CHAPTER 2

DISTRIBUTED MODELS FOR MEDIA ON DEMAND

2.1 INTRODUCTION

With the proliferation of broadband access, the Internet faces the challenge of distributing rich media content to a world wide audience of high-speed users. The challenges for media on demand and streaming media today include high data rates and significant bandwidth required for the uninterrupted delivery of high-quality music and video. Building a scalable and reliable system for on-demand and live streaming in this environment has proved to be very difficult. Media on demand system allows clients to access the media when they want and play it back without interruption after a given start-up latency. In media on demand systems, the outgoing bandwidth from the server is independent of the number of clients and each client behaves independent of each other. Even though most codecs tolerate a certain amount of data loss, a high loss rate can significantly affect playback quality as the server minimizes the aggregate bandwidth required for each increased media stream.

Due to the limited processing resources available on a typical host, multimedia applications are often restricted to simple tasks, covering a restricted number of media streams. The resource requirements for the multimedia application domain are very challenging and will remain so in the
near future, justifying the need for scalability. This limitation on processing resources can often be reduced by parallelizing and distributing an application, utilizing the resources on several hosts. Today in the world of Internet, distributed technologies are available to harness the power of pervasiveness of the resources. However, multimedia applications consist of multiple logical levels such as feature extraction, classification, filtering, managing and streaming. This complexity makes parallelization and distribution a difficult task, as each logical level may require special purpose techniques (Acharya et al 2000, Almedia et al 2001, Cherkasova and Gupta 2002). Therefore a framework is needed to support media process on any given logical level, independent of other levels without much focus on resources, which can be achieved by middleware computing in open distributed environments.

Distributed object computing overcomes the drawbacks with the monolithic architecture and performs computation by a more flexible and cost-effective approach. Distributed object computing allows parts of the media process system to be located in different locations and make use of the resources available at the location. Distributed object technologies are based on the modification of the Remote Procedure Call (RPC) mechanism that has been designed to hide peculiarities of communication and significantly simplify the programming applications. Distributed object middleware is responsible for providing transparent layers that deal with distributed system complexities such as location of objects, heterogeneous hardware/software platforms and numerous object implementation languages.

Distributed object technologies aim at platform, location and language transparencies making it easy to access and use objects on a remote
node as an object on a local node. Handling non-local references is more complex than handling local references but the distributed computing technology removes this burden, freeing to focus on the application. Distributed object computing allows the remote classes to be stored on multiple systems and provides flexible ways to locate and load objects efficiently. This enables remote method calls by passing remote objects as arguments, which improves resource availability and system performance. Distributed models are important for building robust, scalable and interoperable remoting applications.

Multimedia support in distributed object computing technologies has been attracting more attention from researchers during recent years since distributed technologies have become increasingly popular. Berkley’s distributed Video on Demand system is a hierarchical storage system composed of a database, one or more video file servers and archive servers that incur high cost (Rowe et al 1995). Distributed object computing has reduced the development complexity and maintenance costs of large scale non real-time software applications. However, real-time media applications have not fully adopted the concept of distributed computing that is highly effective to reduce the complexity and maintenance costs of media applications (Rowe 2002). The middleware platform provides compliance and compatibility with standards ensuring openness, interoperability, extensibility and general coherency of approach to distribution with realistic use and applicability.

To answer the evolution of multimedia computing, middleware platforms must provide supreme treatment of multimedia information that allows efficient real-time operation and interaction on media that are extensible with regard to types, algorithms and programming. Traditionally, distributed
computing approaches to multimedia have been a behind-the-scene handling approach, with less performance. New media types are developed regularly with new algorithms for processing, interpreting and displaying media data that change continuously. To allow realistic development of multimedia applications in a distributed platform, these media developments are to be easily incorporated irrespective of the particular characteristics of hardware devices or network traffic.

Selecting an appropriate distributed object model for media application is a multi-criteria decision problem, as the development of the media application in a distributed platform should satisfy a number of requirements and the right choice cannot be made without a detailed analysis of distributed models. Developers and users of high-performance multimedia applications often observe performance problems such as unexpectedly low throughput or high latency. The reasons for the poor performance can be manifold and are frequently not obvious. It is often difficult to track down performance problems because of the complex interaction between the objects distributed across the network and the performance issues can occur in several places in the event chain of a distributed system that may be most apparent.

Evaluating the performance of distributed object models is a difficult task and there is no standardized or commonly accepted method for performance assessment of media applications in distributed environment. Multimedia applications when deployed in real-world distributed environment provide inadequate service such as propagation delays influenced by physical distances, intermediate network devices and the scheduling strategies applied to different components of the system which collide with quality requirements.
Hence higher expectations are placed on the performance of distributed models as the growing number of media applications use distributed environment as its infrastructure.

Applications and services for next generation-distributed systems must be capable of providing low latency, quality of service to delay-sensitive applications such as media on demand. Scalability and flexibility are essential to respond rapidly to the changing application requirement and access patterns. Hence when developing an application in a distributed environment it is important to know its limitations and bottlenecks in order to achieve better performance. Identifying the problems due to delay, unreliable service and insecurity in the event chain and taking measures to overcome them can improve the performance of the system. Bottleneck can occur in any of the component along the path through which the data flows, namely the node, the application, the operating systems, and the network components such as switches and routers or due to unwanted malicious activity, which has an impact on the operation of the distributed system. Hence it becomes necessary to analyze the performance of distributed system while deploying a media application.

Distributed object computing uses an acknowledgment-based protocol (Transmission Control Protocol) for data transmission. A shortcoming with the acknowledgement-based protocol is the delay associated during packet losses as it waits for the lost packets to be retransmitted to ensure reliability. This is extremely inefficient and unacceptable in the case of media transmission. Loss of media packets is mostly due to failure of route or due to heavy congestion and hence an alternate route has to be established at the earliest to ensure the availability of the data in a shorter duration and provide a
reliable service. Hence the major performance issues due to network congestion, unreliability and insecurity in transmission are to be alleviated in a distributed environment for media applications.

Due to large size and high bandwidth requirements of digital video, server and network bandwidths are proving to be major limiting factors in the use of video over the Internet. This has led to significant increase in network congestion and latency as perceived by the user. To increase the reliability and efficiency of these distributed applications the idea is to incorporate caching and multicasting. Explosive growth in demand for media applications (Nahrstedt et al 1998, Rejaie et al 1999) justifies the need for caching video at a proxy server close to the client. Similar to the traditional caching of text and image data, storing of video in the proxy enables to reduce the delay without sacrificing the quality. A proxy cache stores the recently accessed video data and serves the clients quickly for future requests without contacting the main server. Prefetching and caching video near clients, reduces start up delays and the possibility of adverse Internet conditions disrupting video playback (Gao et al 1999). Proxy server reduces server load by intercepting a large fraction of server access and increases scalability. Multicasting allows the deployment of multimedia applications on the network while minimizing their demand for bandwidth. Multicasting the media is aimed at optimizing the usage of available bandwidth and preventing multiple copies transmission over the same link.

Mobile agents have the unique ability to transport themselves from one system in a network to another thereby reducing network traffic and providing a means of overcoming network latency with uninterrupted service. Mobile agent has the ability to operate asynchronously and independently of the process that created it and a complete robust distributed environment for media applications is achieved.
As the size of the video database increases then the vulnerability to attack also increases. Hence it becomes necessary to secure the data at storage and transmission, which is achieved through encryption and access control to clients. Many different video encryption schemes have been proposed (Saman Tosun 2001, Cheng and Li 2000, Shin et al 1999, Shi et al 1999, Qiao and Nahrstedt 1997 and Tang 1996). Most of them are joint compression encryption methods, which are specially designed to provide reliable security for MPEG video stream. From the works of Qiao and Nahrstedt (1998), Uehara and Naini (2000) some video encryption schemes are known to be not secure enough from strict cryptographic viewpoint. Hence a secure model for media on demand is proposed with Advanced Encryption Standard.

To meet the challenges faced by the multimedia applications in distributed environment an extensive implementation with performance analysis has been carried out in this thesis for Media on Demand models using RMI, which provides platform independency, RMI-IIOP and CORBA-IDL which provide location independency and .NET Remoting that provides language independency and also a dynamic re-routing model is proposed for media on demand in .NET Remoting environment so as to enhance the reliability and security of media transmission. It is also aimed at developing component model for the reusability of media component, agent model for providing an uninterruptible media service, advanced security model for enhanced protection, prefetch catching and multicast model to reduce the latency and bandwidth requirements.
2.2 REMOTE METHOD INVOCATION MODEL

Java’s Remote Method Invocation is a revolutionary distributed technology that extends the pure Java object model to the network. Modern web applications are commonly built out of components or distributed objects and therefore a special demand is placed on connectivity and interoperability among the objects. Java offers basic communication mechanisms with its support for sockets combining RPC concepts and object paradigm, resulting in a distributed object model namely Remote Method Invocation.

RMI enables objects on one Java Virtual Machine to seamlessly invoke methods on objects in remote virtual machine on a different host. RMI provides a registry that lets programs to obtain references to remote objects and uses Java’s serialization facility to transfer method arguments and return values across network. RMI hides all the details of remote communication such as locating the remote object, transferring the request and returning the result (Cay and Gary Cornell 2002). The client and server of an RMI system are connected by a layered architecture that is transparent to the applications which is shown in Figure 2.1.

![RMI Architecture](image-url)
Three independent layers that constitute the RMI system are
stub/skeleton layer - the interface between the application layer and the rest of
the RMI system, remote reference layer - responsible for carrying out the
semantics of the invocation that sits on top of the transport layer and the
transport layer - responsible for setup and management of the connection and
dispatching the requests to the remote objects within transport layer's address
space. RMI uses HTTP for posting remote method invocations, Java object
serialization for marshalling and unmarshalling the data and Java Remote
Method Protocol (JRMP) for communication.

2.2.1 RMI architecture for Media on Demand

A distributed environment has been established using RMI to deliver
the media demanded by multiple clients from a media server. In this model, a
media provider method located in the remote object can be invoked from the
framework, possibly on different hosts. A client application can make a call to
the method on a remote object once it obtains a reference to the remote object,
either by looking up the remote object in the bootstrap naming service provided
by RMI or by receiving the reference as an argument or by a return value. RMI
uses object serialization to marshall and unmarshall parameters and does not
truncate types. The media clients are interconnected with the media server as
shown in Figure 2.2.

Chronologically the server process should be started first, so that it
can take the initiative to set up a connection line. It then waits to receive a
request from the client. A client can register itself with the remote object
(Server object) just by invoking the registration procedure on the server object,
when it needs a service from it. The remote object obtains the necessary data from the registered clients and responds back to them respectively with the requested media.

In this proposed model, media client calls a method of a remote media object through the stub, which presents the same interfaces as the remote media object itself. In this method an abstract representation of a media file has been created and is converted into bytes, which could be read from the input stream and written to an output stream that can be stored in a sink. The media stub on the client side builds the information block that consists of an identifier of the remote object to be used and a description of the method to be called. The media skeleton always resides on the server side and packages the media data into a block of bytes that can be communicated through the network.
This process of marshalling makes the entire media data into a suitable format for transporting from one virtual machine to another.

When the media stub sends the request information to the media server, the skeleton on the server side receives the requests, unmarshalls the parameters, performs deserialization and invokes the appropriate code on the server. This receiver locates the media object to be accessed and then calls the desired method with those parameters, captures the return value and marshalls the media data as packets onto a marshall stream and sends the media to the stub. The stub unmarshalls the media stream and returns it to the original caller. The runtime object associated with the media application is returned to the client. Then the runtime object executes the media process with a specified environment and controls the media delivery process. The associated RMI data flow model is shown in Figure 2.3.

![Media on Demand - Data Flow Model](image-url)
2.2.2 Self registry service and dynamic stub loading

Bootstrap registry service is provided in RMI to locate remote server objects. Server program registers remote objects with the bootstrap registry service and the clients retrieve stubs to those objects. In this proposed model, media server creates its own registry and maintains the stubs for the remote media objects by itself and hence the media server no longer depends on the bootstrap registry service provided by RMI.

In RMI architecture, clients can communicate with the remote object only when the stub is available with the client. In this model, an external web server is used to load the media stub dynamically on the client side as shown in Figure 2.4.

Figure 2.4 Media on Demand - Dynamic stub loading
The steps involved in dynamic downloading of media stub is as follows:

i. The remote media object’s codebase is specified by the remote media server by using java.rmi.server.codebase property. The media server registers the remote media object, bound to a name with RMI registry.

ii. The media client requests a reference to the named remote media object. The reference to the remote media object’s instance is what the media client will use to make remote method calls on the media object.

iii. The RMI registry returns the stub instance reference to the requested class.

iv. The codebase which the media client uses is the URL that is annotated to the stub instance when the stub class is loaded by the registry.

v. The class definition for the stub is downloaded to the client dynamically.

Now the media client has all the information that it needs to invoke the media provider method in the media server object. The instance of the media stub acts as a proxy to the remote media object that exists on the media server. Any changes in the interface of the media object results in the modification of the stub and it will be made available for all media clients by dynamic class loading through an external server.
2.3 CORBA BASED MEDIA ON DEMAND MODEL

CORBA (Common Object Request Broker Architecture) is based on the Object Management Architecture and the Core Object Model (OMG 1995). CORBA is independent of a programming language that utilizes the strict separation of the interface and the implementation. Interfaces are specified in a proprietary Interface Definition Language (IDL) while implementation is done in any programming language for which a mapping from IDL exists. The architecture of a typical CORBA compliant ORB includes static stubs which provide the static invocation interface that are responsible for marshalling the requests and static skeletons that unmarshall the low-level message representation into typed data. Static stubs and skeletons are generated at compile time by an IDL compiler.

Dynamic invocation interface allows the dynamic request generation, which is useful when the client has no compile time knowledge of the interfaces it is accessing. Dynamic skeleton interface allows to deliver requests to servants that have no knowledge about the IDL interface they implement. The client does not know whether the server is using the static skeleton or the dynamic skeleton interface. Object adapter associates servant with objects, demultiplexes the requests and dispatches the appropriate operation on servant.

ORB Core is responsible for the communication between the client and the objects in the server utilizing the operating system services. For inter ORB communication CORBA standard specifies the General Inter-ORB Protocol (GIOP), which is a language independent wire protocol that defines the common data representation, GIOP message formats and GIOP transport assumptions.
In this CORBA model, media broker allows media clients to invoke operations on distributed media objects without the concern for the location of the media object. The media object is an instance of an IDL interface. The media object is identified by a unique name, which is the object reference used for identification across servers. An object ID is associated with the media object in its servant implementation and is unique within the scope of object adapter. The object adapter associates the media servant with media objects, demultiplexes incoming requests to the servant and collaborates with the IDL skeleton to dispatch the appropriate operation on the media servant. Media servant implements the operations defined in the IDL interface. The operations to be performed are specified in the method in remote media object. ORB delivers the request to the media object when a client invokes an operation on it and returns the response to client as shown in Figure 2.5.

![Figure 2.5 CORBA Implementation of Media on Demand Model](image)

CORBA compliant ORB uses the General Inter-ORB Protocol for the communication between media client and media server. The ORB interface provides standard operations such as object conversion, initializing and shutting of ORB. The IDL stub serves as a bridge between the media client and the ORB and the IDL skeleton serves between the Servant and the ORB. The IDL stub marshalls the media application parameters into a block of bytes that can be communicated through the network. This process of transformation makes the
entire media data into a suitable format for transporting from one machine to another. When the IDL stub sends the information to the media server, the IDL skeleton on the server side unmarshalls it into a meaningful form and gives to the media object and invokes the media method. It then captures the return value and transports the data as packets onto a stream and sends it to the stub. The stub returns the media data to the client and is played.

2.4 MEDIA ON DEMAND MODEL USING RMI – IIOP

In this model, Java Remote Method Invocation technology is run over Internet Inter-ORB Protocol that delivers CORBA distributed computing capabilities to the Java platform for the media on demand application. RMI over IIOP combines the features of Java RMI technology with the best features of CORBA technology. RMI over IIOP is based on open standards and use IIOP as its communication protocol. IIOP eases legacy application development and achieves platform integration by allowing application components written in JAVA or other CORBA supported languages to communicate with components running on the Java platform.

When using RMI over IIOP to produce media on demand applications, no separate IDL or mapping is needed. RMI over IIOP provides flexibility to pass the media object between the media server and the media client by serializing the media object. With RMI over IIOP, remote media interfaces and implementations are written in JAVA and complied with iiop extension, which integrates the media stub and media skeleton with IIOP. Using rmic with the iiop option generates media stub and media tie classes. Media stub is the local proxy for the remote media object that is used by media clients to send calls to a media server. Tie classes used on the server side process the
incoming calls and dispatch the calls to the proper media object implementation. They receive the results from media object and return it to the media client.

Media interfaces can be implemented in any language that is supported by Object Management Group (OMG) mapping and ORB for that language. RMI over Internet Inter-ORB protocol allows RMI media client to reference and lookup the media object using CORBA CosNaming service, thus achieving greater interoperability between architectures. Hence RMI clients are able to access CORBA servers and CORBA clients are able to access RMI servers transparently as shown in Figure 2.6. Based on the usability of RMI or CORBA, either platform independency or language and location independency is achieved.

![Diagram of interaction between RMI client and CORBA server](image)

**Figure 2.6 Interaction between RMI client and CORBA server**
In Figure 2.6, the top section represents the RMI (JRMP) model, middle section represents the RMI-IIOP model and bottom section represents the CORBA model. An arrow represents a situation in which a media client calls a media server. The diagonal arrows that cross the border between the JRMP model and the IIOP model imply that an RMI (JRMP) media client can call an RMI-IIOP media server and vice versa. Thus a media application created using RMI can be exported as either JRMP or IIOP without rewriting or recompiling its Java source code, but simply by changing the Java system properties while running the media application. The same flexibility applies to RMI-IIOP media client also. Using rmic with idl option generates OMG IDL for the media classes specified and any media class can be referenced. IDL provides a language independent capability for specifying a media object that can be written in and invoked from any language providing CORBA binding.

2.5 .NET REMOTING MODEL FOR MEDIA ON DEMAND

.NET Remoting provides a powerful and easy way for cross-application communication providing flexibility and control for audio and video communication. The .NET Remoting framework provides a number of services for activation, formatters, lifetime support and communication channels for application development. A distributed media on demand model has been developed using .NET Remoting framework to deliver the media from a media server when demanded by multiple clients.

Formatters are used for encoding and decoding the messages, which can be a binary encoding where performance is critical or XML encoding where interoperability with other remoting framework is essential. All XML encoding
uses the Simple Object Access Protocol (SOAP) in transporting messages and Hyper Text Transfer Protocol/Transmission Control Protocol (HTTP/TCP) for communication between application domains. Channels implement a communication medium between a media client and a remote media object across application domain boundaries to pass messages. .NET Remoting uses object serialization to marshall and unmarshall parameters. Once the remote media object has been created and a channel has been instantiated the formatter and the proxy will be supplied automatically.

A media client makes a call to the method on a remote media object, which is marshalled by value. Since the media object runs inside a process on a different system that is different from the client process, the client cannot call it directly. To call the media object, the media client uses a proxy, which appears like the real media object with the same methods. When the methods of the proxy are called, messages are created that are serialized using formatters and sent into a media client channel. The media client channel communicates with the server part of the channel to transfer the messages across the network. The media server channel uses formatters to deserialize the message, so that the requests are dispatched to the remote media object. The results from the media object are returned in the same way to the media client. The .NET Remoting architecture for media on demand is shown in Figure 2.7.

Media application in .NET platform exists in its own unique domain and remoting enables applications running in one domain to communicate with applications in other domain and provides a remarkable amount of elasticity for applications. The standard .Net Remoting allows to consume a remoting object using one standard channel such as tcp or http to transport messages between
Figure 2.7 .NET Remoting Architecture for Media on Demand

applications domains, processes or computers across boundaries. It provides a set of common services that are used to execute the application in the form of intermediate code that is independent of the underlying architecture. They operate in a Common Language Runtime (CLR) that manages resources and monitors application execution.

2.5.1 Remote Media Objects

A media object residing outside the application domain of the caller is considered to be a remote object, even if the objects are executed on the same machine. .NET remoting framework provides the necessary infrastructure that hides the complexities of calling methods on remote media objects and returning the results. All media objects that cross the application domain boundary are passed by value and marked with the serializable attribute or
implemented with the I Serializable interface. When the media object is passed as a parameter, the framework serializes the object and transports it to the destination application domain, where the media object is reconstructed.

When a client activates a remote media object, it receives a proxy to the remote object. All operations on this proxy are suitably directed to the remoting infrastructure to intercept and forward the calls appropriately to the media object. The call is then examined to determine if it is a valid method of the remote object and the object lies in that application domain. As the proxy and remote media object are in different application domains, the parameters of the call on the stack are converted into messages and transported to the remote application domain. In the remote domain the messages are turned back into a stack frame and the media method is invoked. The same procedure is used for returning results from the remote method to the media client.

A remote media object is registered in a media application domain on a remote machine is marshalled to produce an Object Reference (ObjRef). The ObjRef contains the information such as details of channels that have been registered, object URI required to locate and access the media object from anywhere on the network. The remoting framework uses the object URI to retrieve the remote object when it receives a request for that object. The remote media object is activated by the activation method when the client calls a method on the media object. When a media client calls one of the activation methods, an activation proxy is created on the client and a remote call is initiated to a remote activator on the server using the Uniform Resource Locator (URL) with object URI as the endpoint. The remote activator activates the object, and an ObjRef is streamed to the client, where it is unmarshalled to
produce a proxy that is returned to the client. During unmarshalling, the ObjRef is parsed to extract the method information of the remote object, handle all function calls on a remote object and automatically handle the return parameter.

2.5.2 Channels

Channels are used to transport messages to and from remote media objects. When a media client calls a method on a remote media object, the parameters as well as other details related to the call are transported through the channel to the remote media object and any result for the call is returned to the media client in the same way. A media client can select any of the channels registered on the media server to communicate with the remote media object and the .NET remoting framework ensures that the media object is connected to the right channel when a client attempts to connect to it.

The HTTP channel transports messages to and from remote objects using SOAP protocol. TCP channel uses a binary formatter to serialize all messages to a binary stream and transport the stream to the target URI using TCP protocol. TCP channel is chosen to transport messages holding information about the remote media object, name of the method and all arguments of the method in binary format. Additional mechanisms are needed in .NET remoting infrastructure to chain and re-route channels between the client and the remote object.

Channels send each message along sinks prior to sending or after receiving message. When a media client calls a method on a proxy, the call is intercepted by the remoting framework and changed into a message that is
forwarded to the proxy. Messages hold information about the remote object, name of the method and all the arguments of the method. The proxy forwards the message to the sink chain for processing. Hence a remote method call is thought of as a message from the media client to the server and back again. The message passes through a chain of message sinks on each side of the transport channel as it crosses boundaries.

This chain contains sinks such as formatter sink, transport sink and custom sink that are required for basic channel functionality and to perform special tasks with a message. The first sink in the chain is a formatter sink that encodes and decodes the messages, generates the necessary headers and serializes the message into the message stream before the channel transports them. The message is then passed through a series of sinks until it reaches the transport sink at the end of the chain.

Custom channel sinks are inserted into the chain of sinks between the formatter sink and the transport sink. Inserting a custom channel sink in the client or server channel enables to process the Message in which a call represented as a message is converted into a stream and sent over the wire. Custom sinks read or write data to the stream depending on the call (outgoing or incoming) and add the desired additional information to the headers. Custom sinks intercept the messages and performs additional actions such as creating a lock, writing an event, performing security checks and so on and then passes the message to the next channel sink.

When the message call is forwarded to the transport sink at the end of the chain, the transport sink writes the headers to the stream and forwards the
stream to the transport sink on the server using the transport protocol dictated by the channel. The transport sink on the server forwards the byte stream through the custom chain to the formatter sink, where the message is deserialized and dispatched to the remote media object that is shown in Figure 2.8.

Figure 2.8 Channels between client and server

Sinks are responsible for transporting messages between the media client and the media server. A remote media object does not own a channel and the media server hosts the media objects that are to be exposed with the remoting framework and registers the channels required by the media object. When a channel is registered, it automatically starts listening for the request
from media clients at the specified port. When a request is received from a client, they are dispatched to the remote media object and the results from the media object are returned to the media client.

2.6 DYNAMIC RE-ROUTING MODEL FOR MEDIA ON DEMAND

A dynamic re-routing model is proposed to overcome the problems due to channel failure or interruptions due to unreliable sources or due to malicious attacks, when a client invokes a media object available in the media server. TCP and HTTP channels are used to transport messages to and from remote media objects in .NET framework. When a media client calls a method on a remote media object, the parameters as well as other details related to the call are bundled together in a packet and transported through the established link to the remote media object and the results for the call are returned to the client in the same way. A media client can select any of the routes on the media server to communicate with the remote media object, which must be registered before the media objects can use it. The remoting framework ensures that the remote media object is connected to the correct route when a media client attempts to connect to it. The .NET remoting infrastructure does not have a direct mechanism to chain and re-route channels between the client and the media object, which is proposed achieving a more reliable media transmission.

Customized routing can be achieved for the well-known remote media object by knowing about the metadata describing the interface contract and the URL address of the remote media object. The interface contract is an abstract definition between the consumer and remote media object located in
the shareable assembly installed in the General Assembly Cache (GAC), which allows building a distributed model easily. The URL address of the media object represents the physical description of the peer-to-peer connectivity, which is retrieved from the custom configuration file programmatically. The URL address consists of the information related to the location of the remote media object and remoting channel used (HTTP or TCP).

Along with URL address, URI that is the remoting end-point address referencing the remote media object published by the media server is used to identify the correct remote media object. The connectivity to the well-known remote media object described by its URL is mapped to the logical URL address and administrated in the configuration file, which is changed during runtime. The logical URL address allows virtualizing a distributed model based on the remoting interface contracts. The URL address is stored in the UrlknowledgeBase and based on the media client request, the format is checked and mapped to the physical URL address required by the remoting infrastructure through the routers as shown in Figure 2.9.

![Figure 2.9 Customized Routing - Block Diagram](image-url)
2.6.1 Router Design

Router is designed based on mapping the physical URL address to unique logical name and re-directing the message to the first custom sink of the chained sinks in the channel. Chaining the sinks in the channel is allowed in .NET Remoting infrastructure by which plugging in a custom sink in the proper position in a stack of the message sinks allows monitoring the route through which the message and data flows. Router is created and inserted into the chain to achieve an increased reliability of the system. Router performs security checks on the route through which the data flows and achieves dynamic re-routing if threats arise. The connectivity to the well-known remote media object is examined by processing the message at the router, which is shown in Figure 2.10.

Figure 2.10 Message Processing at Router
The incoming message is processed and passed by the routers. The router checks for the delimiter in the URL address string and if it does not exist, the standard message flow is invoked and result is returned back to the caller. If the router delimiter is present, then the router logic checks for the reliability of the route through which the message flows and replaces the logical URL address from the local UrlKnowledgeBase if necessary. Router then searches for the next channel to pass the Message to the first message sink of the next channel. The router also returns the result from the server back to the caller.

2.6.1.1 Client Router

Client router is designed for mapping the physical URL address to a unique logical name, which represents the suitable logical URL address on the client side. Client router makes this mapping based on the knowledge of the URL addresses in the UrlKnowledgeBase. The UrlKB located in the client sink provider is configured during the registration service from the configuration file. The remote media object (published as video - custom name) is fully transparent on the client side and the media client creates its proxy based on the unique logical URL address in the TCP channel and communicates with the remote media object. The configuration file contents are updated during the runtime, which allows dynamic reconfiguration of a distributed model. The configuration snippet showing the configuration of the client provider in the TCP channel with the urlKB is shown in Figure 2.11.
2.6.1.2 Server Router

Server router is designed in such a way that it is responsible for transporting messages across channels in the network and for dynamic re-routing. The Server router processes the client's outgoing message so that it can control the flow of the message based on the URL address. The message arriving at the server router is intercepted, reliability and security checks are carried out and dispatched to the next channel as shown in Figure 2.12.
The router of the Channel X-1 checks for the router delimiter ('$') in the message string and indicates to forward the message to the next channel (Channel X) in the route after inspecting the connectivity and security issues. The next channel in the URI string is described by its unique logical URL address, so that the router will replace it by the physical address from its knowledgeBbase (UrlKB). After that, the message can be re-routed to the proper channel. The chaining of routes can be used in any combination of the standard channels.

Based on the proposed concept, the connectivity to the remote media object is transparent to the client regardless of the number of routes that have been chained. Thus the connectivity is described by the unique logical name and the routers in each channel will take care of proper routing to the remote media object. The client and server channel providers are specified as parameters in the configuration file when constructing channels and the remoting framework performs the chaining. Similar to the client provider, the server provider also contains the UrlKB to obtain the physical URL address for the corresponding logical address.

Channel sink providers are used to provide the routers, one for the client and the other for the server. Both of them have the same router-id, which is used to reference them in the configuration files. When multiple channel sink providers are provided in a configuration file, the framework will chain them together in the order in which they are in the configuration file which is shown in Figure 2.13.
<configuration>
<system.runtime.remoting>
<application>
<service>
<wellknown mode="SingleCall"
type="TestObject.ObjectA,PTestObject,Version=1.0.941.28888,
Culture=neutral,PublicKeyToken=144391ea9ae609ae"
objectUri="video" />
</service>
<channels>
<channel ref="tcp" port="9090">
<serverProviders>
<formatter ref="binary" />
<provider ref="router" name="TcpRouterS9090"
</serverProviders>
</channel>
</channels>
</application>
</system.runtime.remoting>

Figure 2.13 Server Router Configuration

When the remoting message flows, RouterClientSinkProvider has been deployed for mapping logical URL address to the physical URL and the RouterServerSinkProvider is used for routing based on chosen route or dynamic re-routing when there are troubles with the route.

The routers are ready for use in the .Net Remoting infrastructure, once they are installed in the General Assembly Cache. The routers of each channel will take care of forwarding the request securely to the media object and the results will be returned to the client thus increasing the reliability of the model. The connectivity to the remote media object is transparent to the media client regardless of the number of channels that have been connected and as the routers in the channels are not tightly coupled, they can be used separately based on the requirements.
2.6.2 Dynamic Updation of Router

The knowledge base of the router (UrlKB) is the key factor in re-routing, which can be updated dynamically during the runtime. The CallContext object is introduced, which travels with the remoting message between the media client and remote media object, which updates the route. RouterLogicalCallContext assembly associated with CallContext object is used for dynamic updation of UrlKB and hence the route is updated as shown in Figure 2.14.

```csharp
namespace cc.MessageRouter
{
    (Serializable)
    public class RouterLogicalCallContext : ILogicalThreadAffinative
    {
        string strUrlKB;
        public string UrlKB
        {
            get
            {
                return strUrlKB;
            }
            set
            {
                strUrlKB = value;
            }
        }
    }
}

router name = TcpRouter99092;
RouterLogicalCallContext urlkb= new RouterLogicalCallContext();
urlkb.UrlKB = "curl=tcp://localhost:1234/myObjectUri, curll=";
CallContext.SetData(routerName, urlkb);
```

Figure 2.14 Dynamic Updation of Router
The UrlknowledgeBase is updated by the above way and hence the logical URL name is changed from one to other which can be used for dynamic re-routing which provides a reliable and safe communication between the media client and the media server systems thereby improving the performance of the system.

2.7 PERFORMANCE ANALYSIS OF MEDIA ON DEMAND MODELS

Selecting an appropriate distributed object model for the media on demand application is a critical problem. The choice of the distributed object model should be made at an early stage as the transition from one model to the other is painful and has bad impact on the costs and the schedule of the project. The right choice cannot be made without a detailed analysis of the models. Traditionally performance issues have a high impact on the decision. The performance analysis of distributed object models is necessary as it has many internal overheads due to marshalling and unmarshalling, demultiplexing and dispatching, data copying and additional remote invocations for resolving object references.

Evaluating the performance of media on demand models in distributed environment is a difficult task and there is no standardized or commonly accepted method for performance assessment of media applications in distributed environment. The performance issues of the media on demand models developed using Java RMI, RMI-IIOP, IDL and .NET framework are analyzed and the results are obtained for media files of various sizes in a single and multiple clients scenario.
A subset of the performance evaluation method described by Matjaz et al (1999) has been used to examine the performance of the media on demand models. The author suggests a performance evaluation method which is independent of the underlying distributed object model, whose results are comparable and do not place much overhead on the model. The performance criteria that are considered in this thesis are Round Trip Time, Data Throughput and Scalability (Performance degradation). The round trip time measures the time needed from the point when the client initiates a media method invocation to the point when the client receives the result. The Data throughput measures the efficiency of data transfer from the media server to the media client for the method invocations. Scalability evaluates the performance degradation when multiple clients simultaneously interact with a single server object. Performance degradation is computed as the quotient between the RTT measured for the multiple clients accessing simultaneously to the RTT measured for a single client.

2.7.1 Client – Server Model (Single Machine)

The round trip time for the media method invocation and the data throughput have been measured for the media files of various sizes by running both the media client and media server in a single machine. The result states that in all the cases RMI shows the best performance. RMI-IIOP, CORBA-IDL and .NET are comparatively slower than RMI. RMI-IIOP is slower by 32 % and IDL is slower by 53% and .NET is slower by 40 %. The results of Data Size vs. RTT and Data Size vs. Data Throughput are shown as graphs in Figures 2.15 and 2.16 respectively and with numeric values in Tables 2.1 and 2.2 respectively.
Table 2.1 Client - Server Model (Single Machine - RTT)

<table>
<thead>
<tr>
<th>Data size</th>
<th>Round Trip Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RMI</td>
</tr>
<tr>
<td>100 KB</td>
<td>192</td>
</tr>
<tr>
<td>1 MB</td>
<td>424</td>
</tr>
<tr>
<td>2 MB</td>
<td>806</td>
</tr>
<tr>
<td>3.5 MB</td>
<td>1022</td>
</tr>
<tr>
<td>4.5 MB</td>
<td>1195</td>
</tr>
<tr>
<td>10 MB</td>
<td>2675</td>
</tr>
<tr>
<td>20 MB</td>
<td>5350</td>
</tr>
</tbody>
</table>

Figure 2.15 Data Size vs. RTT in single machine
Table 2.2 Client – Server Model (Single Machine - Data Throughput)

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Data Throughput in Mbps</th>
<th>RMI</th>
<th>RMI - IIOP</th>
<th>CORBA</th>
<th>.NET Remoting</th>
<th>.NET Re-routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>0.4901</td>
<td>0.3759</td>
<td>0.2481</td>
<td>0.4166</td>
<td>0.2309</td>
<td></td>
</tr>
<tr>
<td>1 MB</td>
<td>2.1008</td>
<td>1.5082</td>
<td>1.3477</td>
<td>1.3440</td>
<td>0.8326</td>
<td></td>
</tr>
<tr>
<td>2 MB</td>
<td>2.5477</td>
<td>2.2222</td>
<td>1.7746</td>
<td>1.5267</td>
<td>0.9638</td>
<td></td>
</tr>
<tr>
<td>3.5 MB</td>
<td>3.4584</td>
<td>2.6080</td>
<td>2.3939</td>
<td>2.5362</td>
<td>1.5865</td>
<td></td>
</tr>
<tr>
<td>4.5 MB</td>
<td>3.6824</td>
<td>2.7256</td>
<td>2.3759</td>
<td>2.5041</td>
<td>1.5822</td>
<td></td>
</tr>
<tr>
<td>10 MB</td>
<td>3.9888</td>
<td>2.8885</td>
<td>2.4931</td>
<td>2.6420</td>
<td>1.5822</td>
<td></td>
</tr>
<tr>
<td>20 MB</td>
<td>4.1254</td>
<td>3.0021</td>
<td>2.5773</td>
<td>2.9468</td>
<td>1.5820</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.16 Data Size vs. Data Throughput in single machine
2.7.2 Client – Server Model

The round trip time for the media method invocation has been measured by running the client and media server in separate machines for different media files of various sizes. The data throughput is measured under the same condition. As per the results obtained, the models based on RMI-IIOP, IDL and .NET remoting are comparatively slower than RMI based media on demand model. RMI-IIOP is slower by 35 %, IDL is slower by 57% and .NET is slower by 44%. The results of Data Size vs. RTT and the Data Size vs. Data Throughput are shown as graphs in Figures 2.17 and 2.18 respectively and with numeric values in Tables 2.3 and 2.4 respectively.

Table 2.3 Client – Server Model (Different Machine-RTT)

<table>
<thead>
<tr>
<th>Data Size</th>
<th>RMI</th>
<th>RMI - IIOP</th>
<th>CORBA</th>
<th>. NET Remoting</th>
<th>. NET Re-routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>220</td>
<td>297</td>
<td>410</td>
<td>330</td>
<td>359</td>
</tr>
<tr>
<td>1 MB</td>
<td>652</td>
<td>880</td>
<td>1043</td>
<td>932</td>
<td>1539</td>
</tr>
<tr>
<td>2 MB</td>
<td>1115</td>
<td>1520</td>
<td>1751</td>
<td>1583</td>
<td>2843</td>
</tr>
<tr>
<td>3.5 MB</td>
<td>1267</td>
<td>1734</td>
<td>2027</td>
<td>1811</td>
<td>4477</td>
</tr>
<tr>
<td>4.5 MB</td>
<td>1437</td>
<td>1969</td>
<td>2270</td>
<td>2084</td>
<td>4670</td>
</tr>
<tr>
<td>10 MB</td>
<td>3193</td>
<td>4312</td>
<td>5045</td>
<td>4534</td>
<td>10370</td>
</tr>
<tr>
<td>20 MB</td>
<td>6384</td>
<td>8634</td>
<td>10023</td>
<td>9065</td>
<td>20770</td>
</tr>
</tbody>
</table>
Figure 2.17  Data Size vs. RTT - client-server (Different Machine)

Table 2.4  Client – Server Model (Different Machine - Data Throughput)

<table>
<thead>
<tr>
<th>Data Size</th>
<th>RMI</th>
<th>RMI – IIOP</th>
<th>CORBA</th>
<th>.NET Remoting</th>
<th>.NET Re-routing</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 KB</td>
<td>0.4545</td>
<td>0.3367</td>
<td>0.2439</td>
<td>0.3030</td>
<td>0.2785</td>
</tr>
<tr>
<td>1 MB</td>
<td>1.5337</td>
<td>1.1363</td>
<td>0.9587</td>
<td>1.0729</td>
<td>0.6497</td>
</tr>
<tr>
<td>2 MB</td>
<td>1.7937</td>
<td>1.3157</td>
<td>1.1422</td>
<td>1.2634</td>
<td>0.7034</td>
</tr>
<tr>
<td>3.5 MB</td>
<td>2.7624</td>
<td>2.0184</td>
<td>1.7266</td>
<td>1.9326</td>
<td>0.7817</td>
</tr>
<tr>
<td>4.5 MB</td>
<td>3.1315</td>
<td>2.2854</td>
<td>1.9823</td>
<td>2.1593</td>
<td>0.9635</td>
</tr>
<tr>
<td>10 MB</td>
<td>3.1318</td>
<td>2.3191</td>
<td>1.9821</td>
<td>2.2055</td>
<td>0.9643</td>
</tr>
<tr>
<td>20 MB</td>
<td>3.1328</td>
<td>2.3164</td>
<td>1.9954</td>
<td>2.2062</td>
<td>0.9629</td>
</tr>
</tbody>
</table>
2.7.3 Multiple Clients Scenario

The round trip time for media method invocation, data throughput and scalability have been measured when multiple clients access the server simultaneously for media. The RTT is measured for all the clients that invoked the method and the time for looking up the object reference is also calculated. The time taken for looking up the media object reference is considered to be transparent of the model and it varies with the model. The look up time for .NET is less when compared to look up time associated with RMI. The results show that RMI preserved its advantages when compared to RMI-IIOP and IDL implementations. The performance degradation reveals that as the media size and number of clients increase, the degradation factor also increases. The performance degradation of various implementations is shown in Table 2.5.
Table 2.5 Performance Degradation of Media on Demand Models

<table>
<thead>
<tr>
<th>Implementation Model</th>
<th>Performance Degradation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>45%</td>
</tr>
<tr>
<td>RMI-IIOP</td>
<td>60%</td>
</tr>
<tr>
<td>CORBA</td>
<td>79%</td>
</tr>
<tr>
<td>.NET Remoting</td>
<td>72%</td>
</tr>
<tr>
<td>.NET Re-routing</td>
<td>80%</td>
</tr>
</tbody>
</table>

From the readings it is found that the time taken to invoke the media method and return the results, increases linearly as the number of clients increases in all the cases. The time taken by the clients increases steadily during the initial stages and increases slowly as the data size and the number of clients increase. When observing the performance degradation the three models of JAVA framework are comparable with slightly higher degradation for CORBA. In .NET framework the results of both the models are comparable.

The round trip time for the media method invocation, data throughput and scalability have been measured for a single server and simultaneous multiple clients scenario using RMI. The round trip time is measured for all the simultaneous clients and the data throughput is calculated. The results of Number of clients vs. RTT and the Number of clients vs. Data Throughput are shown as graphs in Figures 2.19 and 2.20 respectively and with numeric values in Tables 2.6 and 2.7.
Table 2.6  RMI - Multiple Clients (RTT)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>RTT in milliseconds for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>565</td>
</tr>
<tr>
<td>2</td>
<td>944</td>
</tr>
<tr>
<td>3</td>
<td>1201</td>
</tr>
<tr>
<td>4</td>
<td>1280</td>
</tr>
<tr>
<td>5</td>
<td>1364</td>
</tr>
</tbody>
</table>

Figure 2.19  RMI - Number of Clients vs. RTT
The performance parameters have been measured for a single server and simultaneous multiple clients scenario using RMI-IIOP implementation. The round trip time is measured for all the simultaneous clients and the data throughput is calculated. The performance degradation of this model is 60%.
The performance degradation is 20% more compared to RMI. The results of Number of clients vs. RTT and the Number of clients vs. Data Throughput are shown as graphs in Figures 2.21 and 2.22 respectively and with numeric values in Tables 2.8 and 2.9.

Table 2.8  RMI - IIOP - Multiple Clients (RTT)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>RTT in milliseconds for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>1102</td>
</tr>
<tr>
<td>2</td>
<td>1120</td>
</tr>
<tr>
<td>3</td>
<td>1302</td>
</tr>
<tr>
<td>4</td>
<td>1380</td>
</tr>
<tr>
<td>5</td>
<td>1424</td>
</tr>
</tbody>
</table>

Figure 2.21  RMI-IIOP Number of Clients vs. RTT
### Table 2.9 RMI-IIOP - Multiple Clients (Data Throughput)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Data Throughput in Mbps for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>0.9074</td>
</tr>
<tr>
<td>2</td>
<td>0.8928</td>
</tr>
<tr>
<td>3</td>
<td>0.7680</td>
</tr>
<tr>
<td>4</td>
<td>0.7246</td>
</tr>
<tr>
<td>5</td>
<td>0.7022</td>
</tr>
</tbody>
</table>

### Figure 2.22 RMI-IIOP Number of clients vs. Data Throughput
The performance degradation of the CORBA implementation of media on demand model is 79% and it is 40% more than RMI. The results of Number of clients vs. RTT and the Number of clients vs. Data Throughput are shown as graphs in Figures 2.23 and 2.24 respectively with numeric values in Tables 2.10 and 2.11.

**Table 2.10 CORBA – Multiple Clients (RTT)**

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>RTT in milliseconds for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>782</td>
</tr>
<tr>
<td>2</td>
<td>804</td>
</tr>
<tr>
<td>3</td>
<td>871</td>
</tr>
<tr>
<td>4</td>
<td>893</td>
</tr>
<tr>
<td>5</td>
<td>916</td>
</tr>
</tbody>
</table>

**Figure 2.23 CORBA - Number of clients vs. RTT**
Table 2.11 CORBA - Multiple Clients (Data Throughput)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Data Throughput in Mbps for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>1.2787</td>
</tr>
<tr>
<td>2</td>
<td>1.2437</td>
</tr>
<tr>
<td>3</td>
<td>1.1481</td>
</tr>
<tr>
<td>4</td>
<td>1.1198</td>
</tr>
<tr>
<td>5</td>
<td>1.0917</td>
</tr>
</tbody>
</table>

Figure 2.24 CORBA - Number of clients Vs. Data Throughput

The performance degradation of the Media on Demand model based on .NET remoting is 72% and it is 30% more than RMI. The results of Number of clients vs. RTT and the Number of clients vs. Data Throughput with respect to .NET remoting model are shown as graphs in Figures 2.25 and 2.26 respectively and with numeric values in Tables 2.12 and 2.13.
Table 2.12 .NET Remoting - Multiple Clients (RTT)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>RTT in milliseconds for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>471</td>
</tr>
<tr>
<td>2</td>
<td>520</td>
</tr>
<tr>
<td>3</td>
<td>563</td>
</tr>
<tr>
<td>4</td>
<td>642</td>
</tr>
<tr>
<td>5</td>
<td>669</td>
</tr>
</tbody>
</table>

Figure 2.25 .NET Remoting - Number of Clients vs. RTT
In case of .NET re-routing model the performance degradation is 80% and it is 40% more than RMI implementation. The results of Number of clients vs. RTT and the Number of clients vs. Data Throughput are shown as graphs in Figures 2.27 and 2.28 respectively and with numeric values in Tables 2.14 and 2.15.
Table 2.14 .NET Re-routing - Multiple Clients (RTT)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>RTT in milliseconds for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
</tr>
<tr>
<td>2</td>
<td>987</td>
</tr>
<tr>
<td>3</td>
<td>1843</td>
</tr>
<tr>
<td>4</td>
<td>2318</td>
</tr>
<tr>
<td>5</td>
<td>3007</td>
</tr>
</tbody>
</table>

Figure 2.27 .NET Re-routing - Number of Clients vs. RTT
Table 2.15 .NET Re-routing - Multiple Clients (Data Throughput)

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Data Throughput in Mbps for various data size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 MB</td>
</tr>
<tr>
<td>1</td>
<td>1.1627</td>
</tr>
<tr>
<td>2</td>
<td>1.0131</td>
</tr>
<tr>
<td>3</td>
<td>0.5425</td>
</tr>
<tr>
<td>4</td>
<td>0.4314</td>
</tr>
<tr>
<td>5</td>
<td>0.3325</td>
</tr>
</tbody>
</table>

Figure 2.28 .NET Re-routing - Number of Clients vs. Data Throughput
2.7.4 Performance and Reliability Analysis of Media on Demand Models

The performance evaluation carried out for various proposed models shows that the performance of the media on demand application implemented using RMI-IIOP and IDL models is less when compared to that of RMI model. The performance of .NET dynamic re-routing is similar to the performance of RMI. The object reference lookup time (overhead) associated with the media method invocation increases as the number of clients' increases in case of RMI-IIOP and IDL. The lookup time associated with .NET framework does not show much difference in both .NET Remoting and .NET Re-routing. As the time taken for looking up the media object reference is less in the case of .NET Remoting, a dynamic re-routing model is implemented in .NET framework that increases the reliability of the media on demand system and achieves a better performance.

To analyze the reliability of these models, the media on demand system is modeled with data size ($X$) and the number of clients ($Y$) as the input parameters and $R(t)$ is the reliability of the system during the time interval $[0, t]$. Let $\eta(t)$ represent the data throughput of the model and $\omega(t)$ represent the RTT of the model that are represented as a function of $X$ and $Y$. The data throughout and RTT of the proposed media on models are defined by the equations

\[
\eta(t) = X\lambda(t) + [1 - \lambda(t)]Y
\]

(2.1)

\[
\omega(t) = X\mu(t) + [1 - \mu(t)]Y
\]

(2.2)
where $\lambda(t)$ and $\mu(t)$ are the weight factors to balance the tradeoff between the effects of RTT and data throughput and $0 < \lambda(t), \mu(t) < 1$.

Reliability $R(t)$ is a function of $\eta(t)$ and $\omega(t)$. As the data throughput and RTT vary linearly with the media size and the number of clients, these are represented by linear equations and the equations are normalized with respect to the obtained values. The equations for data throughput calculation with media size and number of clients as input parameters are given as follows:

\begin{align*}
\eta(t) &= A + BX \\
\sum \eta(t) &= sA + B \sum X \\
\sum \eta(t)X &= sA \sum X + B \sum X^2
\end{align*}  \hspace{1cm} (2.3, 2.4, 2.5)

where

$s$ is the number of pairs of observations for normalizing the equations that are generated for the obtained data throughput and RTT as in Table 2.6 and 2.7 and

$A$ and $B$ are the unknown constants to be determined as the best estimated values by the principles of least squares

Solving the above-normalized equations by substituting the value for $X$ and for five number of clients will give

$$\eta(t) = 2.06 + 0.009(X - 5)\lambda(t)$$  \hspace{1cm} (2.6)
Similarly the equation for RTT with media size and number of clients as input parameters are given as follows:

\[ \omega(t) = G + HX \quad (2.7) \]
\[ \sum \omega(t) = sG + H \sum X \quad (2.8) \]
\[ \sum \omega(t)X = sG \sum X + H \sum X^2 \quad (2.9) \]

where

\( G \) and \( H \) are the unknown constants to be determined as the best estimated values by the principles of least squares.

Solving the normalized equations by substituting the value for \( X \) data size and for five clients will be

\[ \omega(t) = 4.59 + 0.496(X - 5)\mu(t) \quad (2.10) \]

and \( R(t) = F(\eta(t), \omega(t)) \)

(2.11)

As \( R(t) \) is a function of data throughput and RTT, both the quantities are multiplied together to identify the impact of the input parameters. The result of the product of the two functions is obtained here as

\[ R(t) = 10 + 0.6195\lambda(t) + 14.399\mu(t) + 0.94365\lambda(t)\mu(t) \quad (2.12) \]

Thus the reliability calculated from the above equation with the data obtained for various models shows that the dynamic re-routing model achieves around 85 percent better performance than RMI.
2.8 COMPONENT MODEL FOR MEDIA ON DEMAND

A distributed component model has been constructed for playing media files that are demanded. This model is based on client server architecture without installing isolated server, which could incur high cost (Craig Federighi and Rowe 1995). In the proposed model, the client requests media server for the media clip and the server transfers the request to the database server. The web server reads the data from the database server and transmits the media data to the client. An entity Bean model has been deployed where the client can access the remote media data in the Enterprise Java Bean (EJB) server through Java Naming and Directory Interface (JNDI). The role of the server is to receive the client's request and invoke the entity bean managed persistent media bean. The media bean connects to the database server where the media is stored and reads the media data into the buffer and transmits the media to the client. Figure 2.29 shows the architecture for implementation of enterprise media bean.

![Component Model for Media on Demand](image)

Figure 2.29 Component Model for Media on Demand
The media bean exists within the EJB container, which is inside the EJB server. The client communicates to the media bean through its home interface and remote interface, which are provided by the container. The EJB server provides services to the container, which in turn provides services to the bean and the bean responds to the client.

2.8.1 Media EJB Server

The media server provides an organized framework or execution environment in which EJB container can run. It makes available system services for multiprocessing, load balancing and device access for EJB containers. The J2EE (Java 2 Enterprise Edition) platform enables a multi-tier distributed application model and flexible transaction control. The clients can access the EJB server through the JNDI naming service. Based on the client’s requirement the server communicates with the remote object and fetches the media data and presents it to that specific client. The process is simultaneously done for every registered client by generating a separate thread of control.

2.8.2 Media EJB Container

An EJB container is an environment in which the media Bean executes. It serves as a buffer between an EJB and the outside world. The media EJB container provides services for transaction, management of multiple instances, persistence and security to the component it contains. Since the EJB container handles all of these functions, development of media components is made easy. Media EJB container contains the media bean Home and Remote interfaces.
2.8.3 Media Component

The media bean executes within a Media EJB container, which in turn executes within an EJB server. A media bean is the component, which executes the media method when the media is demanded. In this proposed architecture the media bean that is an entity bean accesses the media data from the oracle database. The records in the database are read and interpreted as objects and the container performs the transformations automatically. EJB containers provide access to the existing persistent media data. Entity Bean can support multiple clients and it can offer better scalability and reusability for media on demand system.

2.8.4 Media EJB Home and Remote Interface

Media home interface serves as the initial point of contact for a client. EJB home interface is a contact between media EJB component class and its container, which defines the construction, destruction and lookup like EJB instances. A media EJB home interface defines the base level functionality through the methods defined in it. The entity bean’s remote interface is the interface to the bean, which the clients invoke. It lists the business methods available in the bean. The EJB object is the client’s view of the enterprise bean and implements the remote interface. While the media bean defines the remote interface, the container generates the implementation code for the corresponding EJB object.
2.8.5 Java Naming and Directory Interface Service

Java Naming and Directory Interface adds value to media bean deployment by providing standard interface to the media clients. Naming service in JNDI is the entity that associates names with objects and it provides a facility to find an object based on its name. Directory service in JNDI is the naming service that has been extended and enhanced to provide directory object operations for manipulating attributes.

JNDI insulates the application from the protocol and from implementation details and locates the media objects. JNDI uses the environment properties employed in creating the initial context factory to look up the media object stored in the directory. The clients can locate the specific media EJB container that contains the media bean through the JNDI service. They make use of the EJB container to invoke the media bean and get the reference to the EJB object instance. In this proposed method, demand by each client is achieved through a servlet to bean communication for a specific period of time. The view-media servlet calls the media bean, which in turn invokes the view-media method, that returns the media data in stream of bytes and then it is displayed for viewing.

2.8.6 Media EJB Data Flow Model

The clients use the servlet to make a request for a media in the media server. It uses the JNDI to lookup the media objects over a network. A client accesses the media bean through its remote and home interfaces. When the client performs a JNDI lookup for a home object, EJB container uses JNDI to
return RMI stub. The RMI stub is a proxy for the media Home object, which is located elsewhere in the network and once the media client has the stub, it can invoke the `view-media` method on the home object through the stub. The EJB object that implements the remote and home interfaces are accessible from a client through the standard RMI APIs as shown in Figure 2.30.

![Figure 2.30 Media Method Invocation on the Remote EJB Server](image)

EJB object communicates with the remote EJB container thus requesting the media bean method and then communicates with the media bean. The media EJB container executes the media bean and sends the media data back to the client via servlet at specific intervals.

### 2.8.7 Media Bean - Life Cycle

The life cycle of a media bean is shown in Figure 2.31 and the steps are explained as follows:
Figure 2.31 Life Cycle of Media Bean

- The media bean leaves the non-existence state and enters the pooled state when it is added in to the pool by the container at which point newInstance() and setEntityContext() are invoked on it.

- The media bean transits from the pooled state to the ready state when the container selects the instance to handle an ejbCreate() request, at which point ejbCreate() and ejbPostCreate() are invoked on the media object. The container selects the instance to be activated at which point ejbActivate() is invoked on the instance.

- The media bean transits from the ready state to the pooled state when the container selects the instance for passivation at which
point ejbPassivate() is called on the media object. The client invokes the remove() method on the instance at which point ejbRemove() method is invoked.

- When the media bean is in the ready state it accepts the invocations of its business methods with ejbLoad() and ejbStore() methods. ejbLoad() method reads the media data from the database and use it to populate instance variable. ejbStore() writes data to the database.

2.8.8 Media Bean - Deploying Procedure

In order to deploy the media EJB component into the server, the steps to be followed are:

- Start the J2EE deploy tool window and select the new application.
- Choose the corresponding enterprise archive file and type the application display name.
- Start the new Enterprise wizard to package the media bean and type the JAR display name.
- Add the media.class, and mediahome.class and mediaejb.class to JAR display box.
- In the General dialog box, choose the bean type as Bean managed persistent entity Bean and choose appropriate interfaces in the Enterprise Bean class and enter the name of the Enterprise bean.
• Open the deploy wizard and give the full path of client’s jar file name which contains the stub classes and it will enable remote access to the media bean.

• Enter the JNDI name and WAR context root and deploy the media bean.

A reusable component for media on demand has been deployed successfully providing increased productivity for media applications.

2.9 MOBILE AGENT MODEL FOR MEDIA ON DEMAND

Multiagent systems have been recently utilized in developing scalable software systems on heterogeneous networks (Fricke et al 2001, Papazoglou 2001, Jennings and Woodridge 1998). Using Java a cross platform language, distributed systems based on agents provide inherent scalability and autonomy. Software agents are envisioned to perform many activities on behalf of human beings and many applications are built using mobile agent paradigm (Griss and Pour 2001). Mobile agent computing on a network of workstations is envisioned to be a powerful paradigm for building distributed applications. Mobile agents have the unique ability to transport themselves from one system in a network to another thereby reducing network traffic and providing a means of overcoming network latency.

Due to large size and high bandwidth requirements of digital video, server and network bandwidths are proving to be major limiting factors in the use of video over the Internet. To reduce the latency due to the limited bandwidth, the idea is to incorporate caching that increases the efficiency of
these media applications in distributed environment. Explosive growth in demand for web-based video applications justifies the need for caching video at a proxy server close to the client, which reduces delay without sacrificing the quality. A proxy cache stores the recently accessed video data and serves the clients quickly for future requests without contacting the main server.

There are several approaches to distributed caching. The harvest group designed the Internet Cache Protocol (ICP) (Wessels and Claffy 1997), which supports discovery and retrieval of documents from neighboring caches as well as parent caches. Another approach to distributed caching is Cache Array Routing Protocol (CARP) (Valloppillil and Ross 1998), which divides the URL space among an array of loosely coupled caches and allows each cache to store only the documents whose URLs are hashed to it. This model differs from majority of existing proxy caches in that it concentrates exclusively on video while other caching approaches are for HTML documents and images.

Caching videos near clients reduces start up and the playback delays and also reduces server load by intercepting a large fraction of server accesses. As loading of the server is reduced, scalability to large number of users is achieved.

Mobile agent based video caching is achieved for a Server-Proxy-Multiclient architecture which enables the clients to access the media server at any time for the required media. When a request comes from the client, the mobile agent dispatches itself to the proxy server and checks for the media data. If the data is unavailable it fetches the media data from the main server, stores a copy in the intermediate server and then displays it to the client. For the
subsequent requests from the clients, the media data is made available quickly. As the mobile agent has the ability to operate asynchronously and independently of the process that created it, it builds a robust distributed environment.

2.9.1 Distributed Caching Architecture for Media on Demand

Caching has been recognized as one of the effective schemes to alleviate the service bottleneck and reduce the network traffic, thereby minimizing the user access latency. In a distributed caching system (Jennings et al 1996) there are no intermediate cache levels to serve the requests other than the institutional caches. In this proposed model, a distributed caching has been set up using agents to deliver media as demanded by multiple clients from a media server and the functional model is shown in Figure 2.32.

![Distributed Caching in Media on Demand Model](image)

Figure 2.32 Distributed Caching in Media on Demand Model
When a request comes from a client, the creation of mobile agent (Media Aglet) takes place in the context available in the client host. The new aglet is assigned an identifier inserted into the context and initialized. The media aglet starts its execution as soon as it has been successfully initialized. The media aglet dispatches itself to the intermediate (proxy) server. Dispatching an aglet from client context to proxy context will remove it from its current context and insert it into the destination context where it will restart execution. This media aglet makes the computation regarding the check for the availability of the media file in the proxy. If the requested media data is available in the proxy it fetches the data and displays it to the client by invoking the media player.

When the media file is not present in the proxy then the media aglet clones another aglet and dispatches it to the main media server and it gets deactivated. The cloned aglet upon arrival at the media server identifies the required media file and fetches it to the proxy, stores a copy of the file, activates the media aglet and disposes itself. The media aglet now obtains a reference to the media object and does object serialization to transport the media data to the client. The client upon receiving the media data displays it.

The media aglet, which is available on the client side, has abstractions that are needed to leverage the mobile agent technology. The aglet has a proxy, which is a representative of an agent provides location transparency for the aglet. The aglet context is a stationary object that provides the means for maintaining and managing aglets in a uniform execution environment is shown in Figure 2.33 as a life cycle Model.
On the server side of this application are media server and database, which store all the media data information. On the client side, the aglet is a mobile agent residing in its context that will dispatch itself to the proxy server side on client request and does the required functions in the proxy. Both client and server objects are considered as remote objects and hence inter-remote object communication is achieved. If the required object is not available in the proxy then the proxy registers with the media server object and fetches the data by invoking appropriate methods using aglets. The server object uses individual thread of control to distribute the media simultaneously to the clients registered with it. The media aglets obtain the necessary data from the registered clients and responds back to them respectively with the requested media.

2.10  PREFETCH CACHING MODEL FOR MEDIA ON DEMAND

The performance of the media on demand model is improved by caching the media data at proxies but the benefit from this technique is limited.
(Douglis et al 1997, Kroeger et al 1997). The maximum hit rate that can be achieved by any caching algorithm is usually not more than 50%. One way to further increase the cache-hit rate is to anticipate the future media request and prefetch those media files in a local cache. Kroeger analyzed the performance limits of prefetching between web servers and proxies and showed that combining perfect caching and prefetching at the proxies can reduce the latency by 60% for high bandwidth clients. Prefetching can be applied between proxy server and media server or media clients and proxies. In the proposed model prefetching is applied between proxy server and media server. The proxy server regularly prefetches the popular media data from the media server and provides it to the clients thereby reducing latency. An RMI based prefetch model has been implemented which improves the performance around 84% in a single machine scenario for the data in Table 2.1.

2.11 MULTICAST MODEL FOR MEDIA ON DEMAND

Multicasting allows the deployment of multimedia applications on the network while minimizing their demand for bandwidth. Distributing significant amounts of identical data from a single sender to multiple clients can take considerable time and bandwidth if the sender sends a separate copy to each client. Multicasting over a network allows a sender to distribute data to all interested parties while minimizing the use of network resources. Multicasting takes the strengths of both unicast and broadcast approaches and sends a single copy of the data to those clients who request it. Multiple copies of data are not sent across the network, nor is data sent to clients who do not want it.
Media clients in a multicast group are dynamic that can join or leave at any time for which no elaborate scheme is required to create or disband a group. When a group has no members, it ceases to exist on the network. Groups also scale upward easily, as more clients join a multicast, it becomes more likely that the multicast is already being routed close to them. When a media client joins a group, a message is sent to the media client's local router to inform the router that the client wants to receive data sent to the group and the media client sets its process and network card to receive the multicast on the group's address and port.

Class D IP addresses, 224.0.0.1 to 224.0.0.15 were used to test the media application that simplifies the implementation of IP multicasting on Ethernet. The media data is requested by the group of clients, media is divided into packets of size 5000 bytes and transmitted to all clients. When a media client leaves a group and is the only one receiving the multicast on that particular sub network, the router stops sending data to the client's sub network, thereby freeing bandwidth on that portion of the network. Media servers and media clients do not require any hardware upgrades in order to take advantage of multicasting. .NET Remoting based distributed multicast model for media on demand has been successfully implemented that provides an efficient way to enable these multimedia applications on the network, which dramatically reduces the network bandwidth and cost.

2.12 SECURE MODEL FOR XMLISED MEDIA DATA

A distributed model is constructed through which a secured media on demand is presented. As the size of the video databases increases, the vulnerability to attack also increases. Hence it becomes necessary to secure the
data at storage and transmission and provide access only to authorized clients. Though the authorized users can only access the video database, the data is not safe during transmission due to the noise or due to attacks by hackers. Hence in media applications it becomes necessary to protect the video data against these attacks, which is achieved through encryption.

Extensible Markup Language (XML) is being used in a totally different environment in order to easily describe the structure of the media stream. XML security appliance features translation, validation archiving and conditional routing of the media data. The present scenario of web applications is to use XML to transport data from databases. The goal of XML is to enable data to be served, received and processed on the web in the way that is as easy as that currently made possible by the use of HTML. As XML is hierarchical, ubiquitous, platform independent, it solves the problems due to incompatibility in networks.

In this proposed model the authorization of the client is initially checked and then the video data requested by the authorized client is converted into XML data, encrypted using Advanced Encryption Standard and then transported to the client. The client decrypts the data and plays back the media. This model outlines a new approach to develop a solution for a secured media on demand by way of distributed computing.

This algorithm is based on AES standard proposed by National Institute of standards and Technology and designed by Joan Daemen and Vincent Rijmen called as Rijndael algorithm. This algorithm has resistance against all known attacks, speed and code compactness on a wide range of platforms. Advanced Encryption standard is a symmetric key encryption
Rijndael algorithm is an iterated block cipher with variable block length and variable key length that can be independently specified to 128, 192 or 256 bits. The block is an organized array of bytes with four rows. The number of columns is equal to the block length divided by 32. The input bytes are mapped to the array and the cipher key is organized in the same form as block. The algorithm consists of number of rounds of computation that varies with respect to the block and key length. The number of rounds needed for a block and key of length 128 is ten and the number of rounds increases as the length increases. In each round four different transformations are carried out to encrypt every block of data. In every round four basic steps namely Bytesub transformation, Shiftrow transformation, Mixcolumn transformation and adding Roundkey are executed. All these operations are done for 10 rounds, as the key length is chosen as 128. During decryption each of the steps is inverted namely Inv Bytesub, Invshift Row and Inv Mixcolumn are performed to retrieve back the original media data.

In this proposed model, media client calls a method of a remote object through the stub, which presents the same interfaces as the remote object itself. In this method at the first phase the media data is converted into XML data by means of base64 encoding. In the second phase the encoded data is encrypted using Rijndael algorithm as shown in Figure 2.34.

![Figure 2.34 Media on Demand Encryption Model](image_url)
In this model, the media data is inserted into the oracle database after the connection has been established between the database server and the media server. When a request comes from the client for the media data, authorization of the client is carried out. For an authorized user, the media is retrieved from the database, encrypted and written to an output stream. The stub at the client unmarshalls the stream and carries out the reverse operations of the method. In the first phase it decrypts the XML data and in the second phase it decodes the XML data into binary data and returns it to the original caller as shown in Figure 2.35.

Figure 2.35 Media on Demand Decryption Model

Thus a secured transmission of media is achieved and the runtime object associated with the media application controls the media delivery process.

2.13 CONCLUSION

Distributed models for media data on demand have been implemented in HP workstations connected in an Ethernet LAN. Although the client-server architecture for media on demand is well established, this thesis emphasizes a unique methodology based on RMI, RMI-IIOP, CORBA and .NET Remoting. The performance analysis of the models has been carried out
and a dynamic re-routing model in .NET framework has been proposed and implemented to enhance the reliability in media transmission.

A component model for media on demand has been proposed and implemented that increases the reusability and scalability of the system. An agent-based model has been implemented with distributed caching that reduces the network latency and provides an uninterruptible service for media clients. A multicast model has been developed for a group of clients that achieves a reduction in the bandwidth consumed by the media application. An XML secured distributed model has been deployed with the media data being encrypted with Advanced Encryption Standard. AES will remain as an unbreakable encryption tool for many years and hence this secured distributed model for media on demand will sustain the security issues in networks. The congestion control for media transmission and other security issues in media on demand are dealt in the forthcoming chapters.