# VoCe: Voice Conference

# CO324 project 1

GROUP 17,

E/12/218 MANAMGODA, M.G.T.R.

E/12/302 SAMARANAYAKA, A.K.L.L.

## CONTENT

## 1. A Brief Introduction to the problem.

Real-time conversational voice over the Internet is often referred to as **Internet telephony**, It is also commonly called **Voice-over-IP (VoIP)**. In this project we will design and implement a code with basic peer to peer voice conferencing application similar to Skype in two iterations.

In the first iteration we will implement voice communication between two parties which allows both parties to talk in both sides in the same time. Also in this iteration we also implemented the system to run multiple voice communication between these two machines.

## 2. Milestones.

### a. Implementing two way communications.

In this case we focused about transmit and receive full-duplex audio over a network between two parties by using UDP protocols. We need to use two separate threads for each end systems,

1. To capture/record Audio and send the byte stream.
2. To receive and Play the byte stream.

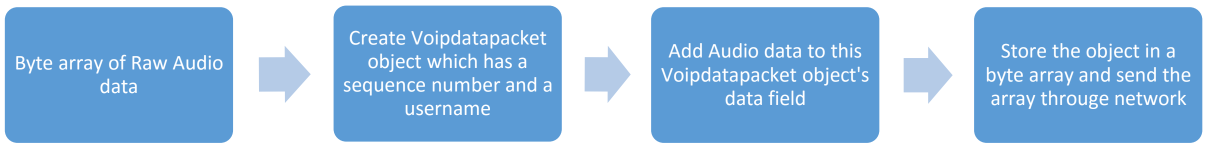
Our application must take the peer's IP and listening port of the other end system. These values are passed as a command line argument.

### b. Data serialization formats.

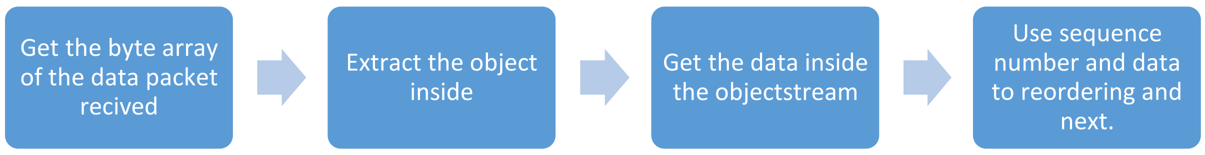
To serialize the byte stream first we get a byte array and then we put it inside of an object of Voipdatagram class which has data field to save audio data, a sequence number field to add a serial number. After this we again convert this into byte array and send over the network.

The benefits of serialisations are easiness of reordering and sorting the packets.

Serialization,


Byte array of Raw Audio data
Create Voipdatapacket object which has a sequence number and a username
Add Audio data to this Voipdatapacket object's data field
Store the object in a byte array and send the array througe network

Deserialization,


Get the byte array of the data packet recived
Extract the object inside
Get the data inside the objectstream
Use sequence number and data to reordering and next.

### c. Handling losses and reordering.

In this case, timing considerations and tolerance of data loss are particularly important. Timing Considerations are important because audio conversational applications are highly **delay-sensitive**. For a conversation with two or more interacting speakers, the delay from when a user speaks until the action is manifested at the other end should be less than a few hundred milliseconds. For voice, (End to End Delay)delays smaller than 150 milliseconds are not perceived by a human listener, delays between 150 And 400 milliseconds can be acceptable but are not ideal, and delays exceeding 400 milliseconds can result in frustrating, if not completely unintelligible, voice conversations.

**End-to-end delay** is the accumulation of transmission, processing, and queuing delays in routers; propagation delays in links; and end-system processing delays. The receiving side of a VoIP application will typically disregard any packets that are delayed more than a certain threshold, for example, more than 400 msecs. Thus, packets that are delayed by more than the threshold are effectively lost or considered lost.

**Packet Loss;** Sometimes the arriving IP datagram may be discarded, never to arrive at the receiving application. In which case packets may be lost. However, retransmission mechanisms are often considered unacceptable for conversational real-time audio applications such as VoIP, because they increase end-to-end delay].

But losing packets is not necessarily as disastrous as one might think. Indeed, packet loss rates between 1 and 20 percent can be tolerated, depending on how voice is encoded and transmitted, and on how the loss is concealed at the receiver.

In case of reordering what we used a fixed delay of 300 milliseconds and wait for packet to arrive while storing them in to an array list. After the play out delay we sorted the array and desterilize data and then play the audio.

To sort the array we used a sequence numbers.

### d. Jitter and Method to remove it.

**Packet Jitter**

A crucial component of end-to-end delay is the varying queuing delays that a packet experiences in the network’s routers. Because of these varying delays, the time from when a packet is generated at the source until it is received at the receiver can fluctuate from packet to packet. This phenomenon is called **jitter**.

If the receiver ignores the presence of jitter and plays out chunks as soon as they arrive, then the resulting audio quality can easily become unintelligible at the receiver. Fortunately, jitter can often be removed by using **sequence numbers** and a **playout delay**.

**Removing Jitter at the Receiver for Audio**

For our VoIP application, where packets are being generated periodically, the receiver should attempt to provide periodic play out of voice chunks in the presence of random network jitter. This is typically done by *Delaying playout of chunks at the receiver* the playout delay of the received audio chunks must be long enough so that most of the packets are received before their scheduled playout times. This playout delay can either be fixed throughout the duration of the audio session or vary adaptively during the audio session lifetime.

In our application we used Fixed Playout Delay scheme.

**Fixed Playout Delay**

With the fixed-delay strategy, the receiver attempts to play out each chunk exactly constant time after the chunk is generated. So the packets that arrive after their scheduled playout times are discarded and considered lost.

What is a good choice for *fixed playout delay (q)*? VoIP can support delays up to about 400 msecs, although a more satisfying conversational experience is achieved with smaller values of *q*. On the other hand, if *q* is made much smaller than 400 msecs, then many packets may miss their scheduled playback times due to the network-induced packet jitter. Roughly speaking, if large variations in end-to-end delay are typical, it is preferable to use a large *q*; on the other hand, if delay is small and variations in delay are also small, it is preferable to use a small *q*, perhaps less than 150 msecs.

In our application we use a fixed playout delay of 300 milliseconds to avoid the jitter as far as possible.

## e. Allowing group conferences. (Multicasting)

Multicast is a special feature of UDP protocol that enable programmer to send message to a group of receivers on a specific multicast IP address and port. Use java Multicast functionality to group chat.

## f. Testing.

Testing the applications for various inputs. We test the project using Java unit testing. Junit tool is used to much better results. Unit testing was done by class wise and also sometimes method wise.

## 3. Design of the solution.

### Iteration 1

In this iteration briefly what we did was focus about transmit and receive full-duplex audio over a network between two parties by using UDP protocols Our application must take the peer's or IP and port that used to communicate in this system as a command line argument.

### Iteration 2

In the iteration 2 we implement multi-party conferencing using UDP multicast. Our application should take a multicast group address as a command line argument. We will assume all participants are directly reachable by their IP addresses (no NAT) and, that multicast functionality is available.

To do these iterations and above explained milestones we modularize the tasks to several classes for easiness of implementation. Those are:

1. **VOIP.java**

In this class we used to use two separate threads for each end systems, One for To capture/record Audio and send the byte stream. And other is to To receive and Play the byte stream. Our application must take the peer's IP and listening port of the other end system. These values are passed as a command line argument.

*private void get_and_play_audioStreme(InetAddress mcast, int port)**;*** This Method get the audio packet from the corresponding port and play the audio.

*private void capture_and_send_audioStreme(InetAddress host, int port);* this Method is used to capture audio and send over the network.

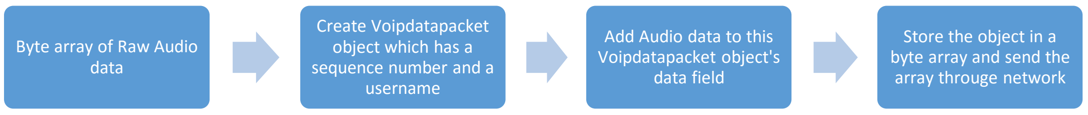
*public VoipDataPacket deserializeVoipPacket(DatagramPacket packet);* Is used to desterilize a serialized byte stream. In Milestones we discussed how serialization happens.

*public byte[] serializeVoipPacket(VoipDataPacket voipDataPacket)*; Method is used to serialize the data packets before sending.

This VOIP class uses other three classes' functionalities to the overall task.

2. **Voipdatapacket.java**

This class is particularly used to serialization and deserialization.


Byte array of Raw Audio data
Create Voipdatapacket object which has a sequence number and a username
Add Audio data to this Voipdatapacket object's data field
Store the object in a byte array and send the array througe network

These are the attributes in this class's object.

*private int sequenceNumber;* // used to save the sequence of the packet flow
*private byte[] data;* // to store data
*private String user;* // to enable multiple users talk at same time

This Class has several methods to help serialization and deserialization. Apart from that there is also a method called 'compareTo' to help sorting packets.

*public int getSequenceNumber()* ; // to get the sequence number of a corresponding object of this class

*public void setSequenceNumber(int sequenceNumber) ;* // to set the sequence number of an object of this class before storing

*public byte[] getData() ;*// to get data from an object of this class

*public void setData(byte[] data)*;// to save data to an object of this class

*public String getUser() ;* //to get user info from an object of this class

*public void setUser(String ip) ;* //to save user info to an object of this class

*public int compareTo(VoipDataPacket o);* //to compare sequence number of this/ current object and another object before sorting or reordering

3. **PacketHandler.java**

In this class we did was:

1. Handle packets and store them in an array list. - *private static synchronized int sizeOferrorDetails()*
2. Implement sorting functions for the array list. - *public static void sort()*
3. Get Data as array list. - *public static ArrayList<VoipDataPacket> getData()*
4. Find the size of the array list (buffer or queue). - *private static synchronized int sizeOfQueue()*
5. Calculate the loss of packets. - *public static void setTimer()*
6. Return the size of the error detail of a specific user. - *public static synchronized ErrorDetails getDetails(String user)*
7. Return the keyset of the array list/ queue. - *private static synchronized Set<String> getKeyset_Queue()*
8. To clear the array list. - *public static void clear()*
9. Add Error Details. - *public static synchronized void putDetails(String user ,ErrorDetails details)*
10. Creating new array lists for new users joining the chat. - *public static synchronized void put(VoipDataPacket packet, String user)*
11. To handle the behavior of the packet and error control - *public static void hanldePacket(VoipDataPacket dataPacket)*

To do these tasks we used the help of ErrorDetail.java class.

4. **ErrorDetails.java**

In this class what we did was implement helper functions for the PacketHandler.java class and store details of error of one user in an array list.

## 4. Performance metrics.

To calculate the loss we used the function:

$$\textbf{Lost Packet as a percentage} = \frac{(\text{No of packet send} - \text{No of received packets})}{(\text{No of packets sent})} \times \mathbf{100\%}$$

But in our application No of received packets are equal to the size of the filled array list in 400 milliseconds.

$$\textbf{Lost Packet as a percentage} = \frac{(\text{No of packet send} - (\text{Size of the fully filled array list} - 1))}{(\text{No of packets sent})} \times \mathbf{100}\ \%$$

When the application is executing packets are receiving at both parties out of order. The status of out of order is returning from the method set timer at PacketHandler class.

Out of order packet is calculated by using the difference of current packets sequence number and previous packets sequence number. If this difference is 1 there is no out of order packet otherwise there are out of order packets. So Number of out of order packets increments.

| **netem configuration** | **Result** |
|---|---|
| Loss 10% | Result 1: Packet lost: 10.086 %<br>Out of order packets: 755<br>Result 2: Packet lost:10.036%<br>Out of order packets: 752 |
| Loss 20% | Result 1: Packet lost: 19.708%<br>Out of order packets: 1313<br>Result 2: Packet lost: 20.308%<br>Out of order packets: 1362 |
| Delay 50ms | Result 1: Packet lost:0.012%<br>Out of order packets:0<br>Result 2: Packet lost:0.0119%<br>Out of order packets:0 |
| Delay 100ms | Result 1: Packet lost:0.011%<br>Out of order packets:0<br>Result 2: Packet lost:0.031%<br>Out of order packets:0 |

## 5. References.

1. Computer Networking A top Down Approach Sixth Edition by James F. Kurose and Keith W. Ross; Chapter 7 Multimedia Networking Section 7.3 Voice-over-IP page 612

2. VoIP help: http://what-when-how.com/voip

3. Java Multicast Example: http://staff.www.itu.se/~peppar/java/multicast_example/

4. Multicast Programming with java: http://lycong.com/programming/multicast-proramming-with-java

5. Java Help:
   http://www.tutorialspoint.com/java
   https://docs.oracle.com/javase/tutorial/