Chatty Design Document

Thomas Chiovoloni, Ricky Riggot, Mevludin Guster

May 1, 2012

1 Design Overview

Chatty is a lightweight zero-configuration distributed chat application. While typically zero-configuration and decentralization are mutually exclusive, Chatty dodges this by only working on the LAN (Local Area Network).

Chatty is a UDP-based application, and does not rely on TCP for data transmission or connection. Every message sent by one of Chatty's users is broadcast to every IP address on the subnet. To handle the lost packets associated with UDP, Chatty sends messages each outgoing message ten times. Because every outgoing messages contains a UUID, the instance of Chatty receiving these messages is able to correctly determine and discard duplicates. From our tests we've found that an individual message has over a 90% chance of reaching its destination on a wireless LAN, and well over a 99% chance (we have never seen it occur, though it is theoretically possible) of reaching its destination on a wired LAN. In the case of the wireless LAN, this still results in a 99.99999999% of reaching its destination. Should the guarantee provided prove not to be sufficient, the number of repetitions is an easily changeable parameter for future developers who are maintaining Chatty.

2 Explanation of UML Diagram

This section describes the class diagram in `chattyclassdiagram.pdf`, located in this folder. It was created by running GraphViz on `chattyclassdiagram.dot`, also located in this folder.

2.1 Overview

Chatty follows a Model-View-Controller architecture for its design.

The controller (package `chatty.controller`) includes the classes which send and receive UDP data, as well as the classes for managing the exchange of data between the Model and the View. The controller is also tasked with abstracting the details of sending and receiving byte-streams across the network away from the rest of the system.

The model (package `chatty.model`) contains classes for representing and managing Users and Messages (both incoming and outgoing), as well as classes for managing the data which chatty stores while running (active users, message lists, etc). The `ChattyModel` class primarily serves to delegate messages sent to it to their appropriate destinations.

The view (package `chatty.view` and subpackage `chatty.view.htmleditor`) contains the classes for representing the model data in a presentable manner. It contains classes for displaying

2.2 Explanation

The Controller

Package `chatty.controller` contains six classes, only 5 of which ended up being used in the final design (`FileTransferManager` is present but useless, and regretably was not deleted before the deadline).

`ChattyController` is the main entry point of the application, so it seems like a reasonable place to start. `ChattyController` does most of its work during initialization. At this time it starts the `Inbox` and `Outbox`, creates instances of `ChattyView` and `ChattyModel` (telling the view to watch the Model), initializes the `Settings`, before finally starting to poll the users every second. After initialization it serves two purposes. First, it organizes the polling of users every second (as well as telling the model to clear its timed out users), and delegates messages between portions of the program which do not need to know details about each other. It is also the class that organizes the final actions before shutting down, such as saving the settings, and sending a sign-off notice.

`Settings` is the simplest class in the controller. It is implemented as a singleton, and as its name might suggest, it manages the users settings. Any data which needs to be when the program closes goes through the settings.

`ByteMaster`, despite it's humorous name, has a very serious task. The `ByteMaster` handles the encoding and decoding of data as bytes after packets are received and before they are sent. Chatty's current design does not elegantly handle messages. If it were to be rearchitected, `ByteMaster` would be renamed something like `MessageFactory`, and would only handle the decoding of messages. The encoding of messages would be added to part of the `Message` interface in the model. The current design of the model will be covered in the next section, including a discussion of its flaws.

The `Outbox` manages the sending of messages. It maintains an instance of a `Sender`, which is an inner class that implements `Runnable`, and runs in a separate thread. There is only ever one instance of `Outbox` running at a time, however it is crucial that it not be a singleton, as it must not be globally available, due to concerns about thread safety. Its inner class, `Sender`, maintains a queue of encoded messages for sending. Five times a second, the sender tries to send each message in its queue, removing it if it succeeds.

Similar to the `Outbox`, the `Inbox` maintains an instance of an inner class which runs in a separate thread, must only have one instance (not globally accessible, so not a singleton) open in chatty at a time. That is where the similarities end, however, as the `Inbox` is concerned with the *receiving* of messages. The Inbox has an inner class, the `Listener`, which waits on a background thread for packages sent to Chatty's UDP group. When a message is received, it decodes it and notifies the `ChattyController`, who forwards the message to the appropriate classes.

The Model

Package `chatty.model` contains classes for representing and managing domain data. It is fairly small, only containing 5 classes and one interface, all of which are mostly straightforward. so the discussion will be briefer than the previous section.

Class `ChattyModel` is the class representing the entirety of Chatty's data. It manages instances of the `UserManager` and `MessageManager` classes, and serves primarily to delegate method calls between them and the rest of Chatty.

`MessageManager` manages a list of messages. It implements `Observable`, so it can be watched for changes by the view. `MessageManager` is also the class which provides the ability to print messages out to a file (or any instance of OutputStream).

`Message` is a simple interface defining some getters that any Message must implement. `IncomingMessage` and `OutgoingMessage` both implement this interface, primarily providing different implementations of `toString`.

`UserManager` manages users, it maintains a set of users, and receives notification when users perform an action, such as sending us a message, or responding to pings. It is also the class which handles performing the action which a message requires be performed (e.g. change a user's name if a user sends a change-name message). Similarly to `MessageManager`, `UserManager` implements `Observable`, so that it may be observed by the view.

The model is the place that suffered the most from our lack of experience. As you may notice, there are only two classes for representing messages, however there are several types of messages being represented. Additionally, you might notice a lack of a `User` class. During the initial implementation, no reasonable way could be found to handle duplicate names. This, combined with flaws in our original design which lead to a difficult implementation of `User`, lead to users being represented internally as strings. This was an enormous mistake, and failed to take advantage of object orientation. In retrospect, the user class should have maintained three fields, it's name, it's IP, and a "last seen" date. During this hypothetical redesign, an `IUser` interface should be created so that in the future, should change become necessary, we can easily create different types of users. Additionally, the way messages are handled could use substantial redesign. The `Message` interface should remain, but methods involving serialization to bytes should be added. An package-private abstract class, `AMessage`, should implement the repetitive parts of `Message` allowing for little overhead for creation of new message types. The system as a whole should not refer to `AMessage`, so not to differentiate between wholly new implementations of the `Message` interface and subclasses of `AMessage`.`chatty.controller.ByteMaster`, then would be renamed to `chatty.model.MessageFactory`, and would handle the creation of messages from their underlying byte representations. This design would greatly simplify the code present throughout the entire system.

The View

Package `chatty.view` is conceptually the simplest package in this program, much of the code inside of it was generated by UI code generation tools, and so while it contains the bulk of the code present in the system, the least amount of time will be spent discussing this package. Primarily it contains 4 classes and one subpackage, with several auxiliary classes which were used for UI widgets.

Class `ChattyView` is created upon initialization of the program by the controller, and is a subclass of `JFrame`, and maintains instances of `InputPanel`, `UserPanel`, and `MessagePanel`, classes for receiving user input, displaying the list of currently signed-on users, and displaying the messages. The `InputPanel` and `MessagePanel` are both instances `HTMLEditor`, from the `chatty.view.htmleditor` which is itself a subclass of JEditorPane, so that it can display rich text, in the form of HTML. The user enters potentially formatted messages via the `InputPanel`, which notifies the `ChattyView` and in turn the `ChattyController`. Via the input panel, the user is able to select several different types of rich text formatting, which is visible in real time, and applied immediately to the relevant portion of the `InputPanel`. Messages travel over the internet, get received by the view, and are displayed on the `MessagePanel` in a formatted manner.