



# DELIVERABLE I

## Peer to Peer Chatrooms

### Program Summary

Portable programs that enable the user to run the program without any networking skills or experience. When running the program connects multiple users to chatrooms that are synced with new arrivals.

Jacob Pebworth, Austin Hoppe, Elexa Evans,  
Christopher Hines, Ryan Porterfield  
4444-Chat\_Group-5

# Ventilate

## About Ventilate

Ventilate is a peer to peer chat application written in C++ on the Qt5 platform. Ventilate offers both public and private chat, and keeps a thorough history of public communications. Ventilate is distributed, peer-to-peer so there is no central server that is susceptible to Denial of Service attacks. The


distributed nature of the application also allows us to keep multiple backups of chat records that can be authenticated against each other to ensure integrity of data. Ventilate uses a simple central server for initial creation of user accounts, and to keep a list of connected peers.

Our team of five is developing a chat room based program, borrowing some elements of the design of IRC clients, many examples of which can be commonly found. Our program is designed to be compatible with a variety of different platforms, but we will specifically be targeting the Mac OS X, Linux and Windows environments for compatibility when compiling binary executable files from our source code. Our primary development languages will be C and C++, especially in light of our group having chosen Qt for our preferred graphics toolset. This is a cross-platform and well-documented library designed for development with C++, which suits our needs appropriately. This is especially important because one of our primary goals is excellent cross-platform compatibility, which can be best facilitated by our choosing a suitable library and programming language ahead of time.

Our primary risks will likely involve communications and equipment problems, along with personnel issues like a lack of familiarity with some of the technologies which we use, and running afoul of network policies with the Computer Science and Engineering (CSE) servers run by the University of North Texas (UNT). One of the technical issues which we may find ourselves confronting is trouble with the complexity of the connections density with many users connected at once. This is a problem because the total number of bidirectional edges for a complete graph (which our system closely models in

theory) is  $\frac{n(n-1)}{2}$  (where n represents the number of users connected),

which has a Big-O complexity of  $O(n^2)$ . Our choice of networking protocol is



Ventilate has the meaning that everyone is familiar with, "to provide (a room, mine, etc.) with fresh air in place of air that has been used or contaminated." [1] I like this because I don't like any of the current major chat programs, and I think the group chat scene needs some fresh air. However, ventilate also means "to give utterance or expression to (an opinion, complaint, etc.)" [1] which seems like the perfect definition for a chat client.

[1]: <http://dictionary.reference.com/browse/ventilate>

part of our solution to this; the number of other users for any given user to query any time, for example, a log file needs to be updated, is simply  $(n-1)$ , which has a Big-O complexity instead of  $O(n)$ . If we were to have chosen instead a completely client/server-based model, the server would have become increasingly laden with overly-complex sets of network links about which to remain updated, which would have negatively affected the performance of our application when scaling to especially large numbers of users over extended periods of time.

Our focus regarding communication protocol is on peer-to-peer networking with connected clients. This presents a practical benefit in that very little interaction is required with a complex central server. The program will still require some measure of interaction with a basic server, mainly for Internet Protocol (IP) address translation and in order to authenticate clients, but this can be served by a simpler program running on a system with a known IP address. This is similar in practice to tracker servers from the BitTorrent protocol, and indeed serves the similar function of facilitating connections between active peers. Two of our main goals are security and distributed networking, which informs our decision to keep the server's functionality as minimal as possible. Whenever the program is started and a user is successfully authenticated, they are automatically entered into a central, "skeleton" chat room which displays connected users. Practically, this will involve querying the server for active users and their IP addresses, and then the client program establishing connections to each peer. Activity thereafter is restricted to peer-based communication, and this is notably the case both for establishing chat rooms and for maintaining records of chat activity. One of the main activities of the server will be to act as a sort of DNS lookup, keeping track of the most previous IP address which each individual user last used to log in with the client and updating this accordingly. When a new user connects, the server pings all users already present with the new user's IP address, which they use to establish a connection with the new user. The new user sends a query to the server likewise for the IP addresses of all users already present, and establishes connections with them individually. A consequence of this design is that multiple simultaneous logins will not be possible for any given user; usernames can only be in use on one system at a time, which the server's authentication system enforces. In the case that several users enter the system at once, the server queues several new IP addresses to existing users, for them to connect to. Once all users have a list of connections equal in length to the size of the initial chat "room" minus one (to account for themselves), the new user is successfully integrated, and is ready to proceed to updating their copy of chat history.

Users are authenticated with a password-based hash system. The client application first requests a password string from the user at login. Then, a hash key is generated from this string and sent to the server, along with another hash value generated from the user's account name. The server will store a simple associative list of these two keys, and checks for a match. If a match is not

found, the user may try a fixed number of times before the server enforces a penalty for wrong password tries (a delay). The possibility of hash collisions is remote but real; at account creation, the server will scan the internal list for the generated hash value, and request a new username if a match is found. This same process is repeated when the user chooses a password. Passwords remain fixed until a user requests a change, at which point they must first enter their current password before providing a new one, which is hashed appropriately on success.

Chat rooms are established either when a user invites another user to join a chat room, or are broadcast publicly. Using the list from the skeleton room which they initially entered, users can invite others to join a private chat room, and this private "tag" establishes visibility and accessibility for other users. Public rooms are broadcast to all other users, and once opened can be entered without an invitation. These chat rooms persist even when the user which opened them leaves the room, but are closed when the last user present disconnects (as do all chat rooms). Chat logs are updated to connected clients every fixed amount of time while the room is opened, and when the room finally closes the last user connected updates the previous version on the server. Because logs from the past will not change, "diff" files can be uploaded which do not require increasingly long copies (as part of their design).

Chat history for public rooms is stored locally in a compressed plain-text format, on the systems of every client machine which has entered that room, but a copy is also mirrored to the server whenever a given chat room is closed. The server additionally holds an encrypted log of the names of all chat room which each user has visited, which allows for chat history to be updated even on machines where the user has never run the client application before. When a user logs in on any system, the program runs a check for the names of chat room logs where they should be stored, and checks the names against those associated with the user's account. If any are missing then client will query the other users currently present, and if none of them have the logs available then the server will provide the client with a full copy of the logs. Chat history for private chat rooms is stored in a similar manner to the lists of chat rooms which users have visited, namely encrypted and uniquely to each user. Beyond this, private chat logs are served and updated in the same manner as those for public chat rooms. Notably, this includes un-encrypted, plain-text copies kept locally. This is not a security concern because any user which connects will have already had access; the added security on the server furthers the aim of private chat rooms approximating private messaging services, and indeed prevents anybody with server access from accessing these logs. A potential complication of providing access to chat logs over time is that of excessive disk space usage on the server. It is not clear just how much space we will have available, nor how well such an amount of space will be able to maintain our system over time, but should this become a significant issue the solution will likely be to implement some sort of queue on the server and the clients, where logs which are over a certain size or older than a particular age are truncated to reasonable levels.

# Timeline

## Server

Milestone	Days to Complete
Connect to clients	2
Create user accounts	2
Update user logs	1
Authenticate/log in users	1
Reset user password	1

## Client

Milestone	Days to Complete
Connect to server	1
Send account create request to server	1
Send login request to server	1
Send password reset request to server	1
Connect to peers	3
Distribute chat history to peers	3
Check authenticity of chat history against multiple peers	3
Distribute list of chat rooms among peers	1
Check authenticity of chat room list	1
Distribute new messages to peers	1
Check authenticity of message	1
Qt GUI	7

## Optional

Milestone	Days to Complete
Self balancing graph data structure to ease load on peers	5
Switching of chat rooms from public to private and vice-versa	1
Generate cryptographic keys for users, to keep private chats encrypted and secured	5
Generate cryptographic keys for chat rooms, to encrypt history for chats with more than two users	2

## Project risks

### Improper Scope

#### The Potential Problem

Due to losing a group member, and students getting confused about who was in their group, we ended up with a group of 5 people, which is the largest group. To compensate for our extra member, we chose an especially ambitious project. Trying to gauge the programming skill of all the members and estimate the time required to complete each milestone is difficult though, so we might end up with a project that is too ambitious, or perhaps not ambitious enough.

#### The Solution

We will start by implementing the core features first, focusing on the user's experience. This way if the scope is too large we still have a functioning core program. Once the core is complete we will organize additional features by time required to implement, chance of failure, and the feature's usefulness. This allows us to adjust our project scope larger or smaller as we need.

### Complications with Local Network Rules

#### The Potential Problem

Firewalls help keep computers safe from malicious attackers, but they also occasionally block legitimate applications. Our new, unknown application running on a typically unused port could be blocked by a user's firewall. Since this is a networked chat application that would break the functionality of the program.

#### The Solution

We can't configure a user's firewall for them, but we can let them know that they have to let us through the firewall if our application can't connect to the internet.

### Loss of Team Members

#### The Potential Problem

This is a difficult senior level class, and each member of our group has a school schedule they have to maintain, as well as a life outside of school including friends and family. Our group was formed after a student dropped the class, so we are well aware that sometimes you lose team members. The loss of team members affects how many man-hours we can put into the project, and therefore affects our timeline and ability to complete the project.

### The Solution

As laid out earlier in our solution for possible over or underestimating the scale of the project, we can dynamically change the scale of our project depending on how much time we have, so in the loss of a team member we could cut out some non-essential features to reduce project scale.

### Equipment Problems

#### The Potential Problem

On top of programs not being compatible across multiple platforms, the applications source code could also be lost in the event of a hard drive failure, accidental deletion, or data corruption.

#### The Solution

We're hosting all of our source code on GitHub, a professional source code hosting service. GitHub stores all of our data across multiple servers and keeps backups, so even if one server goes bad we don't lose any progress on our project.

### Program Incompatibility

#### The Potential Problem

There are multiple operating systems in widespread use today, and applications built for one don't run on the others. In our group there are two users with Apple MacBooks, one user with a Windows PC, one user with a Linux PC, and one user who actively uses all 3 of the above.

As the internet becomes more and more ubiquitous, more and more applications are using the internet in some form or fashion. Lots of people are running multiple online chat applications such as Slack, Skype, GroupMe, Hangouts, Ventrilo, Team Speak, Jitsi, IRC, etc. Even home appliances are starting to connect to the internet. As more things are connected to the internet, there becomes an ever increasing chance of running into a conflict with the port our application uses to connect to the network.

#### The Solution

To solve the first problem of running the program on multiple systems, we've decided to build our application on the cross platform Qt5 framework. This way other developers have done the bulk of the work for us in getting our application to as many users as possible.

To solve the second problem, we will consult the IANA database of registered ports to make sure we choose a port for our application that is in a range that firewall's aren't likely to filter out, and that doesn't conflict with any other applications who have reserved the port.

## Project not Scaling

### The Potential Problem

Networked applications are notoriously difficult to get right. Network constraints are often out of the users or developers control, so networked programs have to account for potential constraints and work around that. Peer to peer programs are even more challenging because you don't have dedicated hardware. One peer might be a brand new \$3000 gaming desktop in a major city with fast, reliable internet and another peer might be a 6-year-old laptop on radio-band internet outside of city limits in the mid-west. This adds additional complexity to balancing network loads and trying to ensure a fast, responsive application.

### The Solution

A smart choice of data structures will handle most of the network balancing for us, and monitoring ping times and network latency of peers will allow us to fine tune network balancing so that each user gets the best possible experience using our application.

### Progress

We have created a repository on GitHub to host all of our source code, and to track each member's progress on the project. Initial commits were made, starting by adding our names to the GROUP-INFO file. The repository is set up so that each user can commit changes using their own GitHub account.

Our team structure is as follows:

- **Team Lead**
  - Austin Hoppe
- **Documentation Lead**
  - Jacob Pebworth
- **Programming Lead**
  - Ryan Porterfield
- **Programmers**
  - Elexa Evans
  - Christopher Hines