**INDEX**

|  |  |
| --- | --- |
| **Declaration** | 2 |
| **Certificate** | 3 |
| **Acknowledgement** | 4 |
| **Abstract** | 8 |
| **1.Introduction** | 9 |
| **2.Java Technology** | 12 |
| 2.1. Introduction | 12 |
| 2.2. Java Platform | 12 |
| 2.2.1 Java Virtual Machine | 12 |
| 2.2.2. Java Application Programming Interface | 13 |
| **3.Java Media Framework** | 15 |
| 3.1. Streaming Media | 15 |
| 3.2.Architecture | 17 |
| 3.2.1.Managers | 18 |
| 3.2.1.Event Model | 19 |
| 3.2.2. Data Model | 19 |
| 3.3.Presentation | 20 |
| 3.4. Capture | 22 |
| 3.5.Media data storage and transmission | 23 |
| **4. Real Time Transport Protocol** | 24 |
| 4.1 Streaming Media | 24 |
| 4.2 RTP Architecture | 24 |
|  |  |

|  |  |  |
| --- | --- | --- |
| 4.3.RTP Applications | | 27 |
| **5. JMF RTP API** | | 28 |
| 5.1. RTP transmission and reception | | 28 |
| **6. Requirements and Specifications** | | 34 |
| 6.1. Feasibility Analysis | | 34 |
| **7.Use Case Diagrams** | | 35 |
| **8. Functionality** | | 38 |
| 8.1 Server side connectivity | | 38 |
| 8.2 Client side connectivity | | 58 |
| **9.Snapshots** | | 61 |
| **10.** | **Conclusion** | 63 |
| 11. | **References** |  |
|  |  |  |

**LAN CHAT- AUDIO/VIDEO CHAT**

Abstract

Webcams are popular, relatively low cost devices which can provide live video and audio streams via personal computers, and can be used with many software clients for both video calls and videoconferencing. LAN Videoconferencing uses audio and video telecommunications to bring people at different sites together which are connected through local area network. This can be as simple as a conversation between people in private offices (point-to-point) or involve several (multipoint) sites in large rooms at multiple locations. Besides the audio and visual transmission of meeting activities text chat facility is also present.

The aim of this project is to create a suite of programs using java that implements audio and video chat on systems connected through LAN without the need of any internet connection, additional installations, creating or handling accounts and passwords.

**INTRODUCTION**

High speed Internet connectivity has become more widely available at a reasonable cost and the cost of video capture and display technology has decreased. Consequently, personal videoconferencing systems based on a webcam, personal computer system, software compression and broadband Internet connectivity have become affordable to the general public.

With the introduction of relatively low cost , high capacity broadband and telecommunication services in the late 1990s, coupled with powerful computing processors and video compression techniques, VideoConferencing usage has made significant inroads in business, education, medicine and media. Like all long distance communications technologies (such as phone and Internet), by reducing the need to travel to bring people together the technology also contributes to reductions in carbon emissions, thereby helping to reduce global warming.

In present scenario , it is not always possible for a person to be physically present at the required place just at the moment . It is at this moment of urgency to be at a particular place that we require softwares that allow video chat over a LAN and are found to be useful. It reduces the physical movements and extends a feel of comfortability to the people by allowing them to do a video chat and text through any two systems connected over a LAN. It also helps to attend meetings if a person could not be physically present at the place of meeting through Video Conferencing. It allows to instantly meet without prior notice and join meetings without anydownloads or installations. No account is needed to enable this type of conferencing.

LAN Videoconferencing uses audio and video telecommunications to bring people at different sites together which are connected through local area network. This can be as simple as a conversation between people in private offices (point-to-point) or involve several (multipoint) sites in large rooms at multiple locations. Besides the audio and visual transmission of meeting activities text chat facility is also present.

Webcams are popular, relatively low cost devices which can provide live video and audio streams via personal computers, and can be used with many software clients for both video calls and videoconferencing

The core technology used in a videoconferencing system is digital compression of audio and

video streams in real time. The hardware or software that performs compression is called

a codec (coder/decoder). Compression rates of up to 1:500 can be achieved. The resulting digital stream of 1s and 0s is subdivided into labeled packets, which are then transmitted through a digital network of some kind (usually ISDN or IP).

VideoConferencing adds another possible alternative, and can be considered when:

* a live conversation is needed;
* non-verbal (visual) information is an important component of the conversation;
* the parties of the conversation don’t need to physically come to the same location;
* Deaf, hard-of-hearing and mute individuals have a particular interest in the development of afford ble high-quality videoconferencing as a means of communicating with each other in sign language.

Videoconferencing provides students with the opportunity to learn by participating in two-way communication forums. Furthermore, teachers and lecturers can be brought to remote or otherwise isolated educational facilities. Students could easily attend their lectures from their hostels or being anywhere in the college.

A few examples of benefits that videoconferencing can provide in campus environments include:

* faculty members keeping in touch with classes while attending conferences
* Virtual meeting room with feature-rich controls in synchronization with all video and web tools.
* guest lecturers brought in classes held in different part of the same institution
* researchers collaborating with colleagues at institutions on a regular basis without loss of time due to travel
* schools with multiple campuses collaborating and sharing professors
* faculty members participating in thesis defenses at the institution

* administrators on tight schedules collaborating on budget preparation from different parts of campus;
* faculty committee auditioning scholarship candidates;
* researchers answering questions about grant proposals from agencies or review committees;
* interviews
* teleseminars.

Videoconferencing is a highly useful technology for real- time telemedicine and telenursing applications, such as diagnosis, consulting, transmission of medical images, etc... With videoconferencing, patients may contact nurses and physicians in emergency or routine situations; physicians and other paramedical professionals can discuss cases across large distances.

Videoconferencing can enable individuals in distant locations to participate in meetings on short notice, with time and money savings. Technology such as VoIP can be used in conjunction with desktop videoconferencing to enable low-cost face-to-face business meetings without leaving the desk, especially for businesses with widespread offices.

**JAVA TECHNOLOGY**

**2.1 Introduction**

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It is the underlying technology that powers state-of-the-art programs including utilities, games, and business applications. Java is fast, secure, and reliable .It is used in desktops, laptops, datacenters, game consoles, scientific supercomputers, cell phones, TVs etc.

**2.2 Java Platform**

Java platform is a software-only platform that runs on top of other hardware-based platforms. The Java platform has two components:

1. Java Virtual Machine
2. Java Application Programming Interface (API)

The API and Java Virtual Machine insulate the program from the underlying hardware.

**2.2.1 Java Virtual Machine**

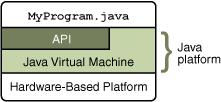
An implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

The Java Virtual Machine Specification defines an abstract machine or processor. The Specification specifies an instruction set, a set of registers, a stack, a "garbage heap," and a method area. Once a Java virtual machine has been implemented for a given platform, any Java program (which, after compilation, is called bytecode) can run on that platform. A Java virtual machine can either interpret the bytecode one instruction at a time (mapping it to a real processor

instruction) or the bytecode can be compiled further for the real processor using what is called a just-in-time compiler.

**2.2.2 Java Application Programming Interface *(API)***

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as *packages*.



**2.3 Java Programming Language**

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

|  |  |  |
| --- | --- | --- |
| i) | Simple | vi) Architecture neutral |
| ii) | Object oriented | vii) Portable |
| iii) | Distributed | viii) High performance |
| iv) | Multithreaded | ix) Robust |
| v) | Dynamic | x) Secure |

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A

.class file does not contain code that is native to one’s processor; it instead contains *bytecodes* — the machine language of the Java Virtual Machine1 (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine. Because the Java VM is available

on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris Operating System (Solaris OS), Linux, or Mac OS.

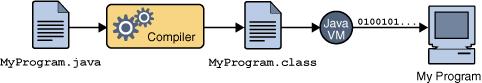
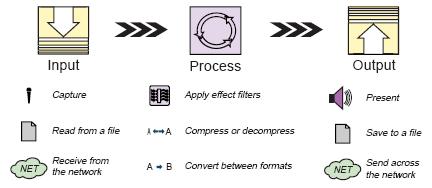


Fig-1 An overview of the software development process.

**JAVA MEDIA FRAMEWORK**

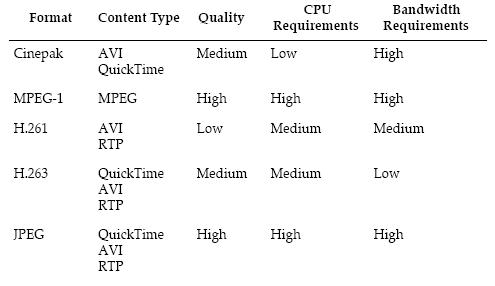
**3.1 Streaming Media**



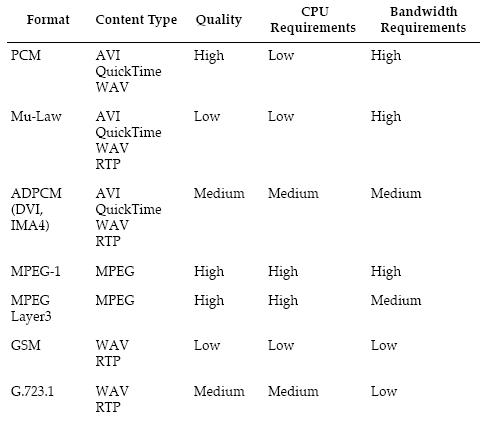
**Fig-2 Media processing model**

Any data that changes meaningfully with respect to time can be characterized as time based media. Audio-clips, movie clips and animation are common form of time-based media. They can be obtained from various sources like network files, camera, microphones, & live broadcasts.

Time-based media is also referred as streaming media- it is delivered in a steady stream that must be received & processed within a particular time frame to produce acceptable results. Media data is in media streams that are obtained from a local file, acquired over network or captured from a camera or microphone.



**Fig-3 Common Video formats**



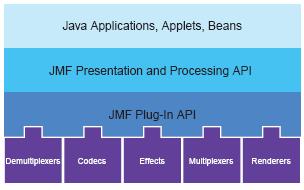
**Fig-4 Common Audio formats**

These time-based media is represented through o/p devices like speakers & monitors. An output destination for media data is referred as a Data-Sink. In many cases, the presentation of the media stream can’t begin immediately. This is latency experienced before begin of presentation. The data in a media stream is manipulated before it is presented to the user.

Time-based media can be captured from a live source for processing & playback. For this we need capture devices. Captures devices can be characterized as either push or pull source. A still camera is a pull source- the user controls when to capture an image. A microphone is a push source-the live source continuously provides a stream of audio.

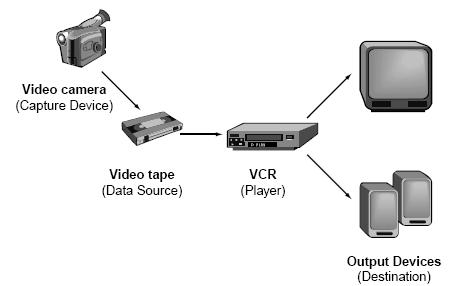
Java Media framework (JMF) provides architecture & messaging protocol for managing the acquisition, processing and delivery of time-based media data. JMF is designed to support most standard media content types such as AIFF, AVI, GSM, MIDI, MPEG, WAV etc. JMF implementations can leverage the capabilities of the underlying operating system, while developers can create programs that feature time-based media by writing to the JMF API. With JMF, developers can easily create applets & applications that present, capture, manipulate and store time-based media.

**3.2 Architecture**



**Fig-5 High-level JMF Architecture**

When you play a movie using VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads & interprets the data & sends appropriate signals to TV and speakers.



**Fig-6 Recording, Processing and presenting time-based media.**

JMF uses this same basic model. A *data source* encapsulates the media stream much like a video tape and a *player* provides processing & control mechanism similar to VCR. Playing and capturing audio and video with JMF requires input and output devices

**Managers**

The JMF API consists of interfaces that define behavior and interaction of objects. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used with existing classes.

There are four managers:

**Manager** handles the construction of Players, Processors, DataSources and DataSinks.

**PackageManager**maintains a registry of packages that contain JMF classes.

**CaptureDeviceManager**maintains a registry of available capture devices.

**PlugInManager**maintains a registry of available JMF Plug-In processing components.

If you extend JMF functionality by implementing a new plug-in, you can register it with the PlugInManager to make it available to Processors that support the plug-in API.

**Event Model**

JMF uses a structured event reporting mechanism to keep JMF-based programs informed of the current state of the media system and enable JMF-based programs to respond to media-driven error conditions such as out-of data and resource unavailable conditions. Whenever a JMF object needs to report on the current conditions, it posts a MediaEvent. MediaEvent is subclassed to identify many types of events.

**Data Model**

JMF media players use DataSources to manage the transfer of media-content. A

**DataSource**encapsulates both the location of media and the protocol and s/w used to deliver themedia. A DataSource is identified by either a JMF MediaLocator or a URL.

JMF defines several types of DataSource objects categorized according to how data transfer is initiated:

**Pull Data-Source** the client initiates the data transfer and controls the flow of data from pulldata-sources. There are 2 types of pull Data-Sources: **PullDataSource**&

**PullBufferDataSource,** which uses a Buffer object as its unit of transfer.

**Push Data-Source** the server initiates the data transfer and controls the flow of data frompush data-sources. Push data-source include broadcast media, multicast media and Video on Demand. There are 2 types push data sources: **PushDataSource**&**PushBufferDataSource,** which uses a Buffer object as its unit of transfer.

JMF defines two types of specialty data sources, cloneable data sources and merging data sources.

A **Cloneable** data source can be used to create clones of either a pull or push DataSource. The

clones don’t necessarily have the same properties as the cloneable data source used to create them or the original DataSource.

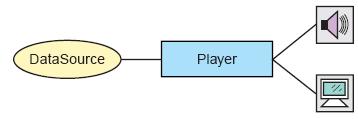
A **MergingDataSource** can be used to combine the SourceStreams from several DataSources into a single DataSource. This enables a set DataSources to be managed from a single point of control.

**3.3 Presentation**

The presentation process is modeled by the **Controller** interface. It defines basic state and control mechanism for an object that controls presents or captures time-based media. It defines phases that a media controller goes through and provides mechanism for controlling the transitions between those phases.

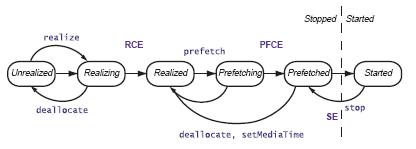
The JMF API has two types of Controllers: **Players** and **Processors.** They are constructed for a data source and not reused to present other media data.

**Players**



**Fig-7 JMF Player Model**

A **Player** processes an input stream and renders it at precise time. A DataSource is used to deliver input media stream to the Player. The rendering destination depends on the type of media being presented. A Player does not provide any control over the processing that it performs how it renders the media data.



**Fig-8 Player States.**

RCE – RealizedCompleteEvent PFCE – PrefetchCompleteEvent SE - StopEvent

A Player can be in one of the six states. The Clock interface defines the two primary states:

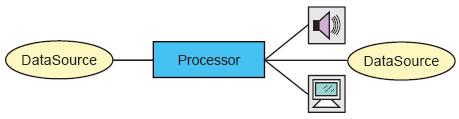
*Stopped* & *Started*. To facilitate resource management, Controller breaks down the stopped stateinto five standby states: *Unrealized, Realizing, Realized, Prefetching* and *Prefetched.*

Let us look at each states:

* A Player in Unrealized state has been instantiated but does not yet know anything about its media.
* When realize is called, a Player moves to the Realizing state. It is in process of determining its resource requirements. This might include rendering resources other than exclusive-use resource.
* After that, Player moves to Realized state. In this state, it knows what resources it needs and information about type media it is to present.
* When prefetch is called, Player enters into Prefetching state. Here, it preloads its media data, obtains exclusive-use resources and does whatever else it needs to do to prepare itself to play.
* When a Player finishes Prefetching, it enters into the Prefetched state. A Prefetched Player is ready to be started.
* Calling start puts Player into Started state. A Started Player object’s time-base time and media time are mapped and its clock is running.

A Player posts TransitionEvent as it moves from one state to another. The ControllerListener interface provides a way for your program to determine what state a Player is in and to respond appropriately. Using this event reporting mechanism, you can manage a Player object’s start latency by controlling when it begin Realizing and Prefetching.

**Processors**



**Fig-9 JMF Processor model**

**Processor** can also be used to present media data. A Processor is just a specialized type of Playerthat provides control over what processing is performed on the input media stream. It supports all of the same presentation controls as a Player. A Processor can output media data through a DataSource so that it can be presented by another Player or Processor, further manipulated by another Processor or delivered to some other destination, such as file.

Additional custom Control types might be supported by a particular Player or Processor implementation to provide other control behaviors and expose custom user interface components. A Player or Processor generally provides two standard user interface components, a visual component and a control-panel component.

A Processor allows application developer to define the type of the processing that is applied to the media data. This enables the application of effects, mixing and composing in real-time.

**3.4 Capture**

A multimedia capturing device can act as a source for multimedia data delivery. E.g. a microphone can capture raw audio input / a digital video capture board might deliver digital video from camera. Such Captured devices are abstracted as DataSource. E.g. a device that provides timely delivery of data can be represented as PushDataSource.

Some devices deliver multiple data streams. The corresponding DataSource can contain multiple SourceStreams that map to the data streams provided by the device

**3.5 Media Data Storage And Transmission**

A DataSink is used to read media data from a DataSource and render the media to some destination. A particular Datasink might write data to a file, write data across the network, or function as an RTP broadcaster.

Like Players, DataSink objects are constructed through Manager using a DataSource. A DataSink can use a StreamWriterControl to provide additional control over how data is written to

the file.

**REAL-TIME TRANSPORT PROTOCOL**

**4.1 Streaming Media**

When media content is streamed to a client in real-time, the client can begin to play the stream without having to wait for the complete stream to download. The term *streaming media* is often used to both technique of delivering content over the network in real-time and the real-time media content that is delivered. Through this, streaming media is changing the way people communicate and access information.

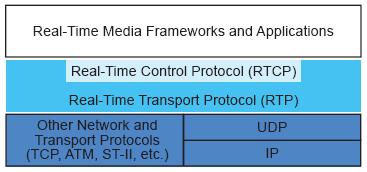
Transmitting media data across the network in real-time requires high network throughput. It is easier to compensate for lost data than to compensate for large delays in receiving the data.

Consequently, the protocols used for static data such as TCP don’t work well for streaming media.

So, underlying protocols other than TCP are typically used. Such a protocol is User Datagram

Protocol (UDP). UDP is a general transport layer unreliable protocol. It’s a low level networking protocol on top of which more application specific protocols are built. The Internet standard for transporting real-time data (audio-video) is the Real-time Transport Protocol (RTP).

**4.2 RTP Architecture**



**Fig-10 RTP architecture**

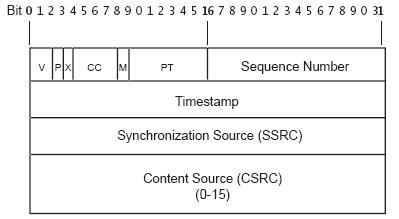
RTP provides end-to-end network delivery services for the transmission if real-time data. RTP can be used over both unicast and multicast network services. Over unicast, separate copies of the data are sent from the source to each destination. Over multicast, the data is sent from the source only once and the network is responsible fro transmitting the data to multiple locations.

This is more efficient for multimedia application such as video conferences.

RTP enable you to identify the type of data being transmitted, determine what order the packets of data should be presented in and synchronize media streams from different sources. RTP data packets are not guaranteed arrive in the order they were sent and not guaranteed to arrive at all. This is augmented by a control protocol (RTCP) that enables you to monitor the quality of data distribution & provides control and identification mechanism for RTP transmissions.

An RTP *session* is an association among set of applications communicating with RTP. A session is identified by a network address and a pair of ports. One port is used for media data and other port is used for control data. A *participant* is a single machine, host or user participating in the session. Each media type is transmitted in different session. E.g. if both audio & video are used in conference, one session is for the audio and second session is used to transmit video data. This enables participants to choose which media types they want to receive.

**Data packets**



**Fig-11 RTP data packet header format**

The media data for a session is transmitted as a series of packets known as RTP stream. Each data packet in a stream contains two parts, a structure header & the actual data.

The header of an RTP data packet contains:

* **The RTP version number (V)**: 2 bits. The version defined by the current specification is

2.

* **Padding** (P): 1 bit. If the padding bit is set, there are one or more bytes at the end of thepacket that are not part of the payload. The very last byte in the packet indicates the number of

bytes of padding. It is used by some encryption algorithms.

* **Extension** (X): 1 bit. If the extension bit is set, the fixed header is followed by oneheader extension. This extension mechanism enables implementations to add information to the

RTP Header.

* **CSRC Count** (CC): 4 bits. The number of CSRC identifiers that follow the fixed header.If the CSRC count is zero, the synchronization source is the source of the payload.
* **Marker (M)**: 1 bit. A marker bit defined by the particular media profile.
* **Payload Type** (PT**)**: 7 bits. An index into a media profile table that describes the payloadformat. The payload mappings for audio and video are specified in RFC 1890.
* **Sequence Number**: 16 bits. A unique packet number that identifies this packetÕsposition in the sequence of packets. The packet number is incremented by one for each packet

sent.

* **Timestamp**: 32 bits. Reflects the sampling instant of the first byte in the payload. Severalconsecutive packets can have the same timestamp if they are logically generated at the same time

N for example, if they are all part of the same video frame.

* **SSRC**: 32 bits. Identifies the synchronization source. If the CSRC count is zero, thepayload source is the synchronization source. If the CSRC count is nonzero, the SSRC identifies

the mixer.

* **CSRC**: 32 bits each. Identifies the contributing sources for the payload. The number ofcontributing sources is indicated by the CSRC count field; there can be up to 16 contributing

sources. If there are multiple contributing sources, the payload is the mixed data from those

sources.

**Control Packets**

Control data packets (RTCP packets) are periodically sent to all of participants in the session. They can contain information about the quality of service for the session participants, the source of the media and statistics pertaining to the data that has been transmitted so far.

There are several types of RTCP packets:

**Sender Report** A participant that has recently sent data packets issues a*Sender Report*thatcontains the total number of packets and bytes sent & other information used to synchronize media streams from different sessions.

**Receiver Report** A participant periodically issues*Receiver Reports*for all sources fromwhich they are receiving data packets. It contains information about number of packets lost, the highest sequence number received and a timestamp used to estimate round trip delay.

**Source Description** all compound RTCP packets must include a source description (SDES)element that contains Canonical name (CNAME) that identifies the source. Additional information might be included.

**Bye** When a source is no longer active, it sends an RTCP BYE packet. It may include thereason that the source is leaving the session.

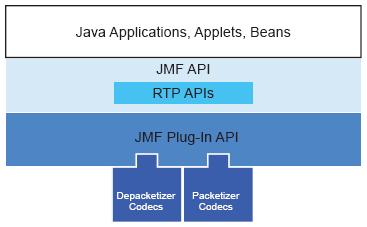
**Application Specific** RTCP APP packets provide mechanism for applications to define andsend custom information via RTP control port.

**4.3 RTP Applications**

RTP applications are often divided into those that need to able to receive data from the network (RTP Clients) and those that need to be able to transmit data across the network (RTP Servers). Some applications do both.

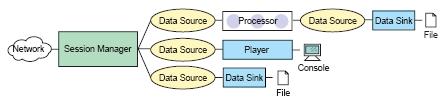
**JMF RTP API**

**5.1 RTP API Architecture**



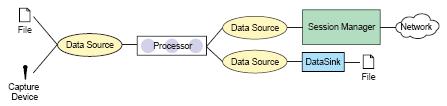
**Fig-12 High-level JMF RTP architecture**

JMF enables the playback & transmission of RTP streams through the APIs defined in the **javax.media.rtp, javax.media.rtp.event**and**javax.media.trp.rtcp**packages. JMF RTP APIsare designed to work with the capture, presentation & processing capabilities of JMF. Players & Processors are used to present and manipulate RTP media streams. JMF can be extended to support additional RTP-specific formats and dynamic payloads through the standard plug-in mechanism.



**Fig-13 RTP reception**

User can play incoming RTP streams locally, save them to a file or both. Similarly they can use APIs to transmit captured or stored media streams across network. The outgoing streams can also be played locally, saves to a file or both.



**Fig-14 RTP transmission**

**Session Manager**

In JMF, a **SessionManger** is used to coordinate an RTP session. The session manager keeps track of the session participants and the streams that are being transmitted. It maintains the state of the session as viewed from the local participant. It also handles the RTCP control channel and supports RTCP for both sender & receiver. The SessionManager interface defines methods that enable an application to initialize & start participating in session, remove individual streams created by the application and close the entire session.

**Session Statistics**

The session manager maintains statistics for all of the RTP & RTCP packets and received in the session. It provides access to global reception and transmission statistics:

**GlobalReceptionStats-** maintains global reception statistics for the session. &

**GlobalTransmissionStats-** maintains cumulative transmission statistics for all local senders.Statistics for a particular recipient are: **ReceptionStats-** maintains source reception statistics for an individual participant. **TransmissionStats-** maintains transmission statistics for an individual send stream.

**Session Participates**

Each participant is represented by an instance of a class that implements the **Participant** interface. Participants can be passive or active. There is exactly one local participant that represents the local client/server participant. A participant can own more than one stream, each of which is identified by the synchronization source identifier SSRC) used by the source of stream.

**Session Streams**

For each stream of RTP data packets there is an **RTPStream** object. There are 2 types of RTPStream: **ReceiveStream** represents a stream that’s being received from a remote participant.

**SendStream**represents a stream of data coming from Processor or input DataSource that isbeing sent over the network.

**RTP Events**

RTP-specific events are used to report on the state of the RTP session and streams. To receive notification of RTP events, you need to implement the appropriate listener & register it with the session manager:

**SessionListener:** Receive notification of changes in the state of the session.

**SessionStreamListener:** Receives notification of changes on the state of an RTP streamthat is transmitted.

**ReceiveStreamListener**: Receives notification of changes on the state of an RTP streamthat is received.

**RemoteListener**: Receives notification of events or RTP control messages received froma remote participant.

**RTP Data**

Data Handlers

The JMF RTP APIs are designed to be transport-protocol independent. A custom RTP Data Handler can be created to work over a specific protocol. The RTPPushDataSource class defines the basic elements of JMF RTP data handler. It has both input data stream and output data stream & is used for the data channel or control channel of an RTP session.

A custom **RTPSocket** can be used to construct Player through the Manager. JMF defines name & location for custom RTPSocket implementation:

**<protocol package-prefix>.media.protocol.rtpraw.Datasource**

**Data Formats**

All RTP specific data uses an RTP specific format encoding as defined in the

**AudioFormat**&**VideoFromat**classes.

AudioFormat define 4 standard RTP encoding strings:

public static final String ULAW\_RTP = “JAUDIO\_G711\_ULAW/rtp”;

public static final String DVI\_RTP = “dvi/rtp”;

public static final String G723\_RTP = “g723/rtp”;

public static final String GSM\_RTP = “gsm/rtp”;

VideoFormat defines 3 standard RTP encoding strings:

public static final String JPEG\_RTP = “jpeg/rtp”;

public static final String H261\_RTP = “h261/rtp”;

public static final String H263\_RTP = “h263/rtp”;

**RTP Controls**

RTP API defines one RTP-specific control, **RTPControl. RTPControl** provides a mechanism to add a mapping between a dynamic payload and a Format and methods for accessing session statistics and getting the current payload Format.

**Reception**

The presentation of a RTP stream is handled by a Player. A **MediaLocator** is used to construct Player which has form:

rtp://address:port[:ssrc]/content-type/[ttl]

The Player is constructed & connected to the first stream in the session**.** If there are multiple streams in the session, session manager is required. User can receive notification from the session manager whenever a stream is added to the session and construct a Player for each new stream.

**Transmission**

A session manager can also be used to initialize and control a session so that you can stream data across the network. The data to be streamed is acquired from a Processor.

For example, to create a send stream to transmit data from a live capture source, you would:

1. Create, initialize, and start a SessionManager for the session.
2. Construct a Processor using the appropriate capture DataSource
3. Set the output format of the Processor to an RTP-specific format. An appropriate RTP packetizer codec must be available for the data format you want to transmit.
4. Retrieve the output DataSource from the Processor.
5. Call createSendStream on the session manager and pass in the DataSource.

Transmission is controlled through the SendSteram**start**&**stop** methods. When it is first

started, the SessionManager behaves as a receiver. As soon as SendStream is created, it begins to

send out RTCP sender reports & behaves as a sender host as long as one or more send streams exist. If all SendStream are closed, it becomes a passive receiver.

**REQUIREMENTS & SPECIFICATION**

6.1 **Feasibility Analysis**

Technical Feasibility

This project doesn’t required advance and higher technology. It requires only knowledge of Core

Java and Java Media Framework (JMF). It requires only knowledge of java.

Economical Feasibility

The development of project want cost too much as it requires microphone & Web Camera as extra hardware besides computer. It involves very few persons and it does not require any outside professionals. As project is based on java so we don’t have any extra cost of setting up a network. The tools that we have used are freeware and can be downloaded from internet.

OperationalFeasibility

This project is live project. Person using these applications does not require extra technical or computer skill. It can be easily operated.

**Application**

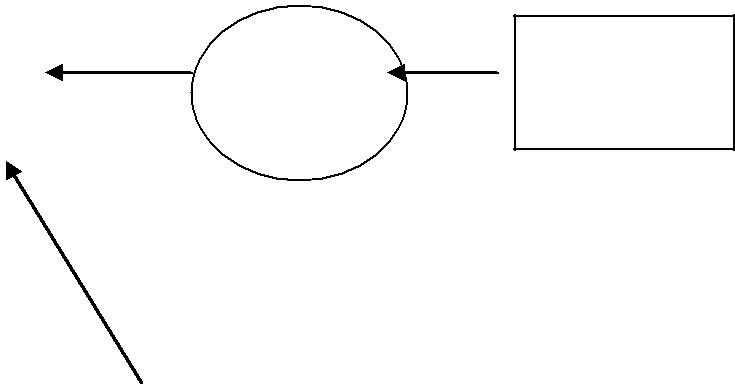
* To make application satisfying customer request.
* To provide good user interface.
* To provide better way of communication.

**Application Installation Process**

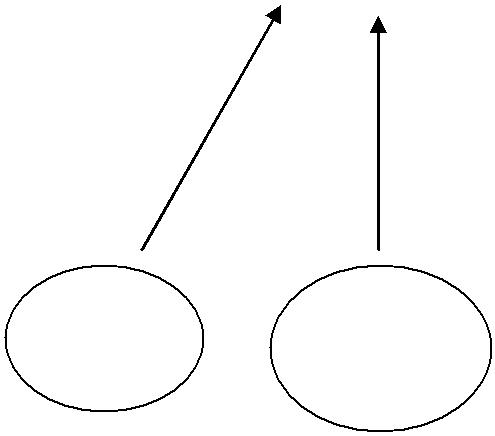
* Install audio drivers for Voice Chat.
* Install Web Camera for Video Conferencing.
* Install Java Development Kit v 1.6.
* Install Java Media Framework.

After successful installation execute the application.

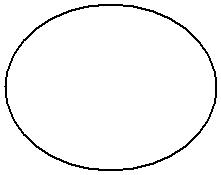
**USE CASE DIAGRAMS**



|  |  |  |
| --- | --- | --- |
| **User Info** |  | **Server** |
|  |  |  |

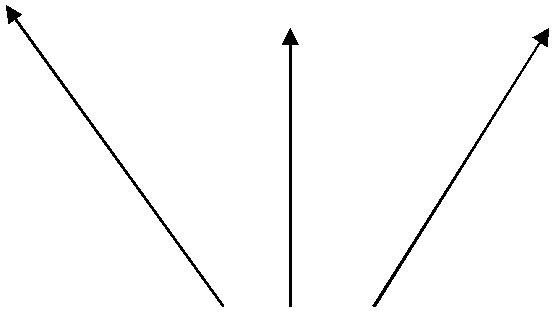


|  |  |  |
| --- | --- | --- |
| **Sign** | **New** |  |
|  |  |
| **in** | **User** |  |

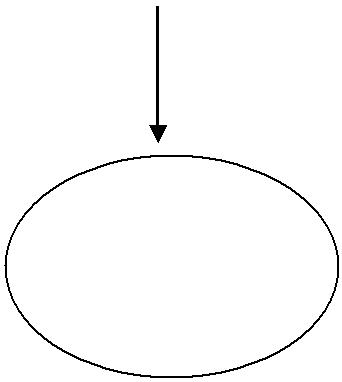


**Login** **Update** **Log**

**out**



**User**



**Chat Or**

**Conferencing**

**Fig-15 DFD Diagram**

**Server**

(Array Consisting

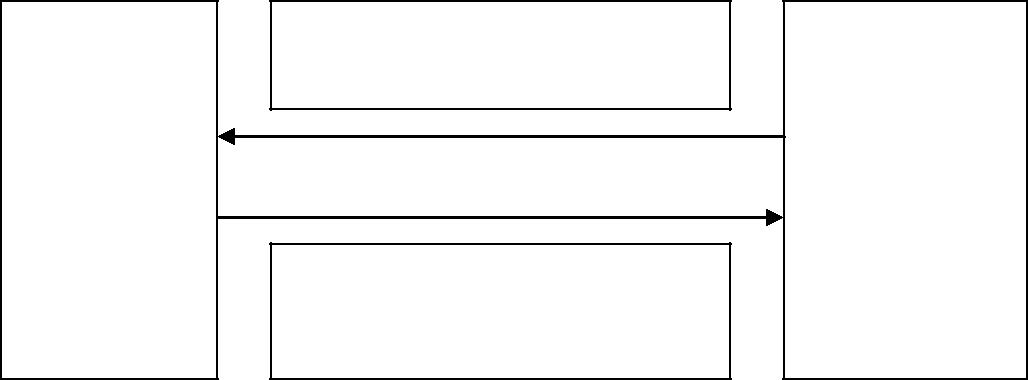
Client ID,

**Server**

(Array Consisting

Client ID,

Login Request



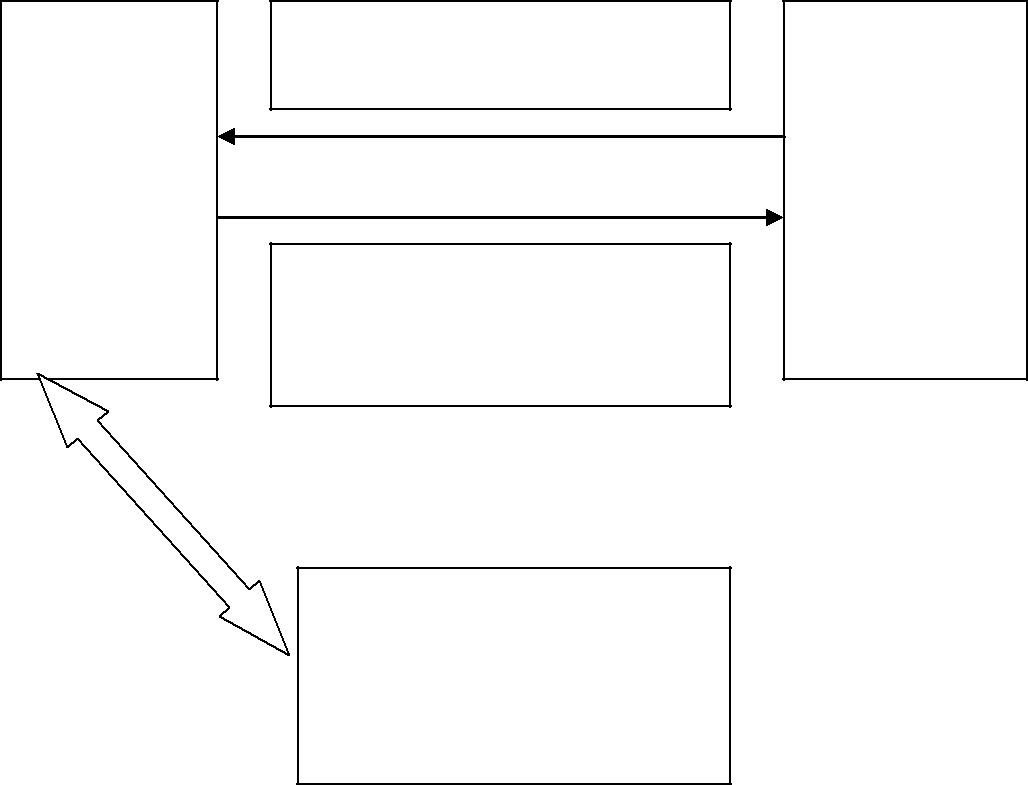
**Client**

(Server IP adr,

|  |  |  |
| --- | --- | --- |
| Request Accept | Server Port) |  |
|  |  |

Changes Status to 1

Fig-16 Login Request sent by first Client



Login Request

**Client N**

(Server IP adr,

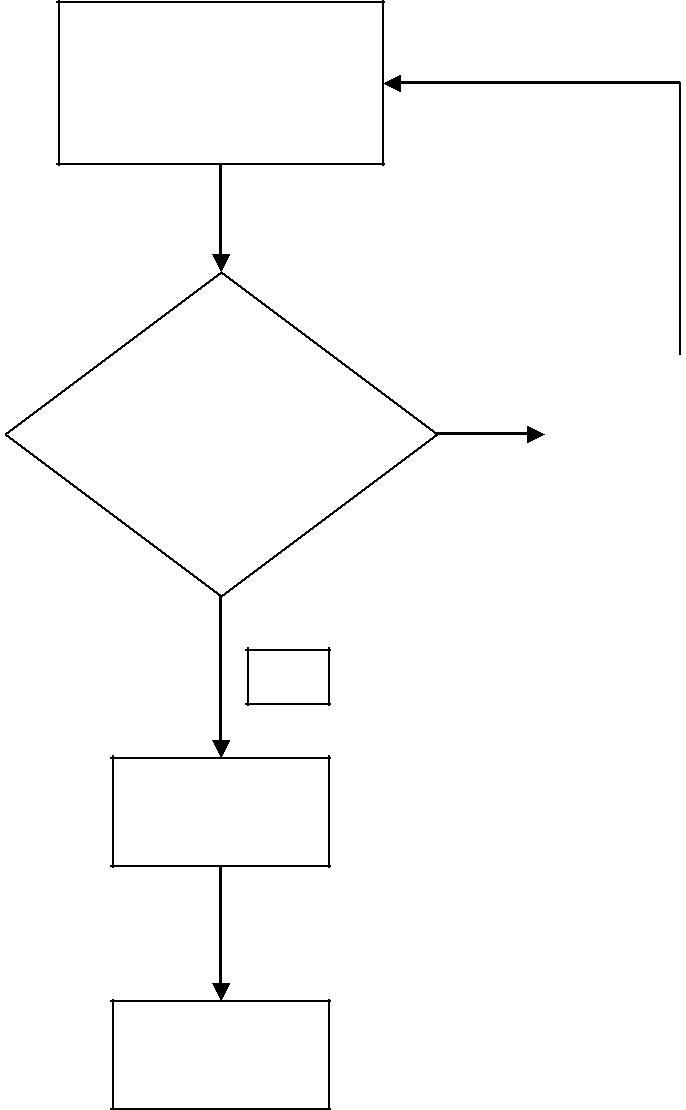
|  |  |  |
| --- | --- | --- |
| Request Accept | Server Port) |  |
|  |  |

Changes Status to 1

**Client**

( 1 to N-1) Receives entry for newly

Fig-17 Request sent by Client except the first one A Client Logs out



Client enters login name & password

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Login Name |  | **Y** |  | Report Client |
| Exists? |  |  |  | about Existence |
|  |  |  |  |  |

**N**

Enter into

Array

Enter into

Database

Fig-18 New User Creation

**FUNCTIONALITY**

**SERVER SIDE CONNECTIVITY**

**Building the Server**

**Step 1**— Start the project :

Go to Server directory and run this command:

npm init

This will start a new project. Provide all the details required. The *package.json*will be created and will look something like this.

{  
 "name": "chat",  
 "version": "1.0.0",  
 "description": "Chat application",  
 "main": "server.js",  
 "scripts": {  
 "test": "echo \"Error: no test specified\" && exit 1"  
 },  
 "author": "Your name",  
 "license": "ISC"  
}

**Step 2** — Install the dependencies.

* **socket.io** — is a *javascript* library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers.
* **express** — is a *Node.js* web application framework. It provides the set of features to develop the web and mobile applications. One can respond to HTTP request using different middlewares and also render HTML pages.

npm install --save socket.io  
npm install --save express

This will install required dependencies and add those to *package.json.*An extra field will be added to *package.json* which will look like this:

"dependencies": {  
 "express": "^4.14.0",  
 "socket.io": "^1.4.8"  
 }

**Step 3 —**Creating the Server

Create a server which serves at port 3000 and will send the html when called.

Initialize a new socket connection by passing HTTP object.

Event *connection* will be listening for incoming sockets.

Each socket emits *disconnect* event which will be called whenever a client disconnects.

* **socket.on** waits for the event. Whenever that event is triggered the callback function is called.
* **io.emit** is used to emit the message to all sockets connected to it.

The syntax is:

socket.on('event', function(msg){})  
io.emit('event', 'message')

Create a server with name *server.js.*It should:

* print a message to the console upon a user connecting
* listen for *chat message* events and broadcast the received message to all sockets connected.
* Whenever a user *disconnects,* it should print the message to the console.

The server will look something like this:

var app = require('express')();  
var http = require('http').Server(app);  
var io = require('socket.io')(http);

app.get('/', function(req, res){  
 res.sendfile('index.html');  
});

io.on('connection', function(socket){  
 console.log('user connected');  
 socket.on('chat message', function(msg){  
 io.emit('chat message', msg);  
 });  
 socket.on('disconnect', function(){  
 console.log('user disconnected');  
 });  
});

http.listen(3000, function(){  
 console.log('listening on \*:3000');  
});

**CLIENT SIDE CONNECTIVITY**

**Building the Client**

Create the index.html in the client directory, style.css in the css directory and app.js in js directory in the client.

***index.html:***

Let us write a simple HTML which can take our message and also display it.

Include *socket.io-client* and *angular.js* in your HTML script.

<script src="/path/to/angular.js"></script>  
<script src="/socket.io/socket.io.js"></script>

**socket.io** serves the client for us. It defaults to connect to the host that serves the page. Final HTML looks something like this:

<!doctype html>  
<html ng-app="myApp">  
 <head>  
 <title>Socket.IO chat</title>  
 <link rel="stylesheet" href="/css/style.css">  
 <script src="/lib/angular/angular.js"></script>  
 <script src="/socket.io/socket.io.js"></script>  
 <script src="<http://code.jquery.com/jquery-1.11.1.js>">  
 </script>  
 <script src="/js/app.js"></script>  
 </head>  
 <body ng-controller="mainController">  
 <ul id="messages"></ul>  
 <div>  
 <input id="m" ng-model="message" autocomplete="off" />  
 <button ng-click="send()">Send</button>  
 </div>  
 </body>  
</html>

***css/style.css:***

Give it some styling so that it looks like a chatbox. You can make use of any libraries.

\* { margin: 0; padding: 0; box-sizing: border-box; }  
body { font: 13px Helvetica, Arial; }  
div { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }  
div input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }  
div button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }  
#messages { list-style-type: none; margin: 0; padding: 0; }  
#messages li { padding: 5px 10px; }  
#messages li:nth-child(odd) { background: #eee; }

***js/app.js:***

Create an angular.js app and initialize a socket connection.

* **socket.on** listens for a particular event. It calls a callback function whenever that event is called.
* **socket.emit** is used to emit the message to the particular event.

Basic syntax of both are:

socket.on(‘event name’, function(msg){});  
socket.emit('event name', message);

So whenever the message is typed and the button is clicked, call the function to send the message.

Whenever the socket receives a message, display it.

The JavaScript will look something like this:

var app=angular.module('myApp',[]);

app.controller('mainController',['$scope',function($scope){  
 var socket = io.connect();  
 $scope.send = function(){  
 socket.emit('chat message', $scope.message);  
 $scope.message="";  
 }  
 socket.on('chat message', function(msg){  
 var li=document.createElement("li");  
 li.appendChild(document.createTextNode(msg));  
 document.getElementById("messages").appendChild(li);  
 });  
}]);

**Running the application**

Go to server directory where our server is present. Run the server using the following command:

node server.js

The server starts running on port 3000. Go to the browser and type the following url:

http://localhost:3000

**How to improve the same application**

You can design a database to save user details and messages. It would be good if the design was scalable so that you could add more features later.

You need to install Mongoose or a MongoDB module to make use of a Mongo database:

npm install --save mongoose

or:

npm install --save mongodb

Here’s the documentation to use [mongoose](http://mongoosejs.com/docs/index.html) and the [mongodb](https://docs.mongodb.com/getting-started/node/client/) module.

Here’s what my schema design looks like:

{  
 “\_id” : ObjectId(“5809171b71e640556be904ef”),  
 “name” : “Sudheesh Shetty”,  
 “handle” : “sudheesh”,  
 “password” : “556624370”,  
 “phone” : “8888888888”,  
 “email” : “sudheeshshetty@gmail.com”,  
 “friends” : [  
 {  
 “name” : “abc”,  
 “status” : “Friend”  
 },  
 {  
 “name” : “xyz”,  
 “status” : “Friend”  
 }  
 ],  
 “\_\_v” : 0  
}

Here, the status of each member can be:

* Friend: Stating that the member is a friend.
* Pending: If the member has not yet accepted.
* Blocked: If the member has blocked the other member.

Suppose the member has rejected a chat request. The sender can then send a chat request again. A user can also save the messages by creating an extra collection. Each document will have the message, sender, receiver, and time.

So design your database according to your specific needs and how you want to handle messages.

2. Create REST APIs to serve the client. For example, an endpoint that sends a home page, from which users can make other requests.

Few of my API endpoints are:

app.post(‘/register’,function(req,res){})

app.post(‘/login’,function(req,res){})

app.post(‘/friend\_request’,function(req,res){})

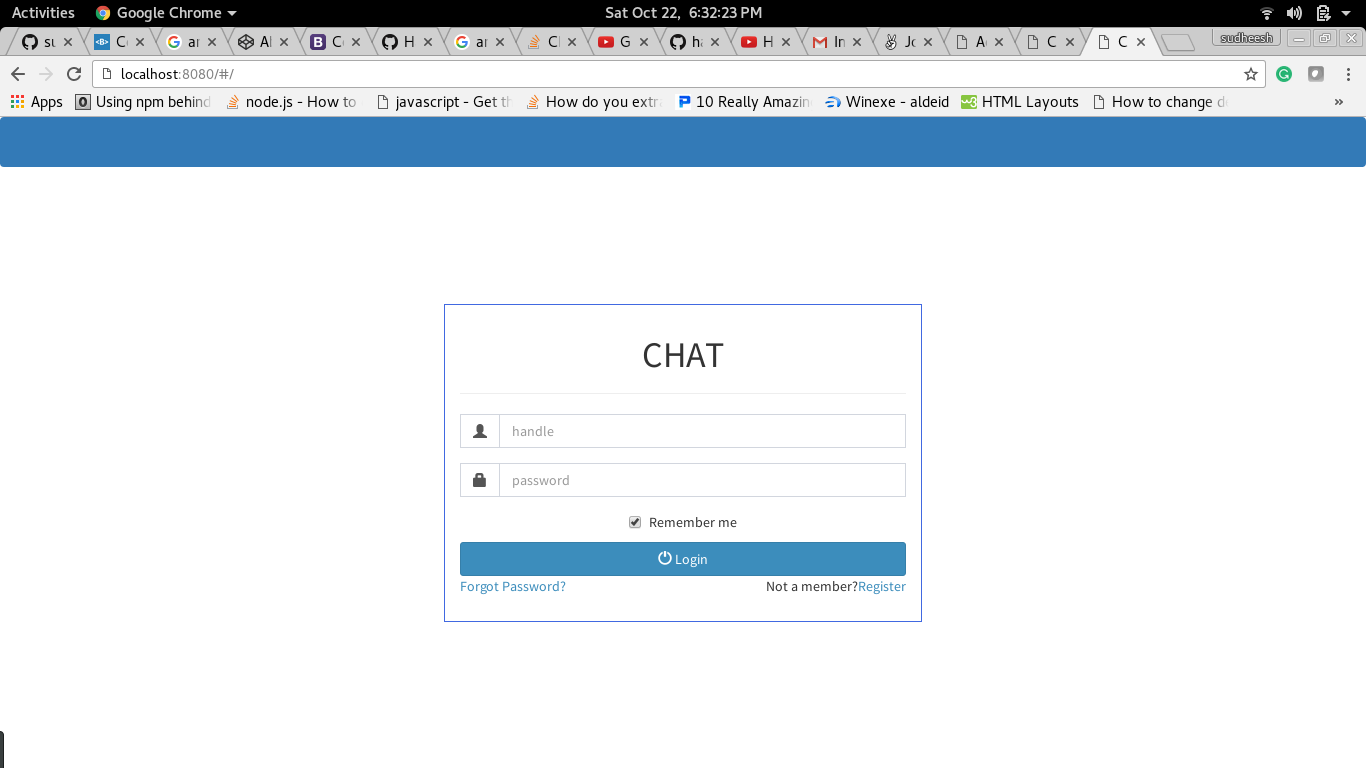
app.post(‘/friend\_request/confirmed’,function(req,res){})

3. Think of some cool additional features and implement them.

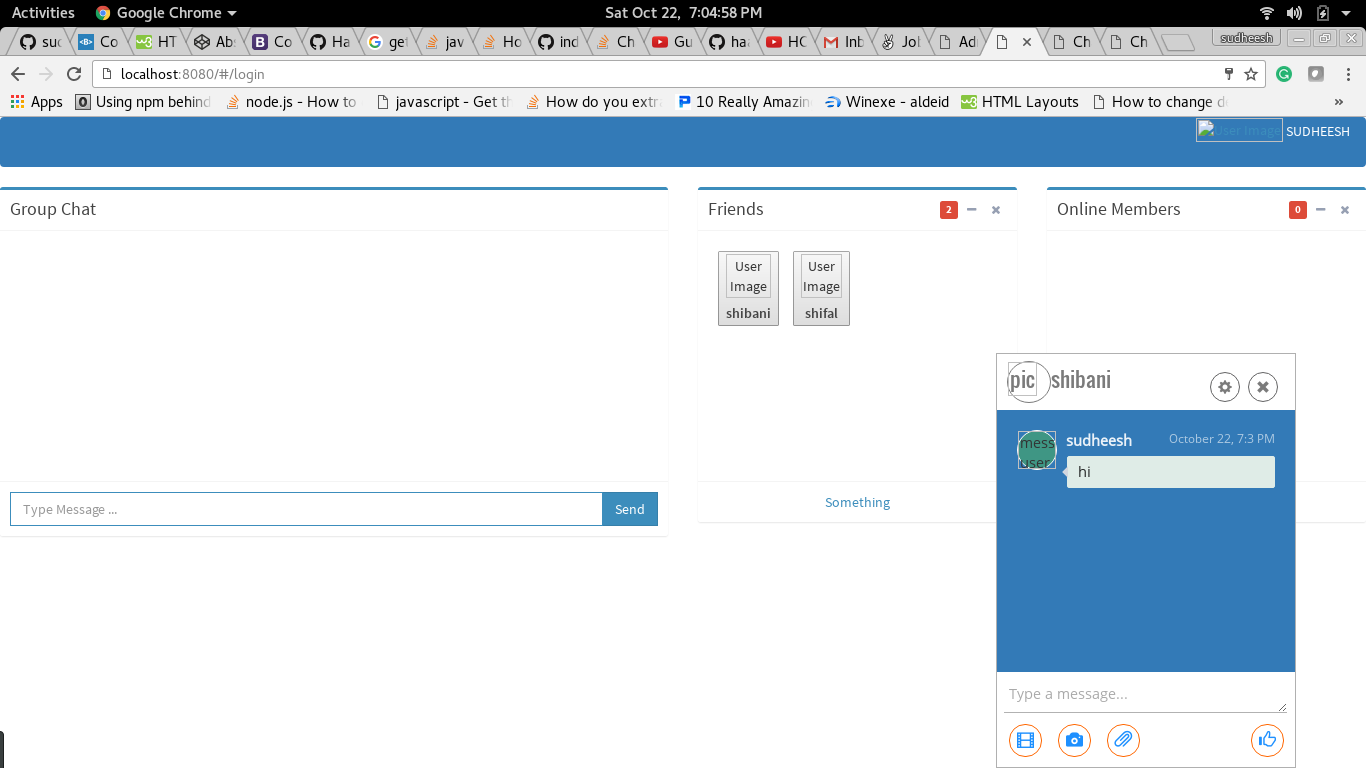
I have created a chat application of my own:

[sudheeshshetty/Chat  
Contribute to Chat development by creating an account on GitHub.github.com](https://github.com/sudheeshshetty/Chat)

Here’s a quick glance at my chat application:

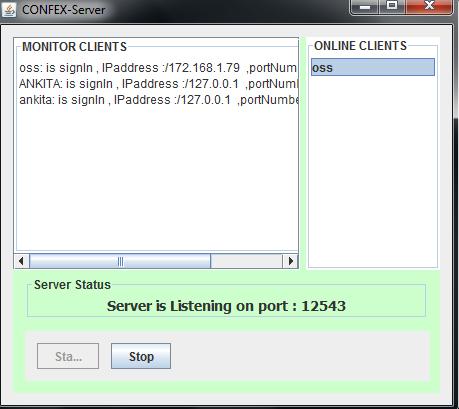


Login screen



How it looks after login.

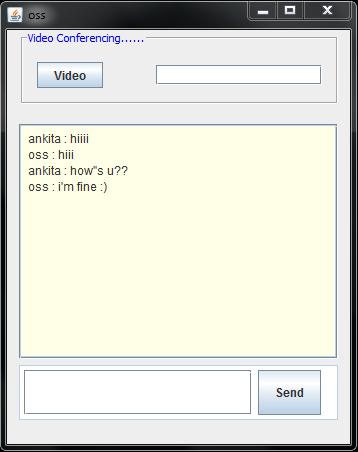
**SNAPSHOTS**



**fig 19: server start screen**



**fig 20:connecting window**



**fig 21: text chat window**



**fig 22: audio timer**

**CONCLUSION**

We have described our experiments as an application for audio/video chat based on Java platform using tools Java Media Framework & Real-time Transport protocol.

This software does not need any internet connection and is really handy as it requires no installations, maintaining accounts and discards the need to remember any password. It can easily be used at an office or an institution level to enhance interaction among the colleagues. Since it can be used over LAN, large number of people can connect to it without any additional expenses of other connections that are mandatory for internet.

The use of java platform makes this application even more convenient to use because java is platform independent, fast, secure, reliable and so it can be run on any machine having java installed in it.

Video conferencing helps in medical field in areas like telenursing and motivates technicality in medicine. It helps in easy diagnosis of problems related to various issues in cases of emergency

.It is also beneficial to the deaf and hard of hearing people as it facilitates the use of sign language.

It provides low cost face to face meetings without the actual physical presence of a person at the meeting place. It is especially beneficial to the institutes that are widespread.

This project can easily be extended to incorporate various other useful features like file transfer, voice mails and video mails.

**REFERENCES**

**Books:**

1. Complete Reference Java 2 by HerbetShieldt
2. Java Media Framework API Guide 2.0

**Websites**

1. www.java.com
2. www.sun.com
3. www.freesourcecode.com

**Tools Used**

1. Java Development Kit v1.6
2. Java Media Framework v2.1
3. Eclipse