Network Protocol Design – Voice Chat

Group 14: Bill Shaya, Jordan Osecki, Bob Cochran

CS544: Computer Networks, Professor Kain

February 11, 2009 – Winter Term 2008-2009

**Voice Chat Network Protocol Design Document**

# Overview

-Description of the service that protocol and server will provide

-Describe basic service and options/conditions available

Building from what we learned during protocol analysis of IRC, our group wishes to design and implement a new protocol that provides a platform 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 provide a mechanism 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 a UDP port for 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 compact voice packet, which may be modeled after RTP, for the transfer of voice packets utilizing UDP. The packet header will be optimized for real-time transfer and consist of concise bit fields to define parameters such as sequence number and payload length.

# 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

1. 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.
2. The voice jitter buffer will be simple, and we may not have time to support complex functions like packet reordering.
3. 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.

# System Architecture

The figure below depicts at a high level the distributed nature of the voice chat system. Each server supports the familiar user commands for joining a chat (e.g., NICK, USER, JOIN, PART, etc.). This provides for a simple and friendly mechanism for channel creation and call control. 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[[1]](#footnote-2).

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 1: Distributed Voice Chat Server Architecture

Considering that the payload for voice chat is digitized voice rather than ASCII strings, the voice packet utilizes well defined bit fields that specify various parameters including voice activity, sequence number, and payload length.

# Messages - XML Representation

-Define how the messages will be delineated

-Define control and data messages

-Show all pieces of the messages including data types, enumerations, etc.

-Probably keep the diagram but move the rest to an appendix

Our protocol proposes using an XML representation via TCP/IP for control messages. Below, in Figure 2, is a sample XSD file (graphical) which defines the schema of the XML hierarchy. As shown, we have a high level root node "VoiceIRC". Child nodes break down into the various class representations. For example, the graphic belows shows the model of "Channel" and "User". Under the "Channel" node, we have nodes for "ChannelName", "ChannelTopic", and "ChannelMode". In addition, we show a "User" node which contains sub nodes for "UserName", "IPAddress", and "UserMode". Another important thing that is needed at the high level node is the "CommandName" because when sending and receiving XML, the software needs to know exactly what command or response the client is initiating, and based on that, the protocol will dictate extract which nodes under the tree to parse or create.

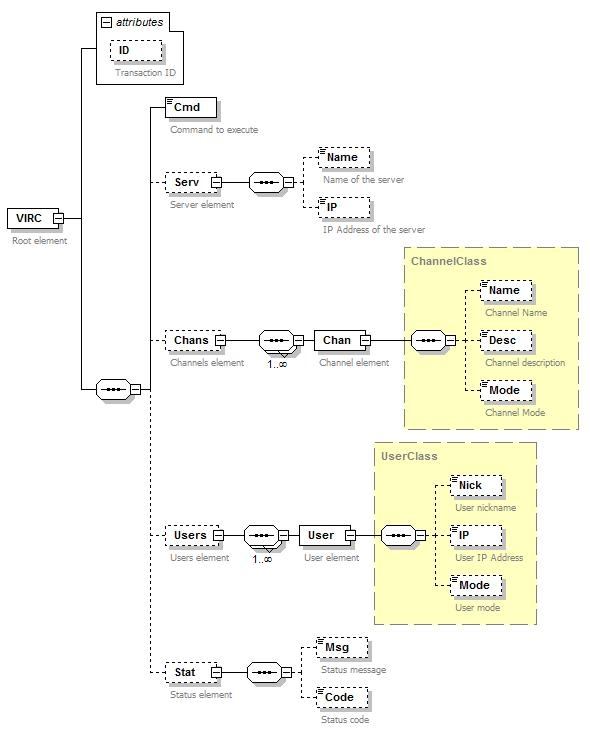


Figure 2: Sample XSD In Graphical Representation

<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="" elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:element name="VoiceIRC">

<xs:annotation>

<xs:documentation>Root element</xs:documentation>

</xs:annotation>

<xs:complexType>

<xs:sequence>

<xs:element name="CommandName">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="JoinChannel"/>

<xs:enumeration value="LeaveChannel"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

<xs:element name="Channel" type="ChannelClass"/>

<xs:element name="User" type="UserClass"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:complexType name="ChannelClass">

<xs:sequence>

<xs:element name="ChannelName">

<xs:simpleType>

<xs:restriction base="xs:string"/>

</xs:simpleType>

</xs:element>

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

<xs:element name="ChannelMode">

<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="UserName">

<xs:simpleType>

<xs:restriction base="xs:string"/>

</xs:simpleType>

</xs:element>

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

<xs:element name="UserMode">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="visible"/>

<xs:enumeration value="invisible"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:schema>

Below is a sample XML message that would be generated if the client invokes a “JoinChannel” command. As you can see, the only relevant information that is needed to join a channel is the “ChannelName” and “UserName”. The client software would create the XML structure and pass it to the server, which would parse the appropriate nodes based on the protocol spec.

<?xml version="1.0" encoding="UTF-8"?>

<VoiceIRC xsi:schemaLocation="" xmlns="" xmlns:xsi="">

<CommandName>JoinChannel</CommandName>

<Channel>

<ChannelName>Software Development</ChannelName>

</Channel>

<User>

<UserName>Bill</UserName>

</User>

</VoiceIRC>

# Deterministic Finite Automata

-Define and show it, at least 4 states, 10 unique messages (only 2 error msgs)

-Describe it’s statefulness

# Extensibility

-Describe how it allows future extensibility

-Version info, handshake negotiation, etc.

# Performance

-Also is supposed to be expressed

# Security

-Describe how it will ensure right people can use it, allow everyone?

-Describe all authentication mechanisms (not all need to be implemented, can be hardcoded)

-How do you ensure it is always working correctly?

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

1. The user can either listen to all users or none from remote servers. [↑](#footnote-ref-2)