach. Hence, the progressive download approach has been implemented in both GAE and Azure cloud environments, where different video files of varying sizes have been considered to analyze the performance in terms of RTT. The result delineates that the performance measure RTT for fetching Video on Demand through progressive download approach in Windows Azure environment is better than the Google App Engine. The research concentrates on progressive download approach, where the video starts playing, when sufficient amount of video contents are downloaded to the client. That is, both downloading and playing of the video file occurs without significant delay. By this approach, there will not be any memory or
buffer problems, as it presumes that the client has sufficient memory and
buffer space to save the downloaded file.

3.9 CONCLUSION

An effective JAX-WS based Web Service has been developed for
dynamic XMLised blob data generation for solving the data exchange
problem in heterogeneous environments. The proposed model generates the
XMLised data dynamically which can be served for any number of video
requesters without any limitation. This dynamic model has been tested by
fetching video files of varying sizes in a distributed environments.

RMI and Web services bring more flexibility in different sectors,
inspite of which there raises new challenges for the multimedia files
especially video files. The download and play approach of Video on Demand
is implemented in different distributed environments namely JAX-RPC,
SOAP Communication and cloud platforms.

An innovative SOAP communication model with modified
message exchange pattern is proposed for enhancing the security of Video on
Demand system. This model insists the video requester or the video provider
to give the request initially to an authentication service provider for a video
file to be retrieved or to be uploaded. The response of the authentication
service provider is an encrypted messageID based on the credentials of the
client, which is enclosed in a SOAP envelope and this response is forwarded
as SOAP request to the video service provider in order to access the
appropriate service. A MulticastMessageContext with MulticastMessageReceiver
is introduced in the SOAP communication model to support unicast as well as
multicast of video data as per the client requirements.
It is found that the performance measure RTT for remote invocation in case of Azure cloud environment is faster when compared to other distributed environments. It is also found that there is very minimal variation in RTT between the local and remote invocation for Video on Demand services in distributed environments. Statistics from the Motion Picture Association of America shows that optical and video piracy around the globe, not just in the US, is the most significant threat to the future of the movie industry. Thus, in order to reduce the piracy, the progressive download approach has been developed and implemented in both the cloud environments. Based on the performance measure RTT, the Azure environment is found to be faster. Thus, a scalable and secure Video on Demand model in a cost effective manner is being proposed.