Voice Chat Protocol Design

1 Overview

Building from what we learned during protocol analysis of IRC, our group is designing a new application layer protocol that provides a mechanism for voice chat across TCP/IP networks. The goal of the protocol is to enable real-time voice chat, conferencing, and announcement broadcast across the Internet and private networks that takes advantage of a simple connection model and distributed server architecture to enable tens, hundreds, or even thousands of simultaneous participants in a single conversation.

During our analysis, we observed that many of the IRC concepts could be reused to develop a protocol for voice chat. Some of the concepts that we will reuse as building blocks for our protocol include the server connection model, joining and parting from a channel, and distributing the servers to support many users. The IRC DFAs from our analysis paper also serve as the basis for the voice chat DFAs.

There are also various important distinctions for the voice chat protocol. These include the following:

- The use of a TCP port for control messages and two UDP ports for full-duplex voice transfer.
- Definition of a new control message set that enables voice chat, utilizes XML for message encapsulation, and is easily extendable.
- An objected oriented approach to our message definition that allows for a simple mapping (serialization) between the message set and the objects within our client and server software.
- Definition of a voice packet utilizing a subset of RTP (RFC 3550) for the transfer of voice packets.

2 Protocol Description

The figure below shows the client-server interaction during a voice chat. As shown for client C, three network connections are maintained per client:

- TCP connection for the exchange of control messages and information.
- UDP connection from server to client that contains the conference (sum) of every other participant in the chat or potentially a mix of server announcements and a conference. In this direction, the UDP connection may actually exist between the server and the client's firewall. In this case, it is the responsibility of the client to configure port forwarding for the firewall on the appropriate port.
- UDP connection from client to server that contains the client's voice.

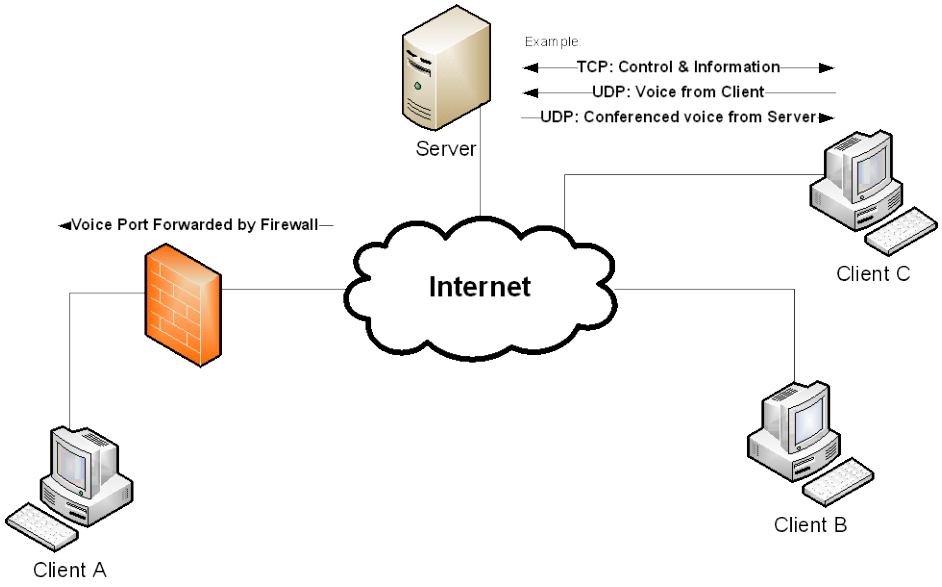


Figure 1: Voice Chat Client-Server Interaction

Note that all communication for Voice Chat is between client and server. The protocol also supports server-to-server communication for extending and distributing conferences, but there is no current support for client-to-client communication[1].

Both the UDP and TCP ports are configurable when the Server is started.

In order to join a chat, the client transmits user commands to the server which are based off of the IRC protocol, such as JOIN and PART. This provides for a simple and friendly mechanism for channel creation and call control.

As mentioned above, the protocol supports distributing conferences across multiple servers. This is conveyed in the figure below. The server implements in software a separate summation / conference circuit per user that sums, on a sample-by-sample basis, each of the incoming streams of a particular channel. The input streams can include the conference sums from other servers on the network. Note that the user may ignore individual streams that are local to its server but cannot separate individual streams from remote servers.

As an example, the figure below shows six clients joined together in a conference call on a channel distributed across 4 servers (A, …, D). Server A implements three conference circuits: Client 1 receives a summation of Client 2 and Server B. Client 2 receives a summation of Client 1 and Server B. Server B receives a summation of Server A and Server C. Server C, which does not have any clients, simply relays Server B's stream to Server D and vice versa.

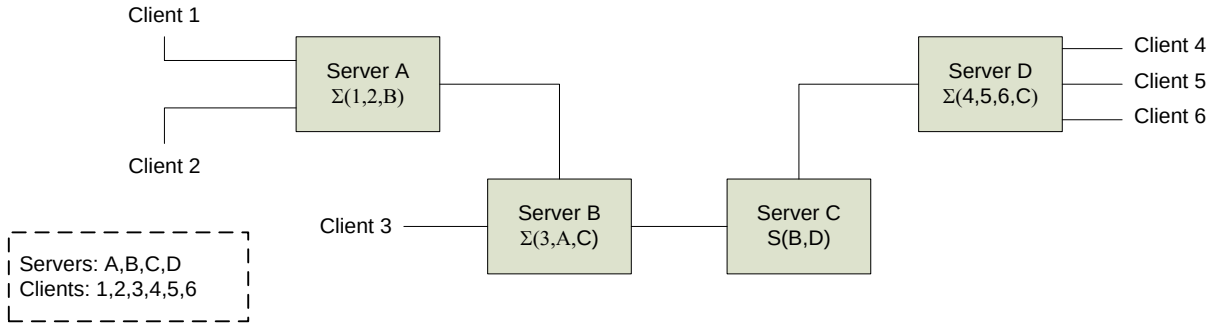


Figure 2: Distributed Voice Chat Server Architecture

3 Message Definition

The message definition for Voice Chat includes both control messages, which are transferred reliably via TCP, and a voice packet, which is encapsulated in an RTP-like header and transported via UDP. The subsections below provide the specifications for the message definitions. All messages will be encoded in a network byte order (big endian).

3.1 XML Control Messages

The protocol utilizes an XML representation for control messages. The figure below depicts the XSD file (graphical) that defines the schema of the XML message set hierarchy. As shown, we have a high level root node "VIRC" and child nodes that break down the various class representations. For example, the graphic below shows the model of Channels (Chan) and Users (User), among others. Under the "Channel" node, there are nodes for channel name (Name), channel description (Desc), and channel mode (Mode). In addition, we show a "User" node which contains sub nodes for Nickname (Nick), IP Address (IP), real name (Name), and password (Pass). Another important thing that is needed at the high level node is the command name (Cmd) because when sending and receiving XML, the server and client need to know the active command or response to provide the correct context for the entire message.

[1] Client-to-client communication could be supported in the future for video chat.

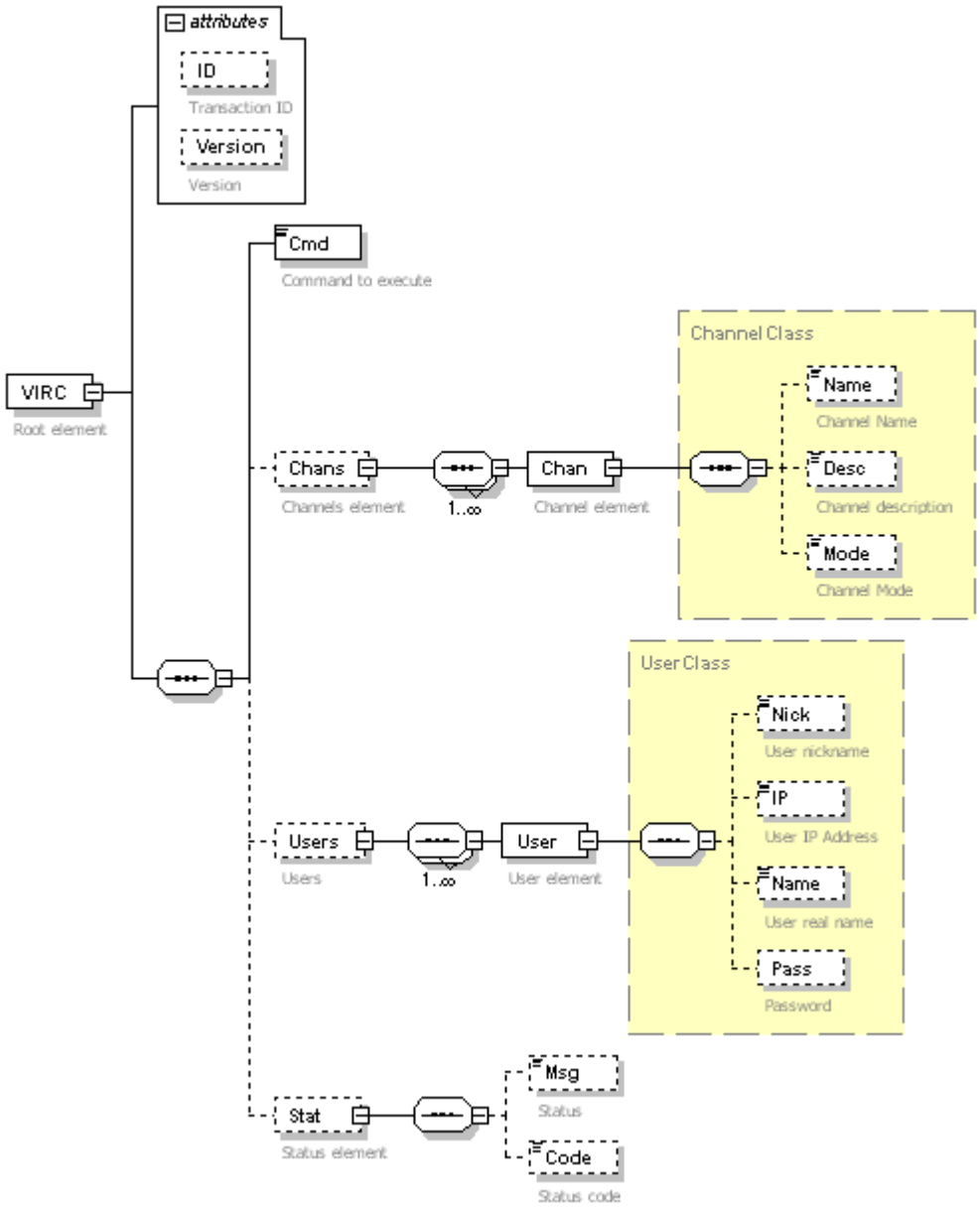


Figure 3: XSD in Graphical Representation

Voice Chat commands are specified below. Note that all commands have an attribute "ID", which is used as a transaction ID. The client will increment this transaction ID with each command sent to the server. The server will process the command and return a response with the same transaction ID so that the client can match the response to the command.

The client will be designed as a Java GUI which supports a simple interactive design in order to support the range of commands listed below. The typical client concept of operation includes the following interactivity:

- The client issues a "TCP Open" event to establish a TCP connection with a remote server.
- The client issues a "Conn" command to connect to a remote server.
- The client issues a "GetChans" command to query a list of existing channels hosted on the server.
- The client issues a "Join" command to enter an existing channel or create a new channel.
- Upon joining the channel, the user will either issue a "GetUsers" command or the server will automatically push the data down to the client such that the GUI displays all users in the conference.
- Upon entering a conference, voice will be enabled such that the user can send voice data to the server, which will handle all processing and distribute the samples accordingly.

- If the user is the first client into the conference, he is automatically promoted to an operator privilege and may issue a "Kick" command to remove any other clients, a "Ban" command to remove and permanently bar another client from the channel, and a "NewDesc" command to change the title and/or description of a channel.
- Any user at any time while in conference can issue a "Mute" command to no longer hear what another client is saying in a certain channel. This is a local event only.
- Any time during the conference, the user may issue a "Part" command to exit the conference.
- When the user wishes to disconnect from the server, he initiates a "Disconn" command.
- The client may close all sockets with the "TCP Close" event.

3.1.1 Conn

Client sends this message to connect to a server.

Precondition: Client must be in a disconnected state.

Server Response: Default command response with '0' for success, '1' for failure, and an optional message to describe the result. *The response will include the SSRC which will be used by the server to determine who to send the voice packets to.* The response will be a '1' if, for example, the nickname sent up is the same as one already in use.

Post Condition: Client transitions into a connected state.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Conn</Cmd>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                        <IP><optional></IP>
                        <Name>Bill Shaya</Name>
                        <Pass>Password</Pass>
                </User>
        </Users>
</VIRC>
```

3.1.2 Disconn

Client sends this message to disconnect from a server.

Precondition: Client must be in a connected state.

Server Response: Default command response with '0' for success, '1' for failure, and an optional message to describe the result.

Post Condition: Client transitions into a disconnected state.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Disconn</Cmd>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.3 Join

Client sends this message to join a voice chat channel.

Precondition: Client must be in a connected state.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result. The server will send a '1' if, for example, the client is attempting to join a channel they are banned from.

Post Condition: Client receives streaming voice from server as long as at least one other client is participating in channel or server is streaming an announcement. Client can begin streaming voice to server immediately after receiving a successful response.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Join</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.4 Part

Client sends this message to a server to leave a voice chat channel.

Precondition: Client must be in a connected state and a participant in a voice chat channel.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result.

Post Condition: Client is connected but no longer a participant in a voice chat channel.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Part</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.5 Kick

Client with operator privilege sends this message to a server to kick a participant out of a channel.

Precondition: Client must be in a connected state and an operator in a voice chat channel, with another participant inside.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result. The server will also send a message to the person being kicked, to knock their client off of the channel.

Post Condition: Client remains connected and operator in the voice chat channel, and the participant remains connected but is no longer a participant in the voice chat channel.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Kick</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.6 Ban

Client with operator privilege sends this message to a server to ban a participant/client from a channel.

Precondition: Client must be in a connected state and an operator in a voice chat channel, with another client who may or may not be a participant inside.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result. The server will also send a message to the person being banned, to knock their client off of the channel forever.

Post Condition: Client remains connected and operator in the voice chat channel, and the client is kicked from the channel if they were a participant and are unable to join that voice chat channel again regardless.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Ban</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.7 Mute

Client with operator privilege sends this message to a server to mute a participant of a channel.

Precondition: Client must be in a connected state in a voice chat channel, with another participant inside. This participant either can currently deliver voice or cannot.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result. The server will also send a message to the person being muted, so knock their client off of the channel.

Post Condition: Client remains connected in the voice chat channel, and the participant remains connected and a participant in the voice chat channel, but their voice is no longer heard by the first client if they weren't muted and are heard again if they were muted (it toggles).

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>Mute</Cmd>
```

```
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
        </Users>
</VIRC>
```

3.1.8 NewDesc

Client with operator privilege sends this message to a server to change the title and/or description of a voice chat channel.

Precondition: Client must be in a connected state and an operator in a voice chat channel.

Server Response: Default Command Response with '0' for success, '1' for failure, and an optional message to describe the result.

Post Condition: Client remains connected and operator in the voice chat channel, and the name and/or description of the voice chat channel are changed.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>NewDesc</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                        <Desc>This is the place to learn C#</Desc>

                </Chan>
        </Chans>
</VIRC>
```

3.1.9 GetChans

Client sends this message to a server to obtain a list of voice chat channels available.

Precondition: Client must be in a connected state.

Server Response: See below.

Post Condition: Client remains connected and receives a list of channels.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>GetChans</Cmd>
</VIRC>
```

Response:

```
<VIRC ID="0" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<Cmd>GetChans</Cmd>
        <Chans>
                <Chan>
                        <Name>C# Development</Name>
                        <Desc>This is the place to learn C#</Desc>
                </Chan>
                <Chan>
```

```
                    <Name>Java Development</Name>
                    <Desc>This is the place to learn Java</Desc>
                </Chan>
        </Chans>
        <Stat>
                <Msg>Some optional text message here</Msg>
                <Code>0</Code>
        </Stat>
</VIRC>
```

3.1.10 GetUsers

Client sends this message to a server to obtain a list of users in each channel.

Precondition: Client must be in a connected state.

Server Response: See below.

Post Condition: Client remains connected and receives a list of voice chat users for a given channel.

```
<VIRC ID="0" Version="1.1" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>GetUsers</Cmd>
        <Chans>
                <Chan>
                        <Name>Java Development</Name>
                </Chan>
        </Chans>
</VIRC>
```

Response:

```
<VIRC ID="0" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd>GetUsers</Cmd>
        <Chans>
                <Chan>
                        <Name>Java Development</Name>
                </Chan>
        </Chans>
        <Users>
                <User>
                        <Nick>Bill</Nick>
                </User>
                <User>
                        <Nick>Bob</Nick>
                </User>
        </Users>
        <Stat>
                <Msg>Some optional text message here</Msg>
                <Code>0</Code>
        </Stat>
</VIRC>
```

3.1.11 Default Command Response

This is the server response messages, used as described in the previous ten messages.

Response for success:

```
<VIRC ID="0" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
        <Cmd><echo Command name></Cmd>
        <Stat>
                <Msg>Some optional text message here</Msg>
                <Code>0</Code>
        </Stat>
</VIRC>
```

Response for failure:

```
<VIRC ID="0" xsi:schemaLocation="http://www.drexel.edu IRC.xsd" xmlns="http://www.drexel.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Cmd><echo command name></Cmd>
        <Stat>
                <Msg>Some optional text message here</Msg>
                <Code>1</Code>
        </Stat>
</VIRC>
```

3.2 Voice Transport

Once a user has successfully joined a voice channel, the server will begin transmitting voice packets to the user on a pre-defined UDP port. The transmission of voice packets from the server is also conditional on there being at least one other source of voice data in the channel. This could be from another user(s) or an announcement from the server.

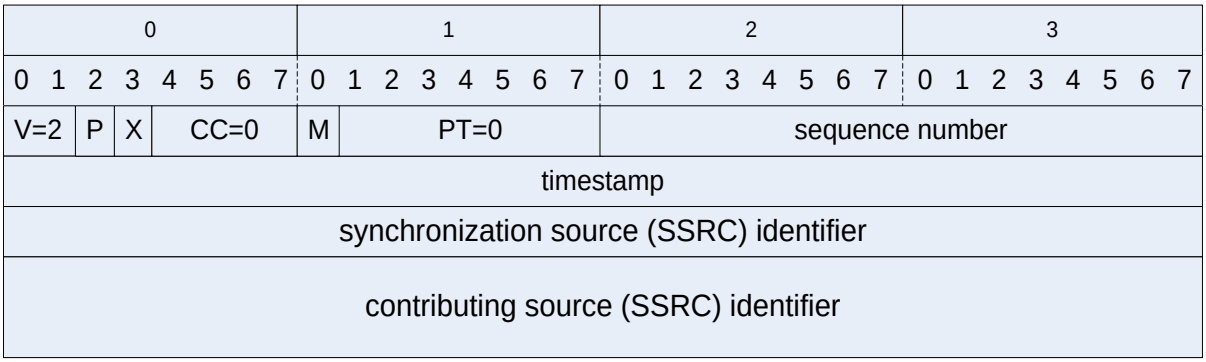
After the channel is joined, the server is ready for voice packets from the client, which it should send to a predefined UDP server port. The server receives the packets into its internal buffer as it prepares to utilize the packet's voice samples in its conferencing / chat algorithm.

Until a successful join occurs, any UDP packets received at either the server or client that is associated with the particular user should be silently discarded (no generation of error messages).

The protocol utilizes the packet format defined for RTP (Real-time Transport Protocol) in RFC 3550. Each RTP packet is encapsulated in UDP/IP. The UDP checksum is not utilized. The overall payload length can be ascertained from the UDP header; however, the packet will have a fixed length for the life of the channel as described below.

RTP provides a well defined and extensible payload format for transferring voice across a TCP/IP network. Common debugging tools such as Wireshark provide native capabilities for analyzing and tracing RTP PDUs. For this initial implementation, the server and client will both implement a simplified jitter buffer and the only associated control of the voice payloads will come from the VIRC message control set[2].

The format of the voice payload is provided in the figure below:



[2] RTCP is not considered for this release

Figure 4: Voice (RTP) Payload Format

Each field of the voice payload for the initial implementation is defined below. The reader should consult RFC 3550 for additional information.

V: version = 2

P: padding = 0, not used. The packets will be fixed size and not require padding, as described further.

X: extension = 0, not used. Extension headers will not be used.

CC: contributing source identifiers = 0. This could be used in the future to define additional sources for the conference data. However, this initial release will not utilize the field.

M: marker = 0 or 1. The client and server should set this bit when the particular packet is known to be the last packet being sent in a continuous stream of voice. For example, the server should set the bit when the packet is the last segment of audio for an announcement.

PT: payload type = PCMU. The PCMU value is defined in RFC 3551 as 0. PCMU is standard North American telephony quality voice, which has been traditionally carried by the Public Switched Telephone Network. The data is sampled nominally at 8 Khz and each voice sample is represented in a single byte. The description provided below is from RFC 3551[3]:

> "PCMA and PCMU are specified in ITU-T Recommendation G.711. Audio data is encoded as eight bits per sample, after logarithmic scaling. PCMU denotes mu-law scaling, PCMA A-law scaling. A detailed description is given by Jayant and Noll [15]. Each G.711 octet SHALL be octet-aligned in an RTP packet. The sign bit of each G.711 octet SHALL correspond to the most significant bit of the octet in the RTP packet (i.e., assuming the G.711 samples are handled as octets on the host machine, the sign bit SHALL be the most significant bit of the octet as defined by the host machine format). The 56 kb/s and 48 kb/s modes of G.711 are not applicable to RTP, since PCMA and PCMU MUST always be transmitted as 8-bit samples."

Sequence Number: This field is implemented as described in RFC 3550. Any "reasonable" approach to random number generation may be used for this initial implementation.

Timestamp: This field is implemented as described by RFC 3550, but limited in the following ways. Each sender (server or client) shall increment the time stamp value by one for each byte that has been sent. Each byte nominally represents 125 µs in time. All packets during the life of a channel contain a fixed size number of samples, as specified via the VIRC control message set. The time stamp should be initialized by a random number generator similar to what is used to generate the sequence number. There will be no effort made in synchronizing clocks.

For example, suppose the channel is configured for a 30 ms payload. This would correspond to 30 * 8 = 240 samples. If the initial timestamp from the random number generator is 1000, then the first timestamp will be set to 1000. The next packet will be transmitted with a timestamp set to 1240 (decimal). If for some reason 240 samples are not available for sending (e.g., microphone muted), then the sender shall either append all 0xff's to complete the payload or choose not to send the packet at all (effectively drop it and potentially truncate the sound).

SSRC: This field is implemented as described in RFC 3550. Any "reasonable" approach to random number generation may be used for this initial implementation.

CSRC List: This field will not be utilized in the initial implementation. However, since the server is basically a mixer as described in RFC 3550, this is potentially a valuable field for future use to identify contributing servers in a distributed conference. Since this field is not utilized, SSRC will be the last field in the RTP payload.

---

3 PCMA is the analogous standard developed for European Public Switched Telephone Networks.

4 Deterministic Finite Automata (DFA)

The figures below depict the DFAs for the voice chat protocol. They are basically reused from our group's work in analyzing IRC since they set the model for our protocol.

4.1 Client to Server DFA

The figure below depicts a compact view of the client-to-server DFA. The client initiated messages are shown in lower case, and the server responses are shown in upper case. In general, whenever an improper command is sent to a server, the server will reply with an error and not proceed to change the client state. If a message is correctly formed and is sent within the proper context, then the server will move the client to the next state and reply with an informational command response.

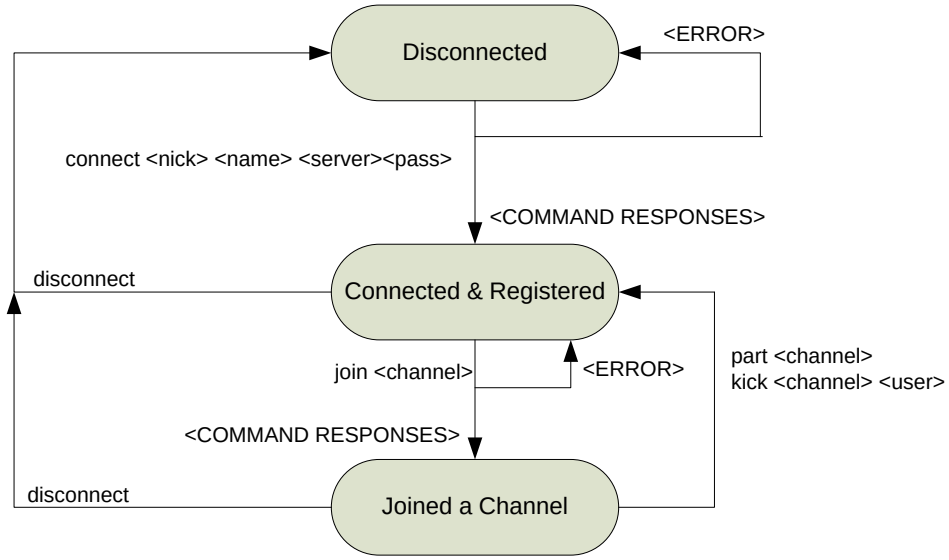


Figure 5: Client to Server DFA

During the life of a connection, the server application must retain the client's state and also share this information with other servers linked on the voice chat network.

For the case of the JOIN message, the channel will be created if it did not previously exist. The user may leave a channel voluntarily by sending a PART message or be kicked off a channel by an operator via a KICK message. There are other messages the operators can use to control the channels.

4.2 Channel DFA

The figure below depicts the DFA for a channel. By default, channels do not exist within a VIRC server until they come into existence upon the issuance of a JOIN message by a user. A channel will change its application state upon operators issuing commands and additional users joining the channel.

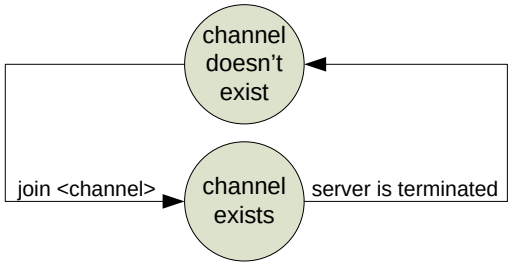


Figure 6: Channel DFA

4.3 Server DFA

The figure below depicts the DFA utilized for the connection of a remote server. The previously defined DFAs are also relevant during this process since a remote server relays the messages from its clients. Once linked, a remote server basically represents all the clients on the other side of the network connection.

The SERVER message is exchanged by the servers trying to connect, and it is also relayed across the server network in order to advertise its information to the entire network of servers that a new server is connecting to the network. If a duplicate SERVER message is received on a new connection, then the connection associated with that message is closed in order to preserve the spanning tree / acyclic nature of the network.

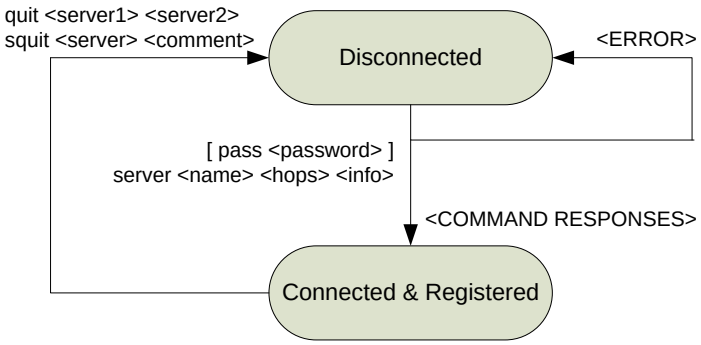


Figure 7: Server to Server DFA

5 Extensibility

This protocol uses an XML message structure between the client and server with regards to control information. XML is extremely extensible due to the fact that the designer defines all elements of the schema. Being an open standard, it is geared towards information sharing across networks between computer platforms. The simple syntax of an XML message makes information sharing effortless between applications since no message conversion is necessary.

In addition, XML easily supports future versions of our protocol. For example, to extend the capability of a message, we can easily append additional nodes to an existing message command with little worry about versioning conflicts, as long as the parsing is accomplished at the node level. If older software receives a message in the extended protocol format, the legacy application will parse the message as normal and disregard the enhanced information contained within the extra nodes, while a newer application will be able to understand the added information, and correctly handle the new functionality. Furthermore, completely new commands can be made and the older versions would just ignore them, as when they parse the “Cmd” tag, it would not give them something they are used to, and it would be disregarded. So the XML allows for both new control messages to be created and new data to be added to existing ones, with no impact on the older versions.

As specified in the Voice Transport section above, this initial specification utilizes G.711 as the payload type. Additional compression schemes can be supported in the future through modification of the XML message set. Other related extensions include the support for the default frame size, RTCP-like metrics, and support for silence suppression and comfort noise injection.

Extending the protocol to support both text and video conferencing is also a natural evolution of the protocol. Text chat can be added by defining a new message command to specify a client’s text message, which would be delivered within the existing TCP connection. Video conferencing would require peer-to-peer transfer of real-time video frames. The connection set up would be facilitated through the use of a client-to-server message exchange. In order for video chat to operate properly, issues such as lip synchronization and exchange of real-time metrics would be required. This is beyond the scope of this initial protocol specification.

6 Performance

An XML protocol structure is being used to convey control commands and responses between the client and server applications. All control commands and responses will be transmitted and received through TCP sockets, which provide a reliable data transfer interface. Since control messages will impact the server's state, TCP is preferred over UDP with regards to the reliability and performance in the quality of service it provides. Although XML is verbose by nature, the structure and frequency of control messages is constrained such that performance is not impacted with regards to the user experience.

A significant real-time processing constraint is imposed on the server due to its requirement to perform a mixer and full-duplex compression / decompression of audio for each client joined in a channel. As described in previous sections, this protocol specifies G.711 for the voice compression. Although this compression scheme imposes a significant load on the TCP/IP pipe (64 kbits/s + overhead per client, per direction), the requirement on the server to convert the data to linear is relatively light. If future versions of the protocol specify additional compression schemes (e.g., deeper compression or higher quality audio), a dedicated daughter board comprising signal processors may be required. However, the forecast by Intel and AMD of a significant number of parallel cores within a single CPU package may alleviate the need for a daughter board and facilitate greater compression rates or support of high quality audio through the use of on-chip parallel processing.

7 Security

When a client desires to connect to a server, the client's IP address is transmitted in the registration command. Currently, the initial version of the protocol uses the IP address for messaging purposes only. The server will maintain an association list between users and channels. The IP addresses are important in the matter of transmitting voice data to the intended target. However, future versions may use IP information for various audit trails. In addition, all control messages are currently transmitted in plain text across the network, however, future versions could easily be extended to provide a cipher text transport channel.

VIRC incorporates a native layer of security with regards to connection registration. The VIRC architecture forms a spanning tree of servers connected to servers and clients connecting to the aforementioned servers. The registration of the connection must be made regardless of whether the connection is server to server or client to server. The VIRC protocol defines a "Conn" command, which contains a password in its message in order to gain access to the desired resource. It is important to note that the password command is not required prior to registration; however, it is recommended for use in order to ensure a minimal level of protection.

Voice Internet Relay Chat has its control messages built on the Transport Control Protocol (TCP) and thus by default, all traffic is transmitted in plain text, as mentioned above. The protocol can be enhanced to use Transport Layer Security (TLS) over TCP in order to provide security over the network. TLS/SSL is a means of ensuring authentication and message privacy over the Internet by means of cryptography algorithms such as RSA, DES, or AES.

The VIRC architecture also incorporates security in terms of user roles, specifically operators. Channel operators are privileged clients (the first client to join a channel) who may initiate server specific tasks such as server connection and disconnection, as well as client disconnection if necessary. Channel operators are owners of a given chat channel who may administer various aspects of the chat room, such as kicking and banning clients from their channel and changing the title and/or description of a channel. Individual users can also protect themselves by muting their own microphones or the microphones of other clients in their room (local change only).

8 Implementation Plan

- A subset of the protocol will be implemented to demonstrate basic voice connectivity and voice transfer (e.g., two party chat). We do not plan to implement the functionality required for multiple, distributed servers.
- The client will be implemented in Java, utilize a GUI, and be tested on at least Microsoft Windows.
- The server will be implemented in C/C++ and target a Linux host.
- An XML package / library will be utilized for parsing the XML messages on the client. A very basic XML parser may be utilized on the server.
- The voice packets will support only G.711 (64 kbit/s) telephony grade voice.

9 Risks

- It is our hope to be able to demonstrate live voice with a microphone and speaker; however, the use of voice files will be a fallback position.
- The voice jitter buffer will be simple and may not have time to support complex functions like packet re-ordering or dynamic buffer sizing.
- Achieving real-time performance out of a generic, desktop Linux system may be difficult. It may be necessary to insert long delays in multiple areas in order to demonstrate continuous voice. Another option is to demonstrate the voice chat system utilizing an embedded Linux system. This will be explored during development.

10 Implementation Modifications

- Added “Version” attribute to XML command messages
- Removed “Server” XML node, Server to Server communication was not implemented.
- Added “Pass” node to be a child of the “User” node
- Modified server responses such that the command initiated is echoed in the response
- Introduction of Altova XMLSpy 2009 for client XML processing
- Introduction of Java Media Framework (JMF) API (version 2.1.1e) for client voice processing
- Added “TCP Open” and “TCP Close” buttons to separate “Conn” and “Disconn” messages from socket operations
- Modification of “GetChans”, “Conn”, and “Default Command Response”
- Tweaked “Mute” so that any client can mute any other client on their local machine, not just administrators
- Tweaked Kick and Ban Control messages to show that a message is being sent down to the user who was kicked or banned

11 Implementation Performance Implications

The current design of the application proves to display no significant performance issues. As mentioned in the former sections, TCP is used for the transfer of control messages, while UDP is used for the transfer of voice. From the client perspective, all communications is handled in its own thread, therefore increasing performance and usability of the user experience. Even though TCP introduces additional overhead for the reliability it provides, control messages are sent infrequently, and thus do not impact performance. All voice communication is provided over UDP, which also does not produce any significant performance issues. To ensure a high quality of service, the protocol design has a constraint that a user may be connected to at most one channel at a given time.

We tested with up to four simultaneous connections, which was more than suitable for testing the protocol itself. We did not obtain any metrics on the number of clients that could be supported based on a particular processor. The memory load on the server per client is relatively light (less than 9Kbytes per client).

12 Bibliography

Oikarinen, J., and D. Reed. "RFC 1459 - Internet Relay Chat Protocol." RFC 1459 (rfc1459) - Internet Relay Chat Protocol. May 1993. Internet RFC/STD/FYI/BCP Archives. 18 Feb. 2009 <http://www.faqs.org/rfcs/rfc1459.html>.

Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson. "RFC 3550 – RTP: A Transport Protocol for Real-Time Applications." RFC 3559 (rfc3550) – RTP: A Transport Protocol for Real-Time Applications. July 2003. Internet RFC/STD/FYI/BCP Archives. 18 Feb. 2009 <http://www.faqs.org/rfcs/rfc1459.html>.

13 Appendix A: Full Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2008 sp1 (http://www.altova.com) by L-3 Communications (L-3 Communications) -->
<xs:schema xmlns="http://www.drexel.edu" xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.drexel.edu" elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:element name="VIRC">
		<xs:annotation>
			<xs:documentation>Root element</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:sequence>
				<xs:element name="Cmd">
					<xs:annotation>
						<xs:documentation>Command to execute</xs:documentation>
					</xs:annotation>
					<xs:simpleType>
						<xs:restriction base="xs:string">
							<xs:enumeration value="Conn"/>
							<xs:enumeration value="Disconn"/>
							<xs:enumeration value="Join"/>
							<xs:enumeration value="Part"/>
							<xs:enumeration value="Kick"/>
							<xs:enumeration value="Ban"/>
							<xs:enumeration value="Mute"/>
							<xs:enumeration value="NewDesc"/>
							<xs:enumeration value="GetChans"/>
							<xs:enumeration value="GetUsers"/>
						</xs:restriction>
					</xs:simpleType>
				</xs:element>
				<xs:element name="Chans" minOccurs="0">
					<xs:annotation>
						<xs:documentation>Channels element</xs:documentation>
					</xs:annotation>
					<xs:complexType>
						<xs:sequence maxOccurs="unbounded">
							<xs:element name="Chan" type="ChannelClass">
								<xs:annotation>
									<xs:documentation>Channel
element</xs:documentation>
								</xs:annotation>
							</xs:element>
						</xs:sequence>
					</xs:complexType>
				</xs:element>
				<xs:element name="Users" minOccurs="0">
					<xs:annotation>
						<xs:documentation>Users element</xs:documentation>
					</xs:annotation>
					<xs:complexType>
						<xs:sequence maxOccurs="unbounded">
							<xs:element name="User" type="UserClass">
								<xs:annotation>
									<xs:documentation>User
element</xs:documentation>
								</xs:annotation>
```

```
                                                </xs:element>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="Stat" minOccurs="0">
                                <xs:annotation>
                                        <xs:documentation>Status element</xs:documentation>
                                </xs:annotation>
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element name="Msg" type="xs:string" minOccurs="0">
                                                        <xs:annotation>
                                                                <xs:documentation>Status
message</xs:documentation>
                                                        </xs:annotation>
                                                </xs:element>
                                                <xs:element name="Code" type="xs:unsignedInt"
minOccurs="0">
                                                        <xs:annotation>
                                                                <xs:documentation>Status
code</xs:documentation>
                                                        </xs:annotation>
                                                </xs:element>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                </xs:sequence>
                <xs:attribute name="ID" type="xs:unsignedInt">
                        <xs:annotation>
                                <xs:documentation>Transaction ID</xs:documentation>
                        </xs:annotation>
                </xs:attribute>
                <xs:attribute name="Version">
                        <xs:annotation>
                                <xs:documentation>Version</xs:documentation>
                        </xs:annotation>
                </xs:attribute>
        </xs:complexType>
</xs:element>
<xs:complexType name="ChannelClass">
        <xs:sequence>
                <xs:element name="Name" minOccurs="0">
                        <xs:annotation>
                                <xs:documentation>Channel Name</xs:documentation>
                        </xs:annotation>
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:element>
                <xs:element name="Desc" type="xs:string" minOccurs="0">
                        <xs:annotation>
                                <xs:documentation>Channel description</xs:documentation>
                        </xs:annotation>
                </xs:element>
                <xs:element name="Mode" minOccurs="0">
                        <xs:annotation>
                                <xs:documentation>Channel Mode</xs:documentation>
```

```
                        </xs:annotation>
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="public"/>
                                <xs:enumeration value="private"/>
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="UserClass">
                <xs:sequence>
                    <xs:element name="Nick" minOccurs="0">
                        <xs:annotation>
                            <xs:documentation>User nickname</xs:documentation>
                        </xs:annotation>
                        <xs:simpleType>
                            <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                    </xs:element>
                    <xs:element name="IP" type="xs:string" minOccurs="0">
                        <xs:annotation>
                            <xs:documentation>User IP Address</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Name" type="xs:string" minOccurs="0">
                        <xs:annotation>
                            <xs:documentation>User real name</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Pass" minOccurs="0">
                        <xs:annotation>
                            <xs:documentation>User password</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
</xs:schema>
```

14 Appendix B: Example Client Graphical User Interface