***Voice Chat Protocol Design***

# Overview

Building from what we learned during protocol analysis of IRC, our group wishes to design and implement 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 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 would like to 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. We also plan to start with the IRC DFAs from the analysis paper in developing the voice chat DFAs.

There are also various important distinctions for our new 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 targets 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 the RTP definition (RFC 3550) for the transfer of voice packets.

# 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:

1. TCP connection for the exchange of control messages and information.
2. 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 live 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.
3. UDP connection from client to server that contains the client’s voice.



Figure : 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]](#footnote-2).

In order to join a chat, the client transmits user commands to the server which are based in theory 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 user streams that the user has chosen to not ignore. The input streams include the conferences 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[[2]](#footnote-3).

As an example, the figure below shows six clients joined together in a conference call on a created 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 Clients 1 and 2. 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

# Message Definition

The message definition for Voice Chat includes both control messages that 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 definition.

## XML Control Messages

The protocol utilizes an XML representation via TCP/IP 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 break down the various class representations. For example, the graphic below shows the model of Channels (Chan) and Users (User). Under the "Channel" node, we have 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), I.P. Address (IP), and real name (Name). Another important thing that is needed at the high level node is the command name (Cmd) because when sending and receiving XML, the software needs to know exactly what command or response the client is initiating. Based on that, the protocol will extract which nodes under the tree to parse or create.

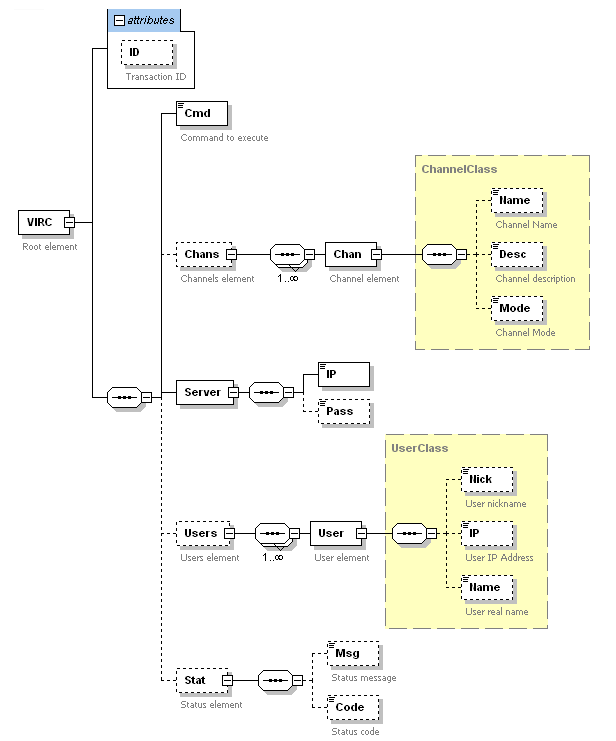


Figure : XSD in Graphical Representation

Commands initiated by the client are displayed 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 “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, a “Mute” command to no longer hear what another client is saying in a certain channel, and a “NewDesc” command to change the title and/or description of a channel.
* 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.

### 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.

**Post Condition:**  Client transitions into a connected state.

<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>Conn</Cmd>

<Server>

<Pass>Optional Password</Pass>

</Server>

<Users>

<User>

<Nick>Bill</Nick>

<IP>192.168.1.1</IP>

<Name>Bill Shaya</Name>

</User>

</Users>

</VIRC>

### 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" 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>

### 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.

**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" 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>

### 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" 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>

### 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.

**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" 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>

### 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.

**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" 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>

### 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 and an operator 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.

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

<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>Mute</Cmd>

<Chans>

<Chan>

<Name>C# Development</Name>

</Chan>

</Chans>

<Users>

<User>

<Nick>Bill</Nick>

</User>

</Users>

</VIRC>

### 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" 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>

### 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" 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>

<Mode>public</Mode>

</Chan>

<Chan>

<Name>Java Development</Name>

<Desc>This is the place to learn Java</Desc>

<Mode>public</Mode>

</Chan>

</Chans>

<Stat>

<Msg>Some optional text message here</Msg>

<Code>0</Code>

</Stat>

</VIRC>

### 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 channels and the users in each.

<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>

</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>

### 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>Join</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>Join</Cmd>

<Stat>

<Msg>Some optional text message here</Msg>

<Code>1</Code>

</Stat>

</VIRC>

## 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[[3]](#footnote-4).

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



Figure : 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[[4]](#footnote-5):

“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.

# Deterministic Finite Automata (DFA)

-This is still from analysis paper. It will be updated once message set is reviewed

As depicted in the figure below, each server in an IRC network is connected via a spanning tree to the other servers. From the perspective of the server, there are three separate sets of states that must be maintained: clients, channels, and other servers. Therefore, each server must implement three distinct DFAs. In addition, each server must share their client and channel states with other servers in the network and automatically resolve conflicts.



Figure 5: Distributed IRC Servers and Clients

The sections below provide information on each of the three aforementioned DFAs (state machines). The DFAs are crafted by following the RFC 1459 specification. The changes and updates specified in the 281X series of RFCs are not included.

## 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 initiated messages are shown in upper case. Only the mandatory parameters are shown in the client messages, and for the sake of clarity in the figure, not all server replies are shown[[5]](#footnote-6). 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. The error messages are specified in section 6.1 of RFC 1459. 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 6: 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 IRC network. The client application need not retain the state information when the client is human, as opposed to an automated client.

The RFC assumes that the underlying protocol is TCP/IP, but it does not preclude another lower layer protocol from being used. In the case of TCP/IP, the connection is initiated by the client with a 3-way handshake, which isn’t shown in the figure. Within the IRC protocol, the client initiates an IRC connection by sending the PASS, NICK, and USER messages. The server responds with replies, which are one or more ASCII strings that either contain a three digit code or are purely informational in the form of a NOTICE[[6]](#footnote-7). The three digit codes can either specify an informational message or specify an error.

The OPER message is sent by the client to gain special operator privileges. Note that the RFC does not clearly specify when the OPER message may be sent; however, it is implied that the user should at least be registered before the message will be accepted. For the case of the JOIN message, the channel will be created if it did not previously exist (see the section below on Channel DFA for more information). Once the user has joined a channel, the user may change their state by issuing the MODE message. This in effect changes the state of the user, but this is not shown in the figure for the sake of simplicity. 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.

The user has various options to retrieve information from the server through the use of messages like LIST, LINKS, and NAMES, but these are not shown in the DFA since they do not change the state of the user. Of course, the user can also send messages to other users. This is accomplished through the use of PRIVMSG and NOTICE; neither of these change the state of the user.

The “any” state represents all states. The user may issue a QUIT command at any time to disconnect from the server (and leave any joined channels). An operator or server can accomplish the same result by issuing a KILL message.

When a server doesn’t detect activity from a user, it may issue a PING message. The user must quickly reply with a PONG. If the user doesn’t reply quickly, then the server closes the connection and returns the client back to the “disconnected” state. This transition is not shown in the figure.

## Channel DFA

The figure below depicts the DFA for a channel. By default, channels do not exist within an IRC server until they come into existence upon the issuance of a JOIN message by a user. Once a channel is created, it exists until the last user issues a PART message. Channels can also cease to exist upon a split in a network where one side of the split ceases to have users**.** A channel will change its state upon operators issuing mode commands and additional users joining the channel.



Figure 7: Channel DFA

## Server to 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. For example, Figure 3 shows that Server A would relay all the messages between Server B and clients 1 and 2. Inter-server communication also utilizes the same message format and protocol as the clients. However, some additional messages are defined for server-to-server communication.

The PASS command is optional, but RFC 1459 strongly recommends the use of it “in order to give some level of security to the actual connections”. The PASS message is the only message shown in the figure that is used exclusively between directly connected servers.

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.

The QUIT message may be issued when a net split occurs. The message contains two parameters: <server 1> is the server that is still connected, and <server 2> is the server that has become disconnected. In this case, the local server would remove server 2 from its database along with all servers behind server 2. The SQUIT message is issued by a remote server when it wishes to disconnect. Both the SQUIT and QUIT messages are transmitted by the local server to existing remote servers that are still connected in order to advertise the network’s configuration. A network or connection failure between servers has the same effect as an SQUIT message.

The optional RESTART message is not shown in the figure. This is a message that may be issued by an operator to restart a server. Upon restarting, all connections, channels, and users should be reestablished utilizing the DFAs described in this document. A similar message is REHASH, which forces a server to reread its configuration file. It is not specified whether the local server would temporarily disconnect itself during this reconfiguration.



Figure 8: Server to Server DFA

As mentioned in the previous section on extensibility, different vendor’s servers may not interoperate. Through experimentation with actual servers, it appears that some servers may send messages that are not defined in either RFC 1459 or the 281X series of RFCs. For example, the Unreal IRC server sends PROTOCTL and SMO messages when connecting. A log file from the connection of two Unreal IRC servers is provided in Appendix A.

# Extensibility – Bob talk about voice messages

-Version info, handshake negotiation, etc.

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. 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.

# Performance – Bob performance of Voice, etc., QOS

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.

# Security – Bob talk about Encryption

-How do you ensure it is always working correctly?

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 di**s**connection, 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, banning, and muting 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.

# Implementation Plan

* Implement a subset of the protocol to demonstrate basic voice connectivity and 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.
* The voice packets will support only G.711 (64 kbit/s) telephony grade voice.

# 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.
* 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.

# 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="Server">

<xs:complexType>

<xs:sequence>

<xs:element name="IP" type="xs:string"/>

<xs:element name="Pass" type="xs:string" minOccurs="0"/>

</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: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:sequence>

</xs:complexType>

</xs:schema>

\*\*\*Will we need a bibliography?

1. Client-to-client connections could be supported in the future for video chat. [↑](#footnote-ref-2)
2. The user can either listen to all users or none from remote servers. [↑](#footnote-ref-3)
3. RTCP is not considered for this release [↑](#footnote-ref-4)
4. PCMA is the analogous standard developed for European Public Switched Telephone Networks. [↑](#footnote-ref-5)
5. The path from “Disconnected” to “Connected and Registered” shows both the error response and command response. [↑](#footnote-ref-6)
6. The number of supported replies have been extended in the 281X series of RFCs and in practice. [↑](#footnote-ref-7)