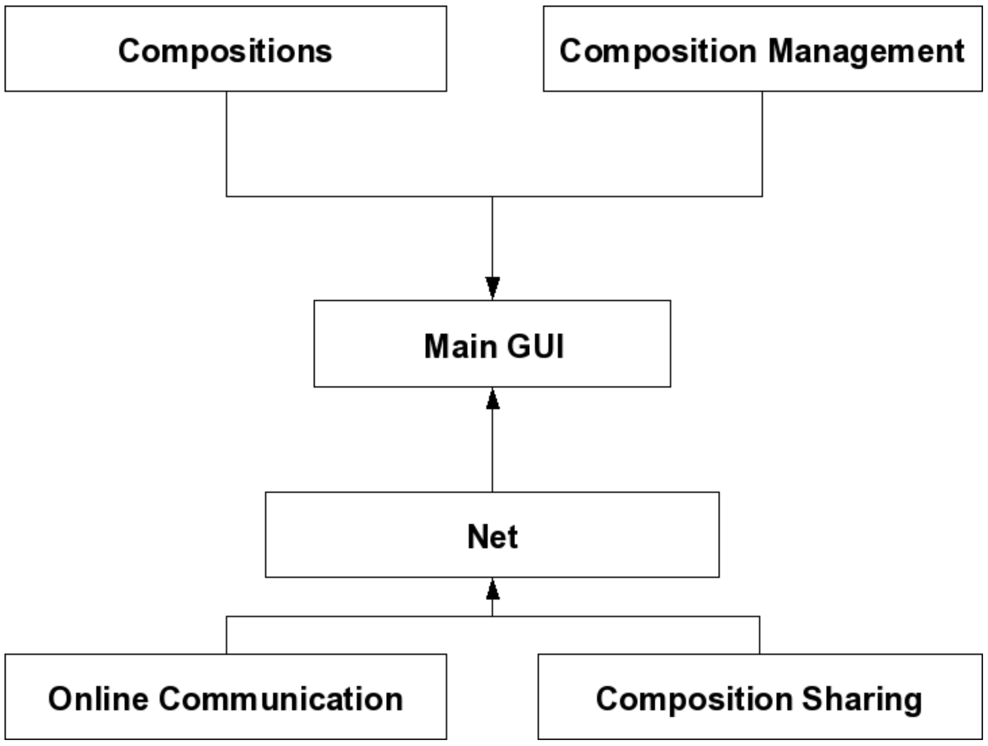
Design and Architecture

Report on the design and architecture of the sub-systems defined in the requirements document.

Table of Contents

Overview

We have split the design document in size general sub-systems. Here is a high-level graphical representation of these subsystems:



For each sub-system there are high-level descriptions as well as the design-decisions made during the design process. UML Class-diagrams and sequence-diagrams have been also been added to show the interactions between our static class-diagrams. We initially designed UML class diagrams before we began our programming iterations to help us develop the system such that it meets the requirements that we gathered at the start of the project; most of these can be found on our CVS website. Some of the old UML class diagrams have been added to show the comparisons between what we initially designed and what we actually implemented.

GUI Sub-System

Purpose

This section describes the design decisions of the Graphical User Interface (GUI) – from structuring the display of data to managing windows and displaying file transfer invitations.

General Priorities

Decisions in the user interface area were based on the following priorities (most important first):

1. Integrating well with other subsystems.
2. Aesthetically pleasing.
3. Simple and easy to use, including a fast learning curve.
4. Quickly achieve desired actions. This means responsiveness and low click-count.
5. Maintainability.

Outline of design

The GUI is based on the Model-View-Controller design pattern.

Major Design Issues

Issue 1: GUI implementation and design pattern.
It seemed like it was going to be complex to integrate our musical structures (like track, chord, composition, etc.) with JMusic. Thorough testing would be required, and we didn't want to mix the GUI up in that, knowing that a user interface is very hard to unit test. I wanted a user interface that would be responsive.

Option 1.1
The first option was to use a responsive GUI system written in C++ and adapted to work with java through an interface. This satisfied the fourth priority, but would be impossible to integrate with junit. This ruled option 1.1 out instantly because the first priority wasn't met at all.

Option 1.2
Swing combined with Model-View-Controller pattern. This would allow us to test as much as we could in the underlying model data structures, and to create a visual display representing the underlying data at all times. The display is kept in sync with the data through subjects and observers. The model is a subject and the view is an observer. JMusic also used swing, and would integrate seamlessly into our program.

Decision 1
Option 1.2: Swing combined with Model-View-Controller was chosen. Doing it this way allowed us to split the job up into tasks - have one person design and implement the model, and another implement the user interfaces to change and display that model. Edit actions would typically be used as a "controller" to edit the model. The GUI simply has to create the correct action and execute it to change the model. The model will then notify it's observers of the change, and the gui observer would reflect the change visually. Edit actions can also be queued in an undo manager and if they are coded carefully such that they can redo their action to the previous state, we can then easily implement undo and redo into the composition editing.

Issue 2: Managing windows/panels
After realizing that there is going to be a lot of different windows: compositions, online conversations, download transfers and several more… we saw a need to design a window system that would allow us to manage so many separate windows.

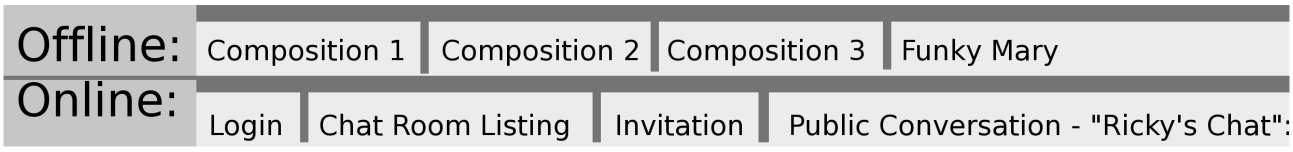
This was especially important to our mission statement of integrating sharing and editing into one application.

Initially, Brook wanted to reside conversations in a separate window to the program (like MSN messenger), but Ryan pointed out that it could become confusing and the windows may look like a separate program. This also takes away from the whole notion of having the composing/sharing all-in-one.

After sketching several designs; including layers of tabs, we were stuck on how to organize the tabs. We didn't want all the tabs in one line because the user would have to search through a mixture of tab-views to locate the one they wanted. Also the tab control didn't support arranging/moving tabs.

Option 2.1
Separating offline-tabs such as compositions from online-tabs such as chat-rooms by completely dividing the screen area in a split – for example like excel and eclipse do.

Option 2.2
Have the tabs individually close-able, and listed in two separate rows - one tacked above the other. This would require a custom tab control. For example:



Decision 2
It would have been too time-consuming to implement a custom tab-control to do option 2.2. So we ended up having the tabs separated by using a split-pane: offline tabs on the top and online tabs on the bottom. This turned out great because it allows the user to do things simultaneously like chat while working on a composition – which meets our goal by bridging the gap between sharing/composing; users can discuss what they are working on live!

Issue 3: Invitations less "in your face"
Brook initially wanted invitations (such as requests to become a friend) to popup in a top-most windows and demand a response. Ryan pointed out that this would be interrupting to the user – especially since they could be in the middle of working on a composition, it could really disrupt and annoy them.

Option 3.1
Pop up invitations in small windows. This would satisfy all priorities.

Option 3.2
Pop up invitations in new tabs, without bringing that tab to be actively visible on the screen. This would satisfy all priorities, but rated better by being more aesthetically pleasing.

Decision 3
We decided to use the tab-system and notify the user in a more passive manner by adding a tab to the online tab view (and giving it focus) – this way when the user is working on a composition, they can finish the task they are working without having to make a decision / click away a dialog.

Details of design

The Model-View-Controller pattern was used extensively throughout our system. This provided more familiarity when working with each other's subsystems. It also enabled us to split tasks – for example Ryan worked on the GUI views for a lot of the networking data models Brook did.

Composition Management Sub-system

Purpose

This section describes the design of requirement 1.

General Priorities

Decisions in the library management area were based on the following priorities (most important first):

1. Fulfilling use-case requirements.
2. Simplicity.
3. Able to work with file transfers as well as offline editing.
4. Ease of implementation, considering the short time span.

Outline of design

The composition library manager is a panel that typically sits on the left side of the user interface. It is responsible for loading, saving, renaming, importing, and exporting compositions in various formats.

Major Design Issues

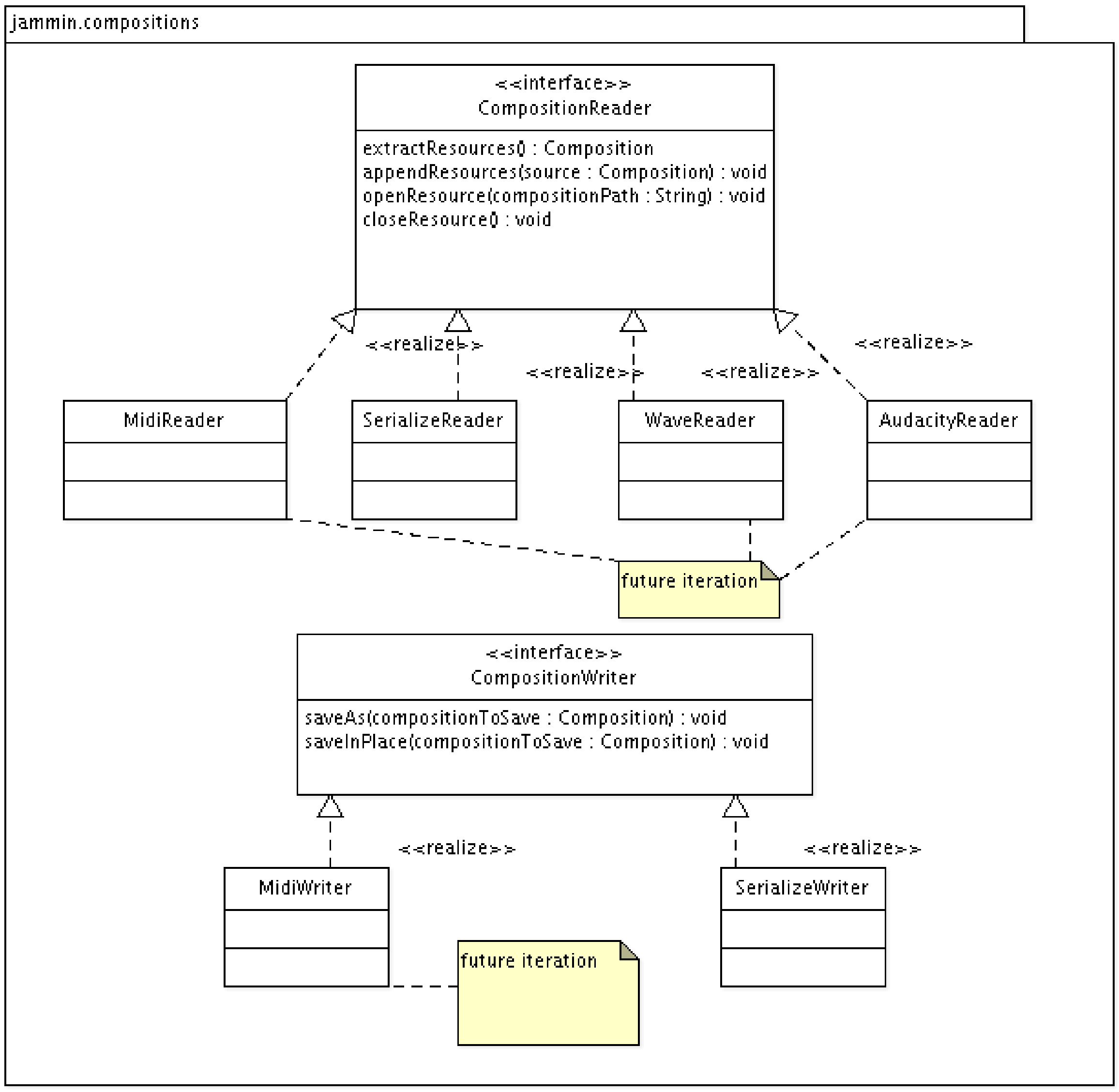
Issue 1: Scope
The first issue encountered with the composition library manager was how much was it going to do?

Option 3.1
This idea is to keep a repository of all the user's compositions in one place where they can always be seen from within the program, and where they can be edited quickly with external tools. All files would be kept in a subfolder of the user's Home or Documents directory (depending on operating system), and can be accessed outside the program while the program is running. This would provide greater interoperability with other software. One draw-back is that security is compromised: composers may want to encrypt their compositions so no one else can edit them or distribute them (intended for development in a future iteration).

Option 3.2
This idea is to store all compositions in an internal file, such as a database. Only the jammin program would be able to access the files. This would provide an excellent security layer, but would make interoperability difficult for other programs.

Decision 3
The program was designed to use option 3.1, but with an interface that may later become option 3.2 without changing the rest of the program, should security become a priority in the future. Although this went against our generally agile design philosophy, I felt it was important to generalize the interface for the purpose of security.

Managing Compositions Sub-system UML Diagrams



Compositions Sub-system

Purpose
The compositions sub-system allows user to create musical scores by creating tracks of MIDI music and then arranging them into sections and a complete composition. This relates to sections eight to twelve of the requirements.

Priorities
- Create realistic sounding musical compositions
- Ease of use
- Re-use as much JMusic functionality as possible
- Create a component which will work within the overall system
- Create a design which is easily maintained

Outline
The composition sub-system allows users to create and arrange compositions. The model-view-controller and observer design patterns are used to create a thin GUI over top of the data system. The composition system builds on the functionality provided by JMusic.

Design Issues

Issue 1
How should JMusic be incorporated into our software?

Option 1.1
Extend the classes provided by JMusic and add functionality to the new subclasses

Option1.2
Create a new design which uses JMusic internally

Option 1.3
Add code directly to the JMusic source code

Decision:
1.2 Create a design which uses JMusic but also fits in with the design patterns being used throughout the system.

Issue 2
How do we structure the music so it is both easy to view and edit?

Option 2.1
Construct compositions which are just long single midi tracks for simplicity. When played the screen scrolls across to show the point of playback

Option 2.2
Similar to 2.1 but instead of scrolling across the page have tracks wrap around so it is like using a typical musical score.

Option 2.3
Break the composition up into small sections that can be easily viewed and then arrange them into a complete composition.

Decision
2.3. By using the idea of sections, small repeating parts of a song can be created and

reused. This closely resembles the process of creating verses and choruses used when creating songs.

Details of Design

Initial research into creating MIDI music within a Java application found an open source library called JMusic. JMusic provided classes and tools for storing and manipulating MIDI data. There was also tools for playing MIDI sound from the JMusic objects and classes for displaying JMusic objects on musical staves. It was decided that this functionality should form the basis of our system.

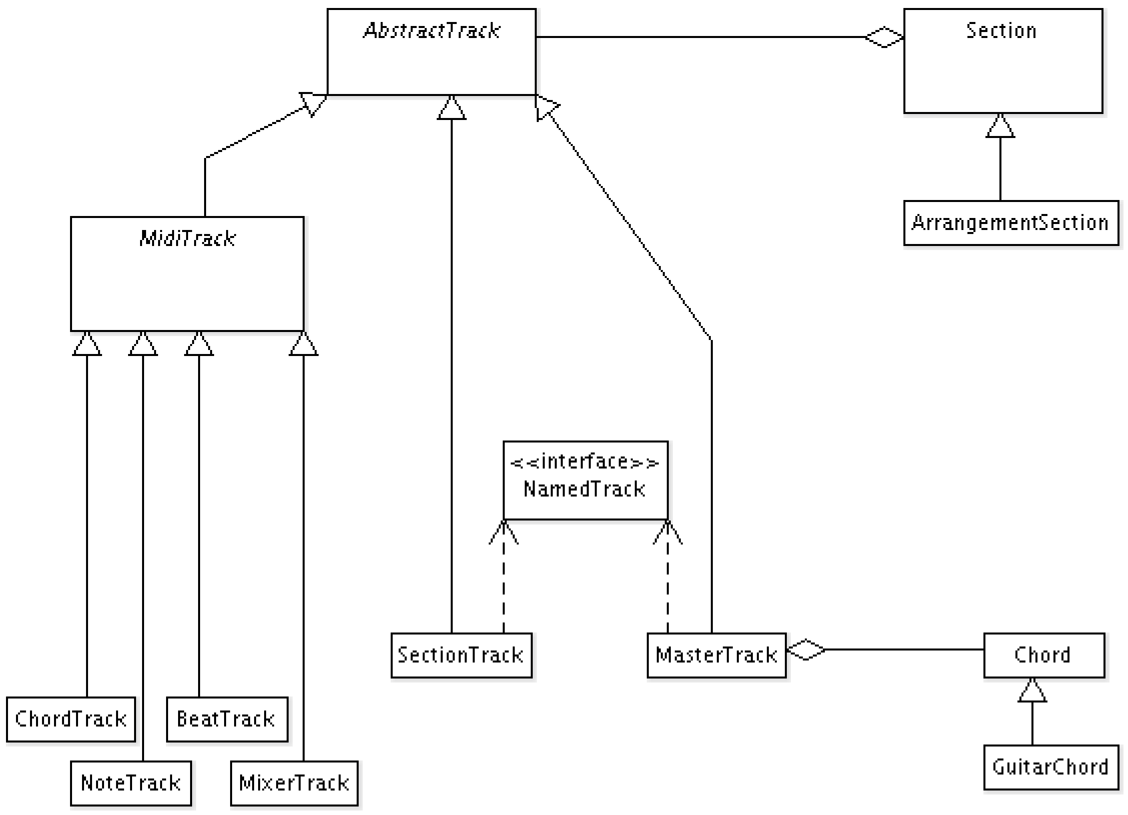
The composition sub-system uses the observer and model-view-controller design patterns which are used throughout the project. We wanted to try and keep the GUI to just a thin layer of the top of our data so that the data model could be unit tested. By using the model-view-controller pattern we were able to achieve this. In most cases the controller and view were combined because we only had one type of view for each object. The observer design pattern was used so that the data could exist without having to know any implementation details about the GUI. Any objects that wanted to be infromed of changes to another object just implemented the Observer interface and then decided themselves how the change to the subject should be handled.

The composition data structure uses a top down approach to collect all the data used in a composition.
The top level composition object represents the highest level of abstraction and the further down the design goes the more specific each object gets. Each composition is made up of a collection of sections and an arrangement. The arrangement is used to specify what order the sections are played in and how many times each section is played. Within each section there are multiple tracks. At the track level the musical notions of instruments, notes and chords are introduced.

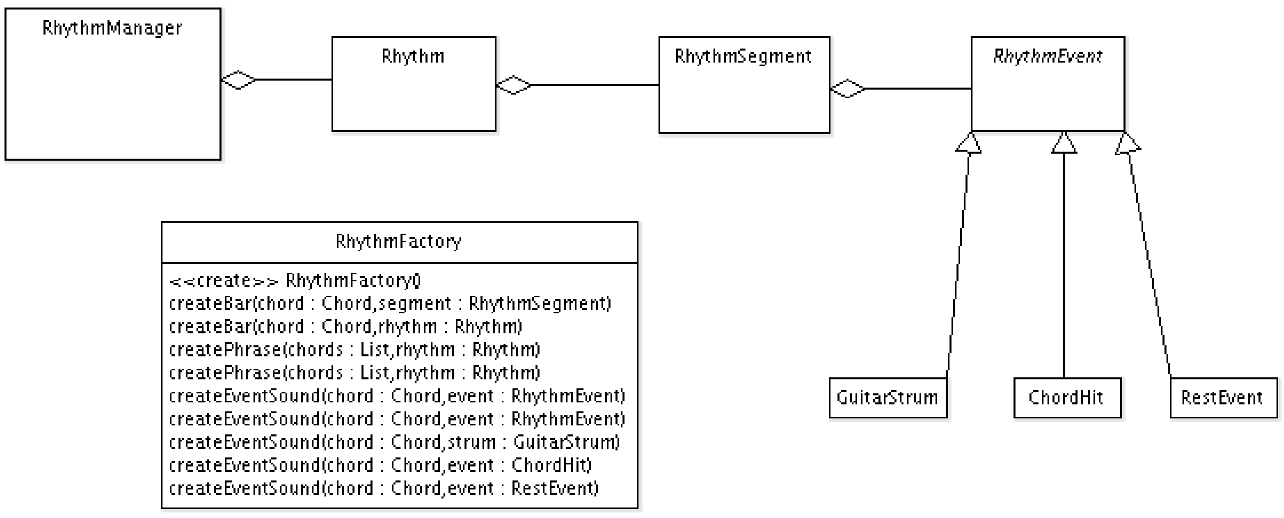
The JMusic structures are used within tracks to store MIDI information. By implementing our tracks as a wrapper around the JMusic data we could make use of the functionality provided by JMusic and also make sure that the our data structures would fit in with the observer design pattern. JMusic did not use the observer pattern so any changes to the midi data that would result in a change to the GUI needed to be done through the Track class which could raise a event signalling that the model had changed.

Compositions Sub-system UML Diagrams

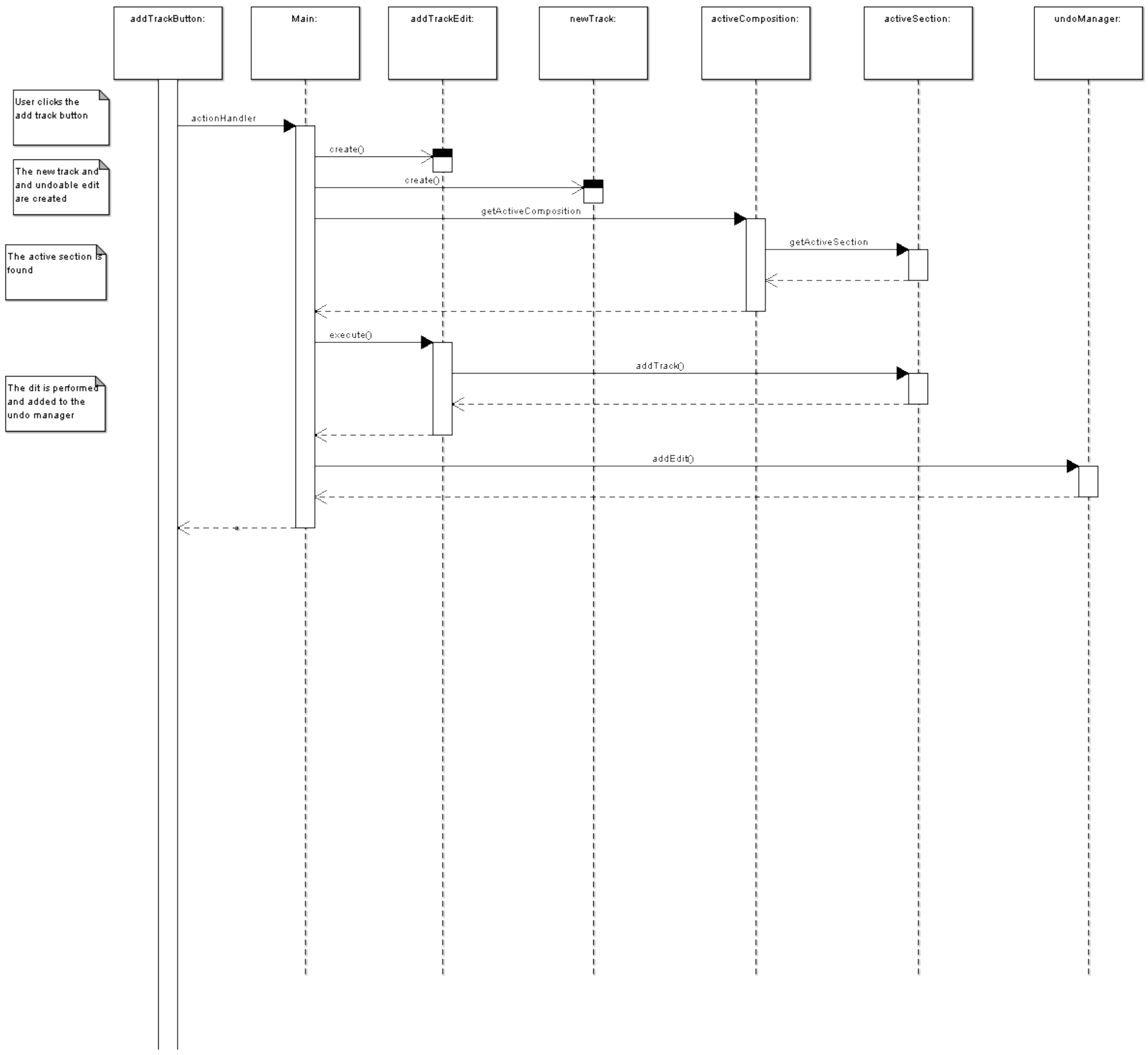
These models represent various parts a Jammin composition:



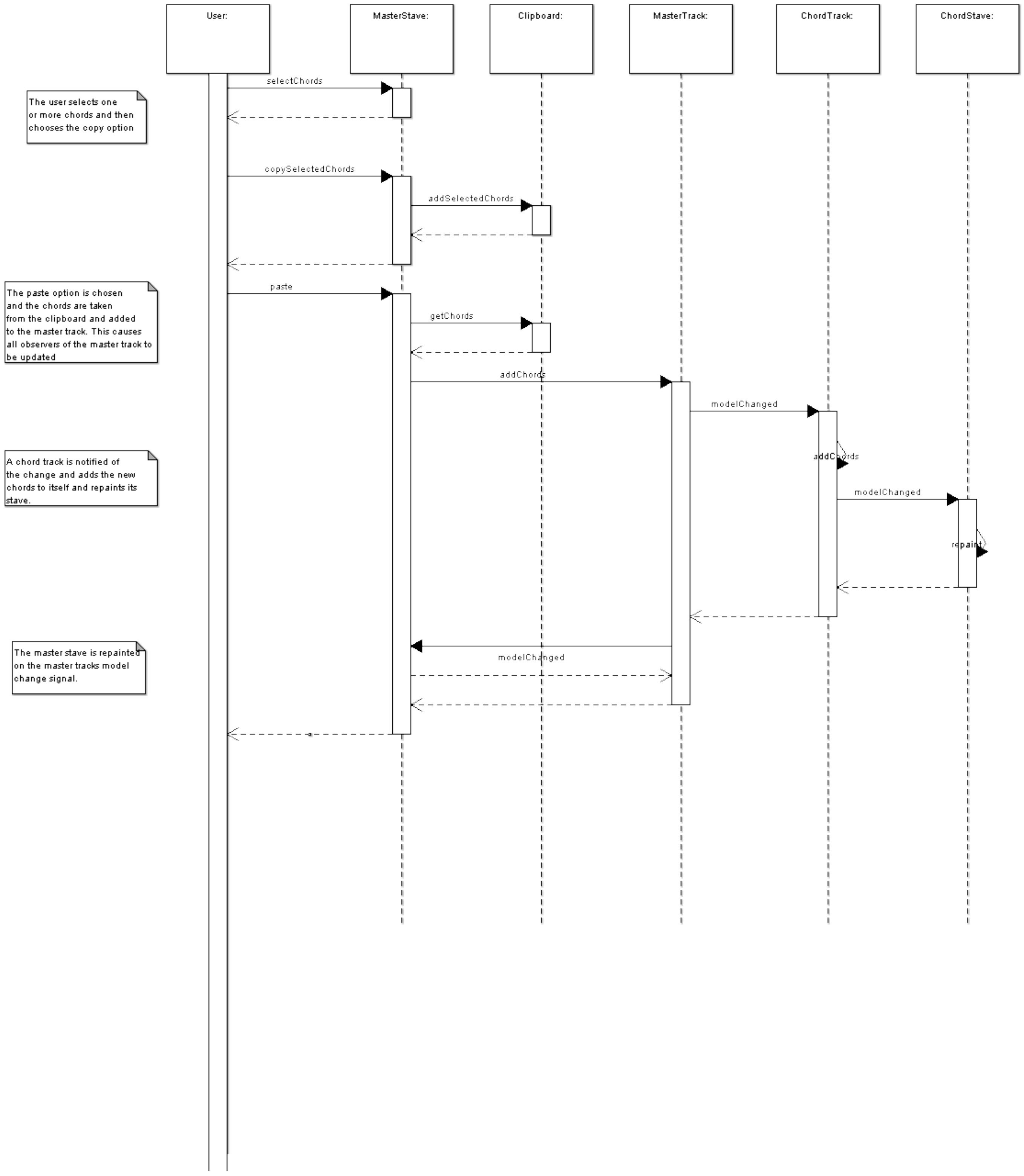
This is the data-system used for constructing and managing rhythms:


RhythmManager
Rhythm
RhythmSegment
RhythmEvent
RhythmFactory
<<create>> RhythmFactory()
createBar(chord : Chord,segment : RhythmSegment)
createBar(chord : Chord,rhythm : Rhythm)
createPhrase(chords : List,rhythm : Rhythm)
createPhrase(chords : List,rhythm : Rhythm)
createEventSound(chord : Chord,event : RhythmEvent)
createEventSound(chord : Chord,event : RhythmEvent)
createEventSound(chord : Chord,strum : GuitarStrum)
createEventSound(chord : Chord,event : ChordHit)
createEventSound(chord : Chord,event : RestEvent)
GuitarStrum
ChordHit
RestEvent

Here is a sequence showing the interaction between composition classes when a user actions to add a new track, it also shows the action is undoable by interacting with the undo-manager:



Here is a sequence diagram showing the interactions between composition classes when copying and paste chords:


User:
MasterStave:
Clipboard:
MasterTrack:
ChordTrack:
ChordStave:
selectChords
The user selects one or more chords and then chooses the copy option
copySelectedChords
addSelectedChords
paste
The paste option is chosen and the chords are taken from the clipboard and added to the master track. This causes all observers of the master track to be updated
getChords
addChords
modelChanged
addChords
A chord track is notified of the change and adds the new chords to itself and repaints its stave.
modelChanged
repaint
modelChanged
The master stave is repainted on the master tracks model change signal.

Net Sub-system

Purpose
This section outlines the network-design decisions which requirement 3.x, 4.x, 5.x and 6.x depend on. It also describes important aspects of the implementation for requirements 3.x (accounts and logging in).

General Priorities
Decisions for this sub-system are made based on the following priorities (most important first):
- Maintainability
- Simplicity
- Consistency (with rest of systems core architecture).
- Reliability
- Security
- Integrity (user Account data)
- Efficiency

Outline of design
We decide on a network architecture to support all the online functionality for the system. We then address how we meet the requirements under 3.x and which requirements we have decided to omit from the project. It discusses why we have chosen to use the Model-View-Controller architecture for the client-side and finishes up with general thoughts/idea on how we will approach testing.

Major Design Issues

Issue 1:
What sort of network architecture should we use to support all the system online features?

Option 1.1:
Client-Server Architecture: Have a Root Server for managing the accounts and managing communication between Clients.
This design is simple, straight forward /and easy to maintain. However it does have a central-point of failure and its efficiency is poor (especially if it were to support many users internationally).

Option 1.2:
P2P architecture: Still have a root-server for managing accounts, but have direct communication links between clients rather than routing all info via a server. This design has better efficiency than option 1.1. However it is an extremely complex design (For example, there are issues with NAT boxes + firewalls, an issue highlighted in net file-share prototype). Even if we used a free existing library that handles the technical / complex issues (like JXTA), they are still very complex and difficult to use. This design would be harder to test than option 1.1 which will make it hard to produce something more reliable. Also it still has a central point of failure because the system will still rely on some root-service – P2P would just be more easy-going on the root server (i.e. Reducing packet traffic).

Decision 1:
1.1 mainly for simplicity. If the system was a commercial product designed to support many international users option 1.2 would be the best option, however it would take too long to implement for the duration of this small-sized project.

Issue 2:
Where and how will the account data be stored / maintained and what level of security should we provide?

Option 2.1:
SQL Database on an SQL server

Option 2.2:
Text file local to the servers.

Decision 2:
2.2, for simplicity. Although data integrity is not as easy to maintain using text files, simplicity is a much higher priority for this project. However, the design calls for the account system to be a self-contained system, so it would be easy to replace when moving beyond the course into a commercial product. Encryption would also be simple to implement, if that option was decided on. As the system no longer supports friends due to time constraints, SQL would also have required a large overhead in processing and maintainability which would have been efficient, given our low requirements in this area. This decision meets requirements (3.3.1), (3.4.1) and (3.6). Due to time constraints, several requirements have been dropped, including profiles (3.4) and friends (5)

Details of design

The server will be running a separate program to the client. The Client code for providing net functionality will be all accessed (by the rest of the system) from a singleton class called “Client” in the jammin.net.client package. The server program will reside in the jammin.net.server package and will be a command-line application.

The root server and the clients will communicate by sending packets over a TCP connection – Java's sockets will be used to accomplish this. All communication between clients are send to the root-server to be routed to and from the clients. Both the server and the client will have a “MessageManager” class that can send packets over a connected TCP channel and also listen for incoming packets. As mentioned previously; the server will analyses incoming packets and re-route or respond to the client depending on the packet type. When the client receives a packet from the server it's message manager will analyze the packet an route it to a specific model class stored in the Client singleton.

To meet requirements under 3.6, established all communication channels must be authenticated to ensure at least a basic level of security. An authentication packet is to be sent first by the client containing a username and password to authenticate the connection. Because security is a low-priority for this project we are not planning to address many network-security issues.
For all the network-based sub-systems, they contain a lot of non-deterministic code due to sockets and threads. We cannot unit-test a lot of our code, although we aim to unit-test as much as we can. We are planning to write some test-harnesses that can simulate the use-cases and hopefully introduce some variations and test for exceptions. This will help make our system more stable / reliable.

Client-Side Details

For Client-side code we want to keep it's architecture as consistent as we can to fit in with the rest of the systems design. So I have decided to use the Model-View-Controller architecture. The controller in this case is both from the GUI and from the message-manager (remotely being controlled by a server to some extent). The Model classes will be designed so that a view can observe them and wait for notifications for changed data. This also another benefits:

1. The model code can now be more unit-testable since the model classes will be less reliant on network / threaded code. This helps improve the reliability of the network code.

2. We can now divide up tasks; so that one person can work on the client classes (model/controller) and another can work on the GUI (view).

Server-Side Details

The server-side architecture was designed to be as modular as possible, allowing new features to be added in easily. This also allows modules to be replaced with equivalent, more efficient modules without disrupting the rest of the system. The server-side system was furthermore divided into two separate parts, each capable of being run on separate machines. These were the:

- **Root Server**:
  Manages user accounts and file transfers between clients.

- **Dedicated Server**:
  Holds the chat-rooms which clients converse in.

This was done to distribute the load between several machines, allowing more users to be connected simultaneously.

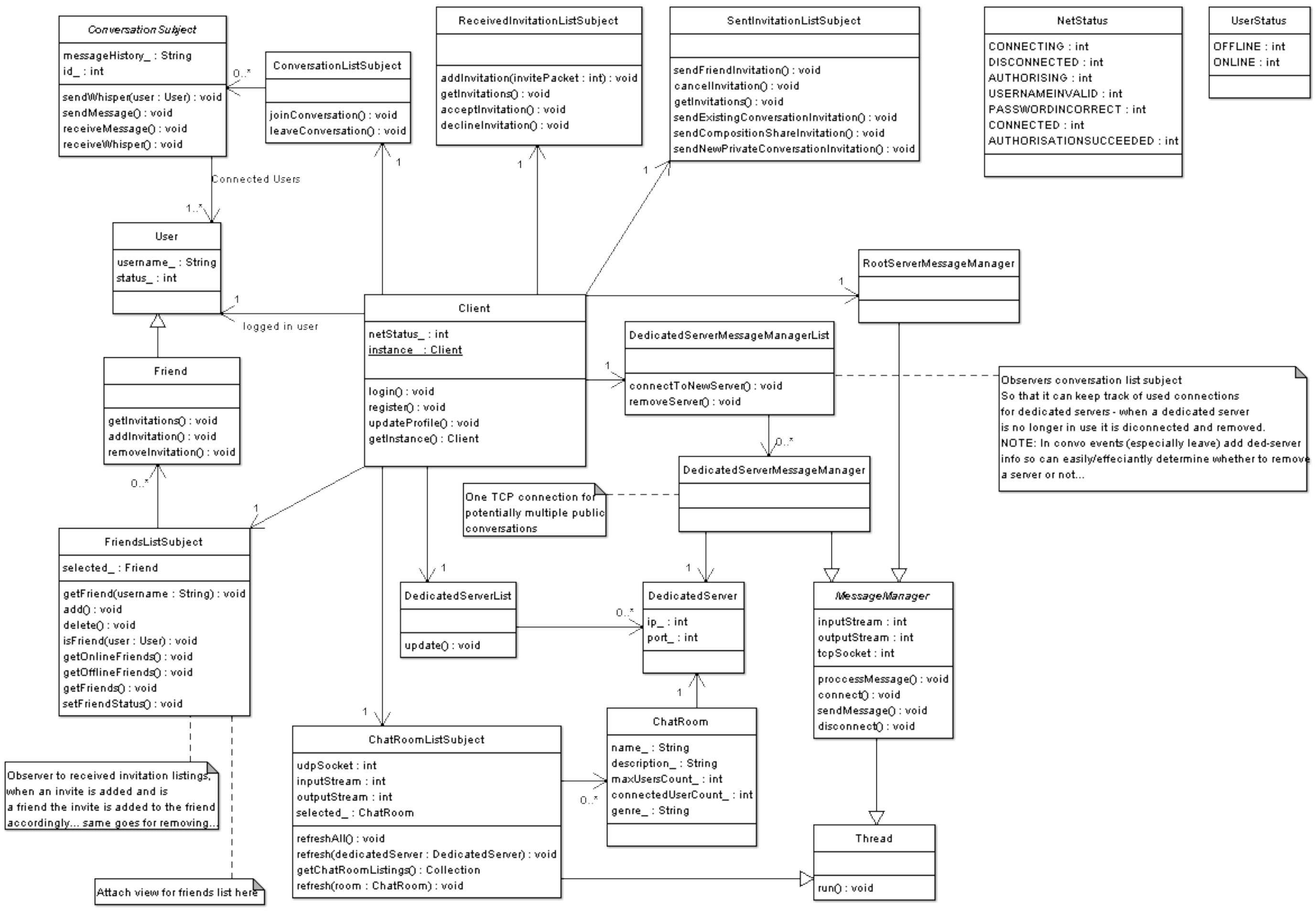
Protocol Details

When connecting to the server a user must send an authentication packet as the first packet to the server. The server sends back a response (a success or failure depending on the credentials).

To logout, the client simply disconnects the socket (ends the TCP connection).

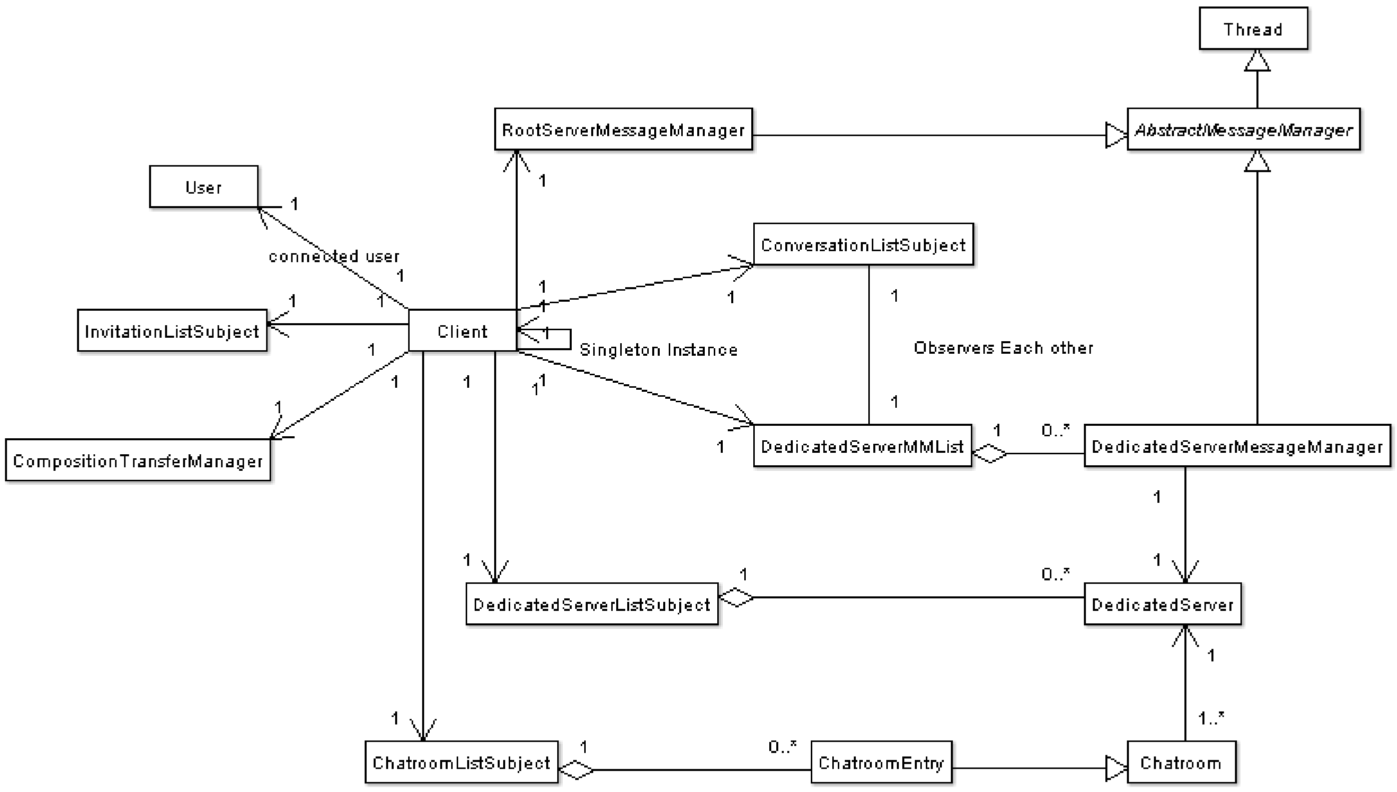
Net Sub-system UML Diagrams

We initially designed some UML class diagrams for a reference to aid us during development. This upfront design really helped us in the long-run. Here is the initial Client-Side overview UML class diagram:



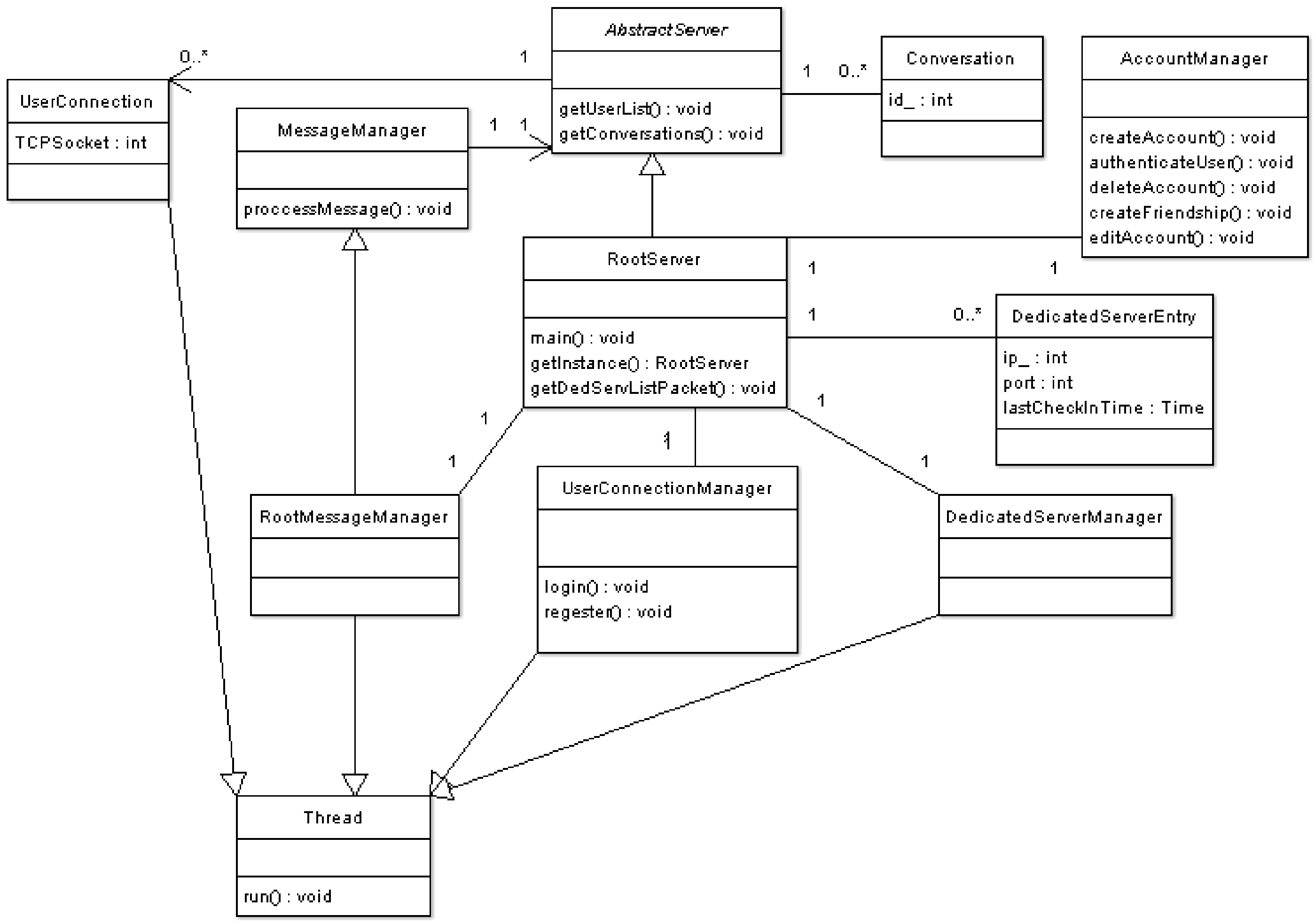
Conversation Subject
messageHistory_ : String
id_ : int
sendWhisper(user : User) : void
sendMessage() : void
receiveMessage() : void
receiveWhisper() : void
ConversationListSubject
joinConversation() : void
leaveConversation() : void
ReceivedInvitationListSubject
addInvitation(invitePacket : int) : void
getInvitations() : void
acceptInvitation() : void
declineInvitation() : void
SentInvitationListSubject
sendFriendInvitation() : void
cancelInvitation() : void
getInvitations() : void
sendExistingConversationInvitation() : void
sendCompositionShareInvitation() : void
sendNewPrivateConversationInvitation() : void
NetStatus
CONNECTING : int
DISCONNECTED : int
AUTHORISING : int
USERNAMEINVALID : int
PASSWORDINCORRECT : int
CONNECTED : int
AUTHORISATIONSUCCEEDED : int
UserStatus
OFFLINE : int
ONLINE : int
0..*
Connected Users
1..*
1
1
1
User
username_ : String
status_ : int
1
logged in user
Client
netStatus_ : int
instance_ : Client
login() : void
register() : void
updateProfile() : void
getInstance() : Client
RootServerMessageManager
1
DedicatedServerMessageManagerList
connectToNewServer() : void
removeServer() : void
1
Observers conversation list subject
So that it can keep track of used connections
for dedicated servers - when a dedicated server
is no longer in use it is diconnected and removed.
NOTE: In convo events (especially leave) add ded-server
info so can easily/effeciantly determine whether to remove
a server or not...
0..*
DedicatedServerMessageManager
One TCP connection for
potentially multiple public
conversations
Friend
getInvitations() : void
addInvitation() : void
removeInvitation() : void
0..*
1
FriendsListSubject
selected_ : Friend
getFriend(username : String) : void
add() : void
delete() : void
isFriend(user : User) : void
getOnlineFriends() : void
getOfflineFriends() : void
getFriends() : void
setFriendStatus() : void
1
DedicatedServerList
update() : void
1
DedicatedServer
ip_ : int
port_ : int
0..*
MessageManager
inputStream : int
outputStream : int
tcpSocket : int
proccessMessage() : void
connect() : void
sendMessage() : void
disconnect() : void
1
ChatRoom
name_ : String
description_ : String
maxUsersCount_ : int
connectedUserCount_ : int
genre_ : String
1
ChatRoomListSubject
udpSocket : int
inputStream : int
outputStream : int
selected_ : ChatRoom
refreshAll() : void
refresh(dedicatedServer : DedicatedServer) : void
getChatRoomListings() : Collection
refresh(room : ChatRoom) : void
0..*
Thread
run() : void
Observer to received invitation listings,
when an invite is added and is
a friend the invite is added to the friend
accordingly... same goes for removing...
Attach view for friends list here

Here is the actual Client-side overview UML class diagram:

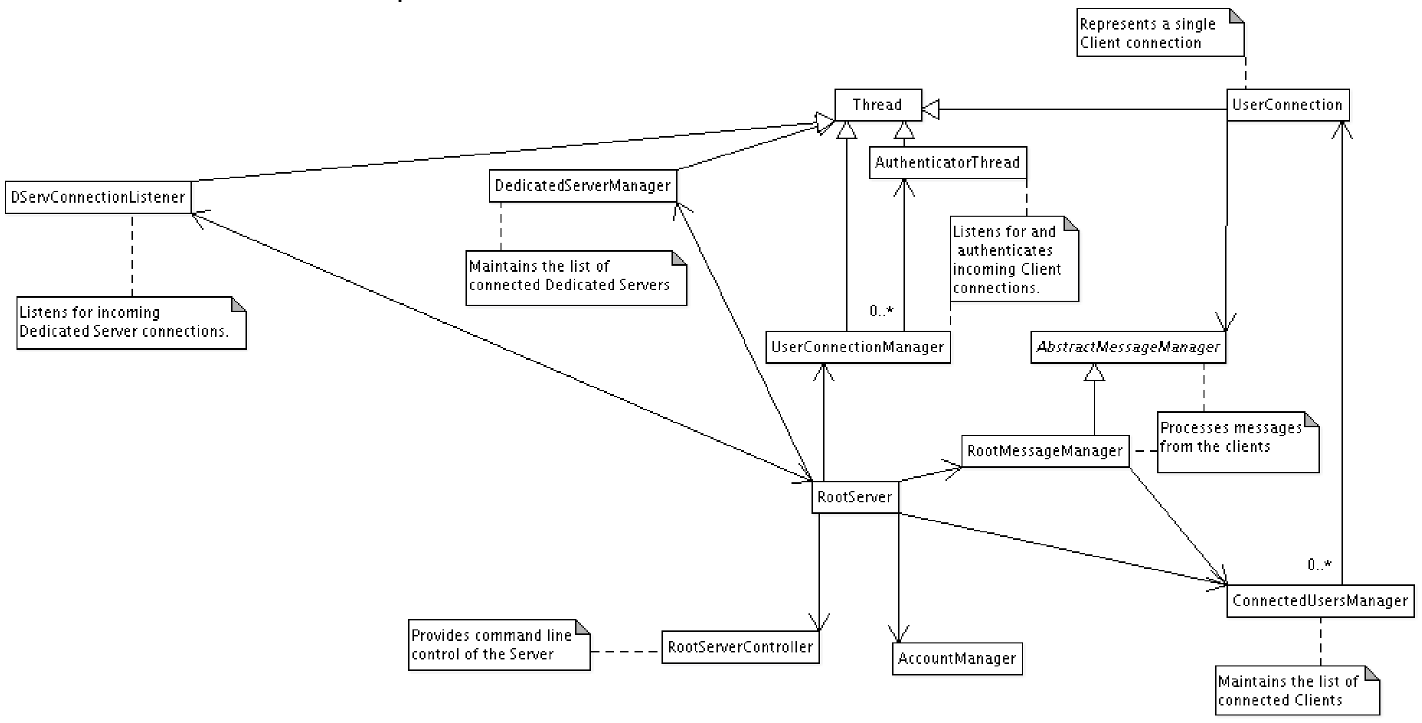


Many of these classes are models (subjects) so that views can observe them. All net-functionality in the client can be access through the Client's singleton.

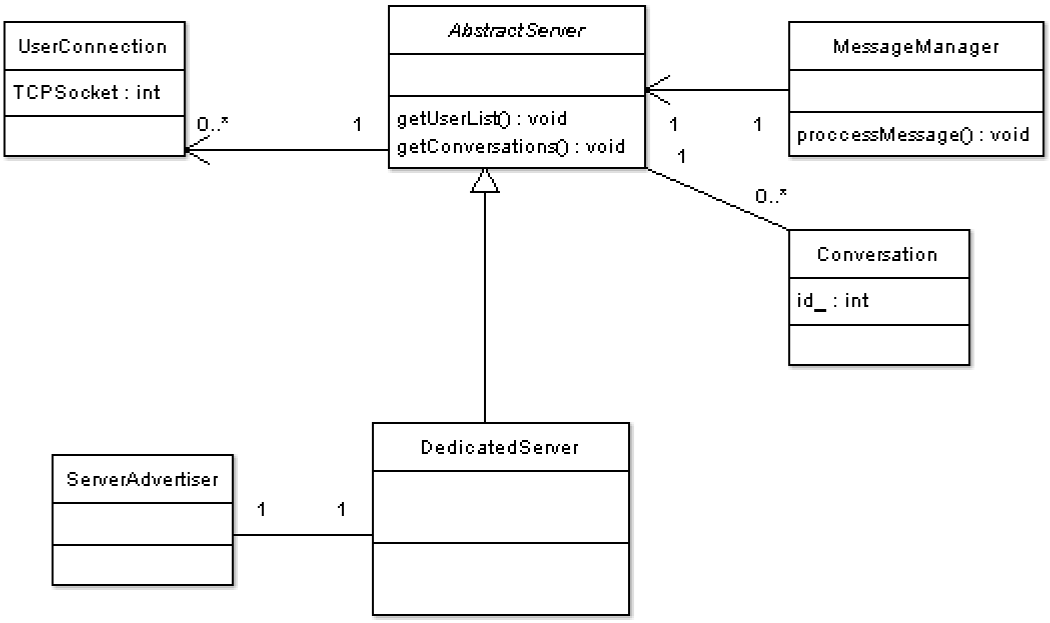
Here is the initial UML diagram for the root server:



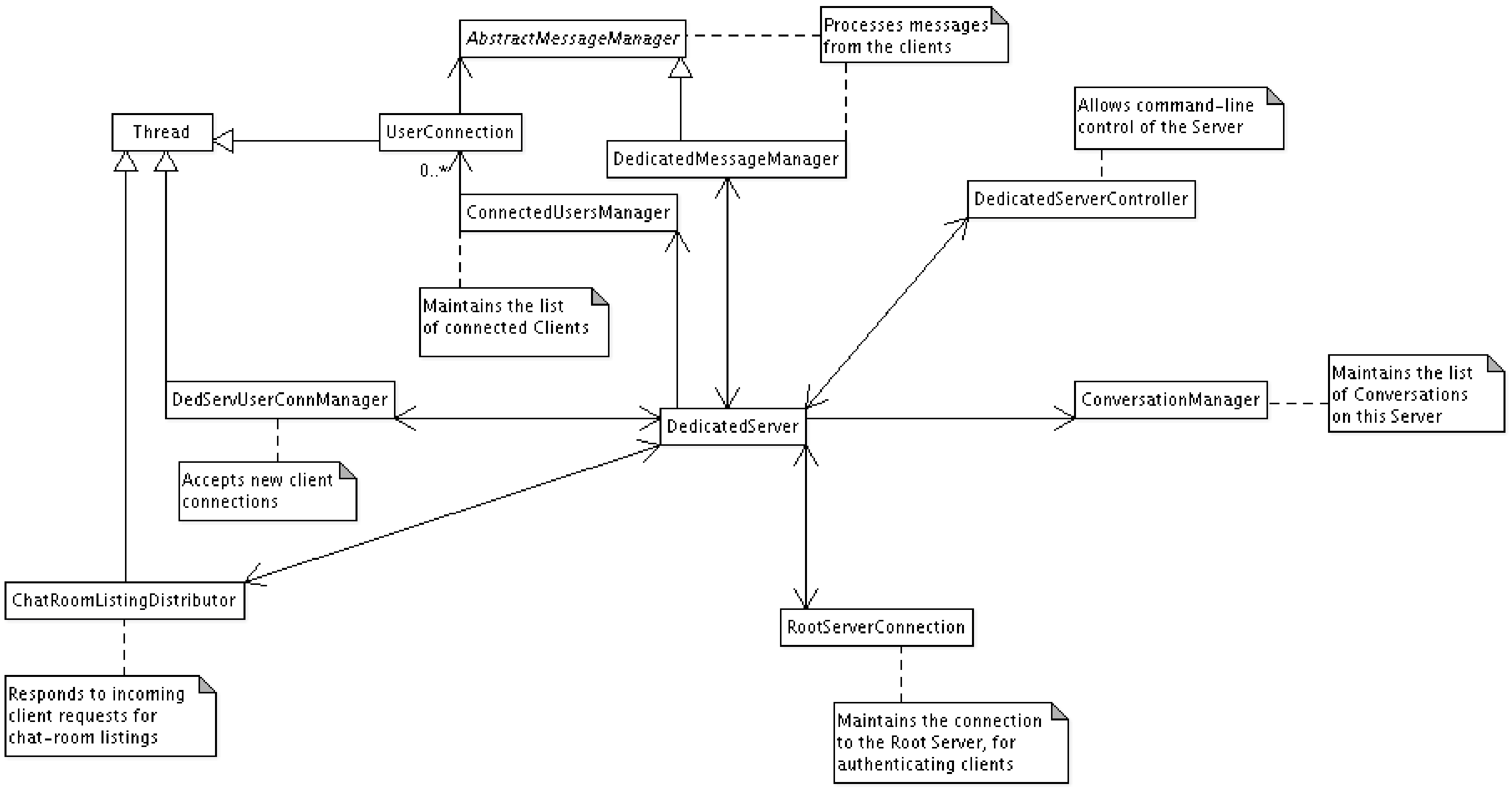
This is the actual implementation of the server:


Represents a single Client connection
Thread
UserConnection
AuthenticatorThread
DServConnectionListener
DedicatedServerManager
Maintains the list of connected Dedicated Servers
Listens for and authenticates incoming Client connections.
Listens for incoming Dedicated Server connections.
0..*
UserConnectionManager
AbstractMessageManager
Processes messages from the clients
RootMessageManager
RootServer
0..*
ConnectedUsersManager
Provides command line control of the Server
RootServerController
AccountManager
Maintains the list of connected Clients

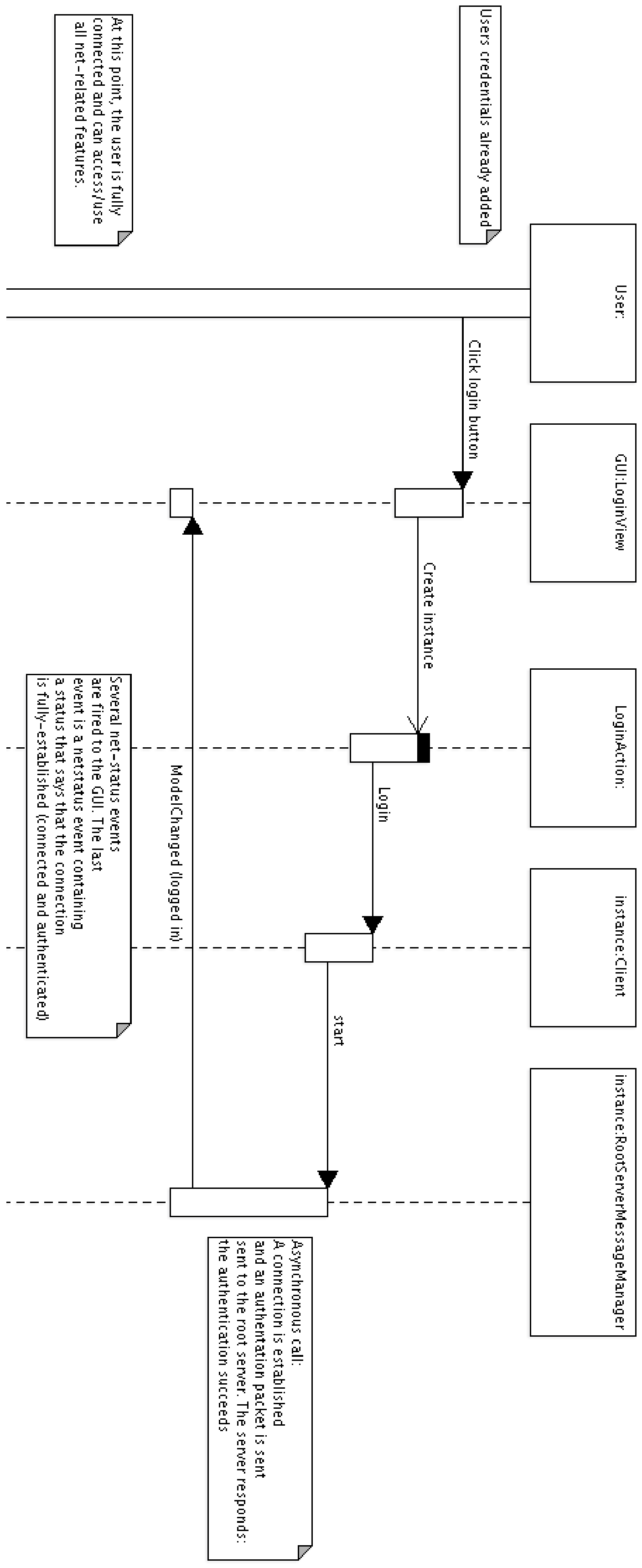
Here is the initial (rough) design for the dedicated server:



This is the actual implementation:


AbstractMessageManager
Processes messages from the clients
Thread
UserConnection
DedicatedMessageManager
Allows command-line control of the Server
DedicatedServerController
0..*
ConnectedUsersManager
Maintains the list of connected Clients
DedServUserConnManager
DedicatedServer
ConversationManager
Maintains the list of Conversations on this Server
Accepts new client connections
ChatRoomListingDistributor
RootServerConnection
Responds to incoming client requests for chat-room listings
Maintains the connection to the Root Server, for authenticating clients

Here is a sequence diagram to show the interaction within the client models; this sequence illustrates what happens when a user logs into the root-server:


User:
GUI:LoginView
LoginAction:
instance:Client
instance:RootServerMessageManager
Users credentials already added
Click login button
Create instance
Login
start
ModelChanged (logged in)
At this point, the user is fully connected and can access/use all net-related features.
Several net-status events are fired to the GUI. The last event is a netstatus event containing a status that says that the connection is fully-established (connected and authenticated)
Asynchronous call:
A connection is established and an authentation packet is sent sent to the root server. The server responds: the authentication succeeds

Online Communication Sub-system

Purpose

This section describes the important design aspects of the sub-system that provides online communication between two or more users. The online communication sub-system addresses requirements 4.x.

Caveat

The online communications is specifically talking about supporting conversations between two or more users. When I talk about conversations, I talk about them in an abstract manor; our system is flexible and can support more than one type of conversation. We plan to implement two: A public conversation which can be also be referred to as a chatroom, and a private conversation which is a conversation that users must be invited in order to join.

General Priorities

Decisions for this sub-system are made based on the following priorities (most important first):

5. Maintainability
6. Simplicity
7. Usability
8. Reliability
9. Security
10. Efficiency

Outline of design

We decide how conversations will be supported using a client-server architecture.

Major Design Issues

Issue 1:
How should conversations be supported by our network-sub system?

Option 1.1:
Host all conversations on the Root Server.
This option would be very simple, but its very naive; the root-server's network and host would be the bottle-neck for performance; all conversation traffic would have to be routed through the root server (since not using peer-to-peer). The reliability is rather poor; It wouldn't take many clients to connect to the rootserver for the clients network-features performance to drop and all of the rootservers' host's resources to be fully consumed (and hence hang or crash).

Option 1.2:
Host public conversations on the dedicated server, but private conversations on the root server.
By hosting the public conversations on the dedicated server the load-factor on both the root-servers host and it's network would be reduced and hence provide a solution that enables adequate performance. This design would introduce more complexity into the net-subsystem (instead of just having one server-the root server) thus require more time to implement. This solution is also more prone to security breaches; clients still must be authenticated when connecting to dedicated servers so more authentication protocols are needed.

Option 1.3:
Host public and private conversations on the dedicated server.
This option reduces the load-factor on both the root-servers host and it's network such that it has a better performance than option 1.2. The trade-off being that it would become more complex than option 1.2.

Decision 1:
1.2 because it not too complex for this project (where as option 1.3 looks as if it could be a little too time-consuming to implement) and its more reliable than option 1.1: it reduces chance of failure; however underlying architecture is still prone to a central-point-of-failure. Another reason why we choose 1.2 over 1.3 is because it allows private conversations between clients that don't have to be connected to the same dedicated server in order to chat, they can chat via the root server which they will always be connected to.

Issue 2:
How should the listings of available chat-rooms be made available to clients?

Option 2.1:
The Root Server queries each Dedicated Server, and maintains its own list of all chat-rooms, which it provides to clients upon request.

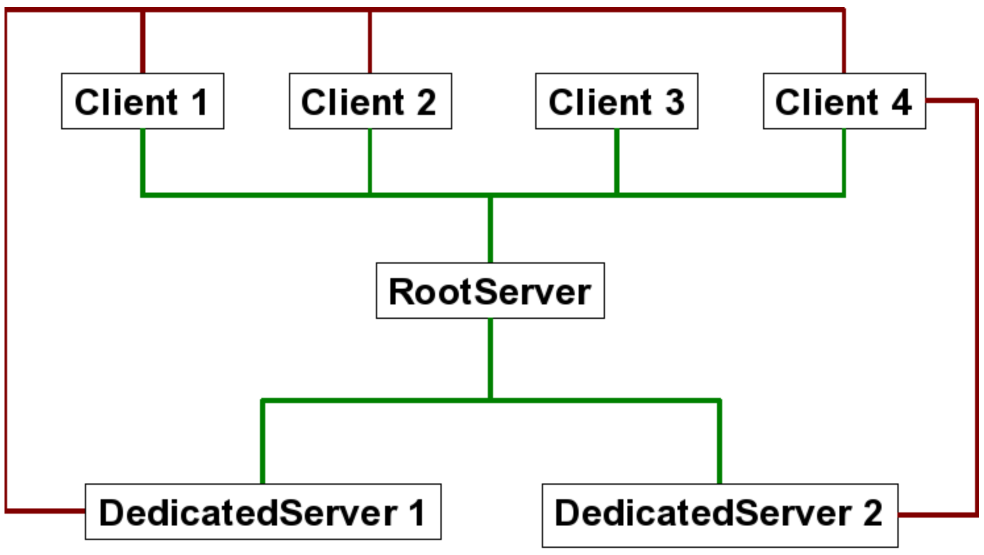
Option 2.2:
Let the clients request a listing from each Dedicated Server individually, and process it client-side.

Decision 2:
2.2 was decided to be a generally more efficient method. It reduces the amount of processing which the Root Server must perform by offloading work to the client. It also reduces memory usage, as the list does not need to be maintained on the Root Server. This is important with large numbers of Dedicated Servers and Chat Rooms.
While there is slightly more overhead in requiring each client to ask each Dedicated Server individually, the reduced load on the Root Server is considered more important. Furthermore, the use of UDP also helps alleviate the communication overhead.

Details of design

Dedicated servers can run on separate networks to each other and the root-server (as long as they can connect to the root server and allow clients to connect to them). They are designed so that they can be used by anyone (users, companies etc...) so that anyone can host their own chatrooms.

Here is a graphical representation of four clients logged-in to the root server, it also shows two dedicated present (available for connecting to and joining chatrooms):



🟩 = TCP Connection

🟥 = TCP Connection and Connectionless UDP Channel

The TCP connections between the dedicated server and the root server not only allows the dedicated servers to advertise themselves as an available dedicated-server on the Jammin network but the connection is also used for forwarding authentication packets from connecting clients so that the root-servers user-account manager can tell the dedicated servers that the a connecting client 's credentials is legit and is currently connected.

The UDP communication channels between the dedicated servers and clients are connectionless. When a client wants a list of chatrooms from dedicated server they query them via UDP; no connections are established. This is for the dedicated servers and clients benefit because it is a many to many relationship – by avoiding connection establishment and teardowns a lot of load is reduced.

The TCP connections between the dedicated server and clients only needs to be established if a client wants to join a chatroom. These connections are used for sending conversation packets and various conversation-control packets.

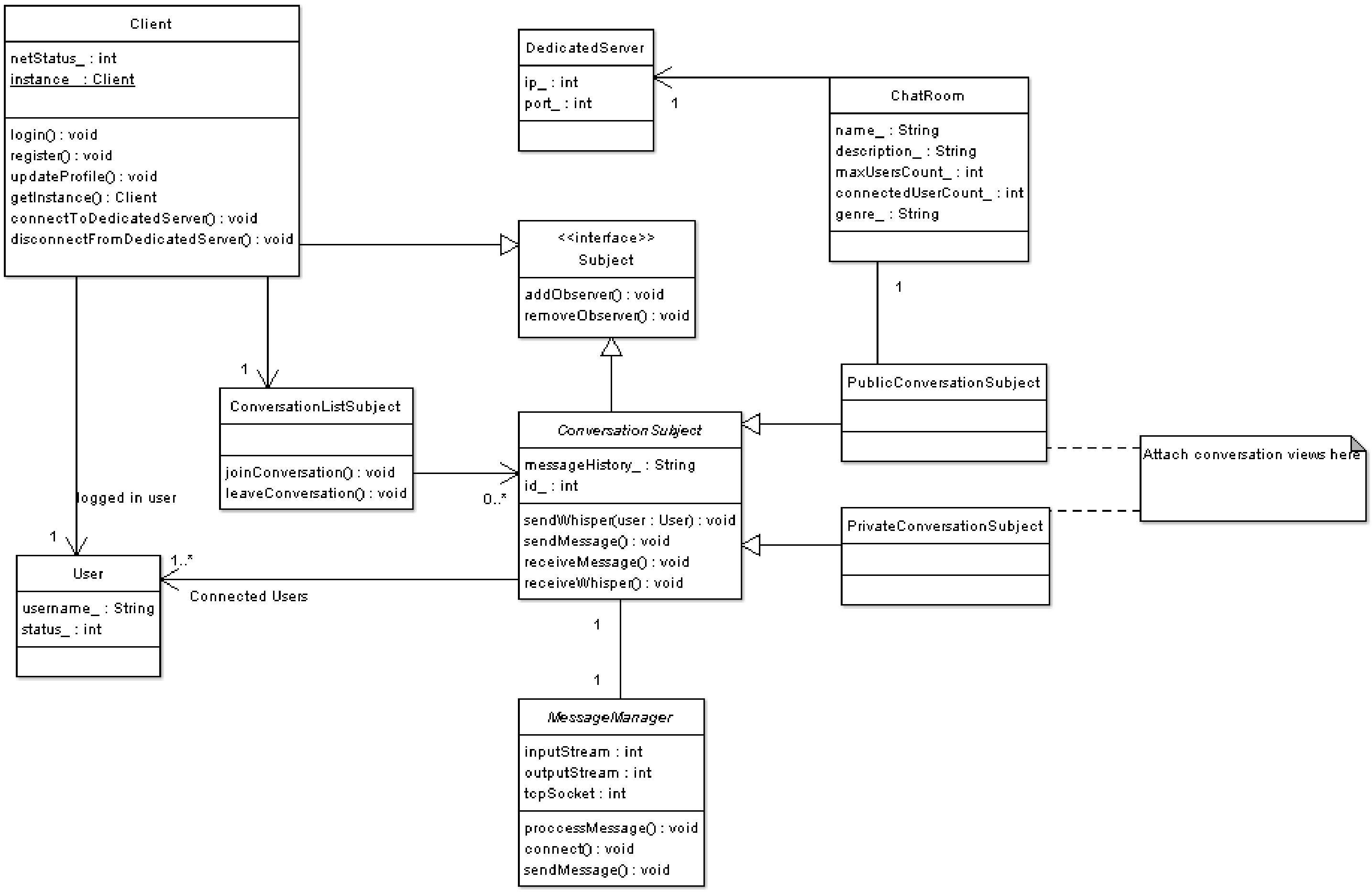
Protocol Details

When a user connects to a dedicated server that client must be authenticated. The client must send the first packet as an authentication packet. The dedicated server forwards this authentication packet to the root server to ensure that the connecting client is legit. The dedicated server forwards the response (also taking note of the response) to the client. From the clients perspective the authentication process looks the same as connecting to the root server.

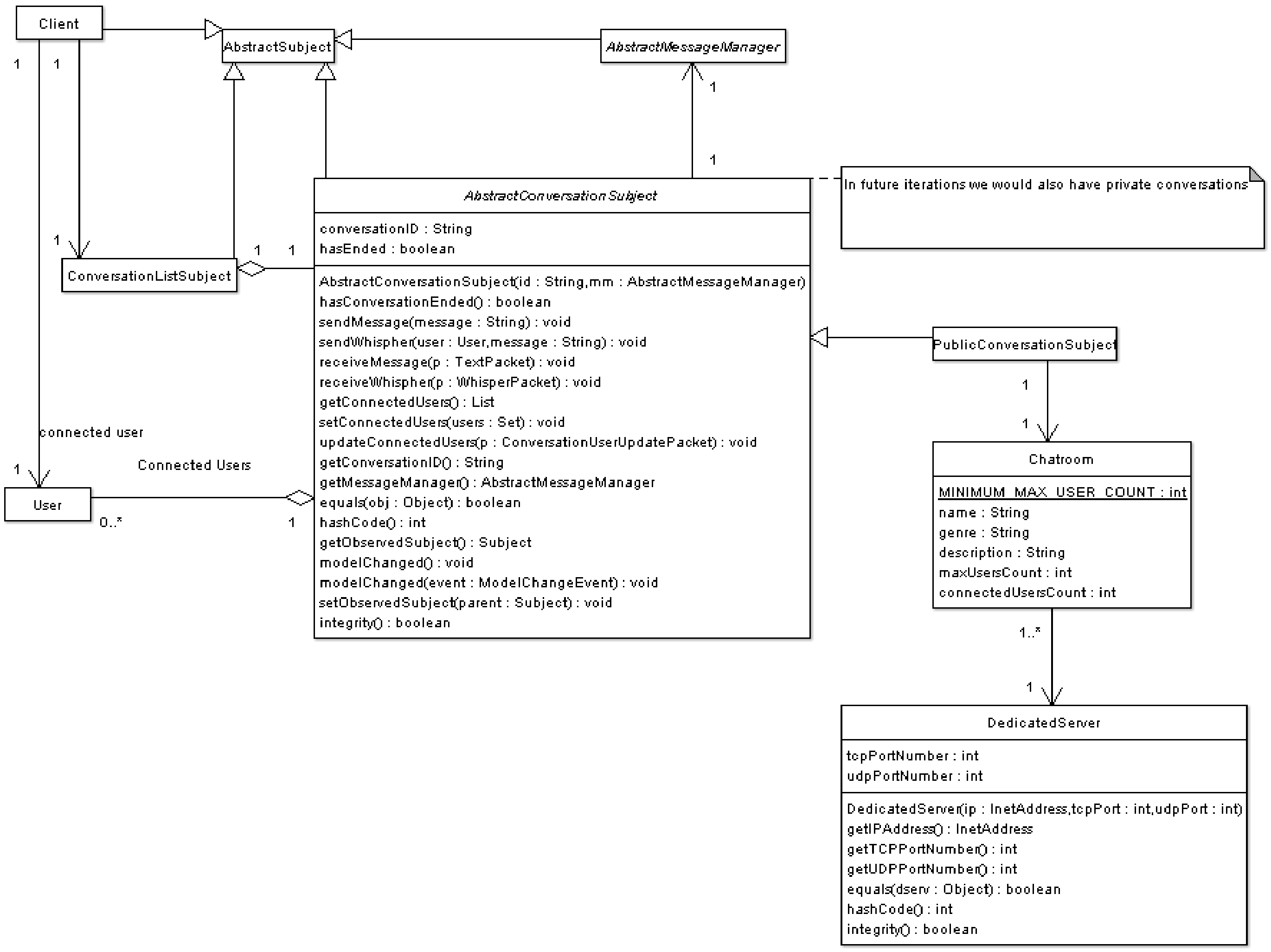
For a client to join a conversation the client must send a join-request packet to the hosting server (i.e. A root or dedicated server). The hosting server sends a type of success or failure response to the join request. If its a success the user can now send and receive conversation packets for the joined conversation. To leave a conversation a user can either disconnect from the hosting server or if they want to stay connected (for example they might be joined in other conversations hosted by the same server) then they send a leave-packet.

Online Communication Sub-system UML Diagrams

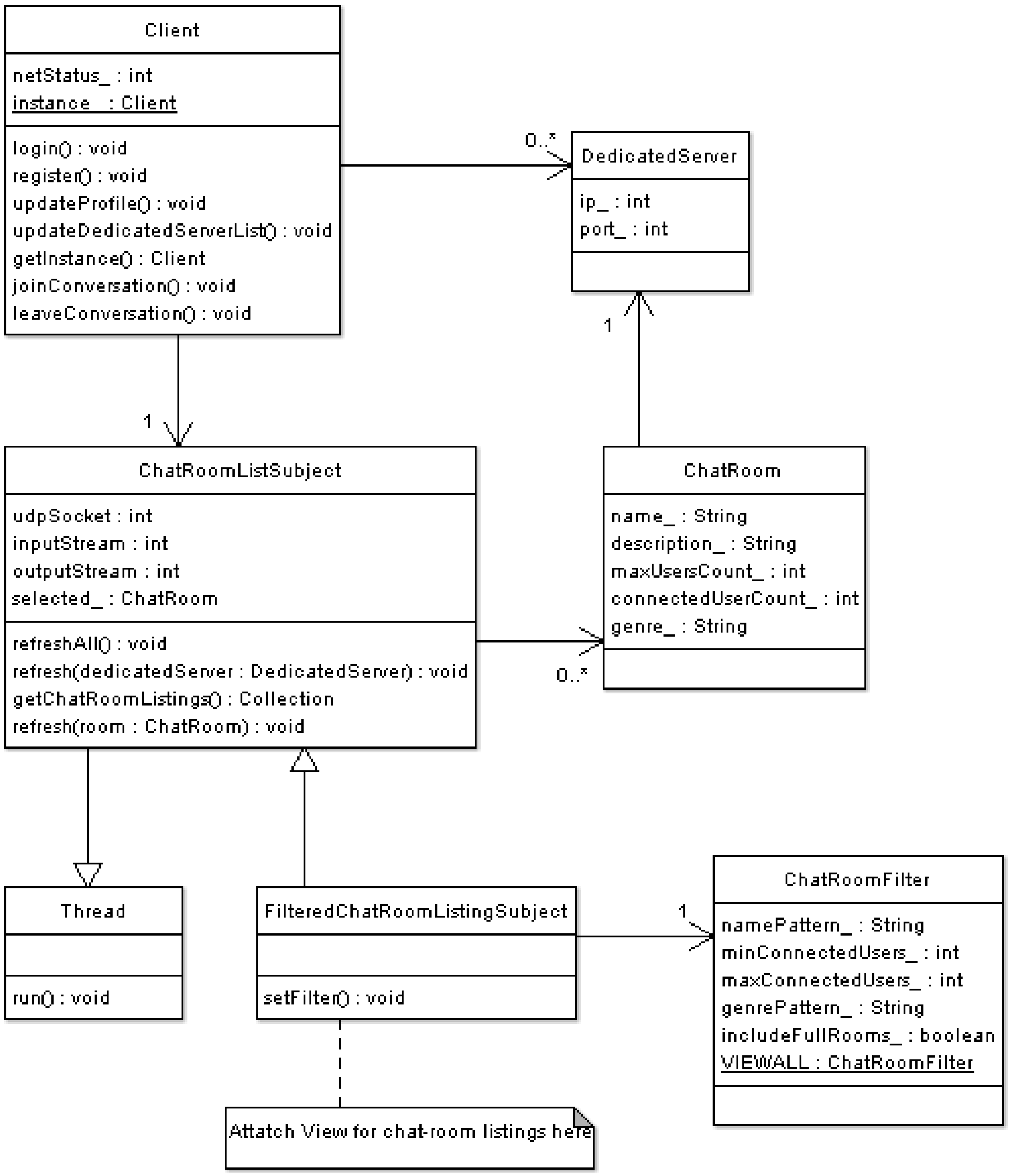
This is a class-diagram of the conversation-models for the Client that was designed as a reference for directing the implementation:



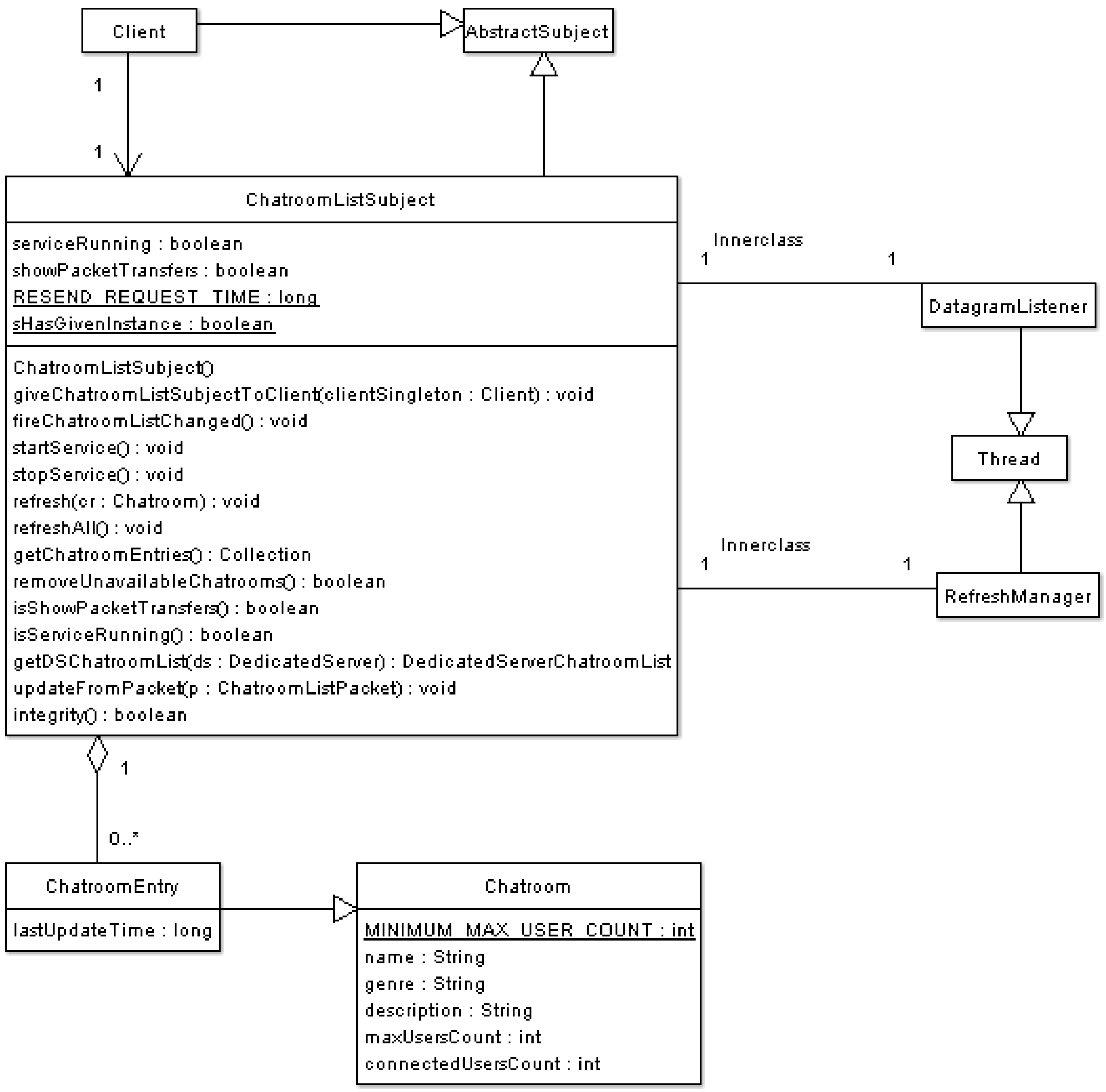
This the actual class diagram for the conversations:


Client
AbstractSubject
AbstractMessageManager
ConversationListSubject
User
connected user
Connected Users
AbstractConversationSubject
conversationID : String
hasEnded : boolean
AbstractConversationSubject(id : String,mm : AbstractMessageManager)
hasConversationEnded() : boolean
sendMessage(message : String) : void
sendWhispher(user : User,message : String) : void
receiveMessage(p : TextPacket) : void
receiveWhispher(p : WhisperPacket) : void
getConnectedUsers() : List
setConnectedUsers(users : Set) : void
updateConnectedUsers(p : ConversationUserUpdatePacket) : void
getConversationID() : String
getMessageManager() : AbstractMessageManager
equals(obj : Object) : boolean
hashCode() : int
getObservedSubject() : Subject
modelChanged() : void
modelChanged(event : ModelChangeEvent) : void
setObservedSubject(parent : Subject) : void
integrity() : boolean
In future iterations we would also have private conversations
PublicConversationSubject
Chatroom
MINIMUM_MAX_USER_COUNT : int
name : String
genre : String
description : String
maxUsersCount : int
connectedUsersCount : int
DedicatedServer
tcpPortNumber : int
udpPortNumber : int
DedicatedServer(ip : InetAddress,tcpPort : int,udpPort : int)
getIPAddress() : InetAddress
getTCPPortNumber() : int
getUDPPortNumber() : int
equals(dserv : Object) : boolean
hashCode() : int
integrity() : boolean
1
1
1
1
1
1
1
1
0..*
1
1
1
1..*
1

Here is the initial class diagram for the chatroom-listing models for the client:

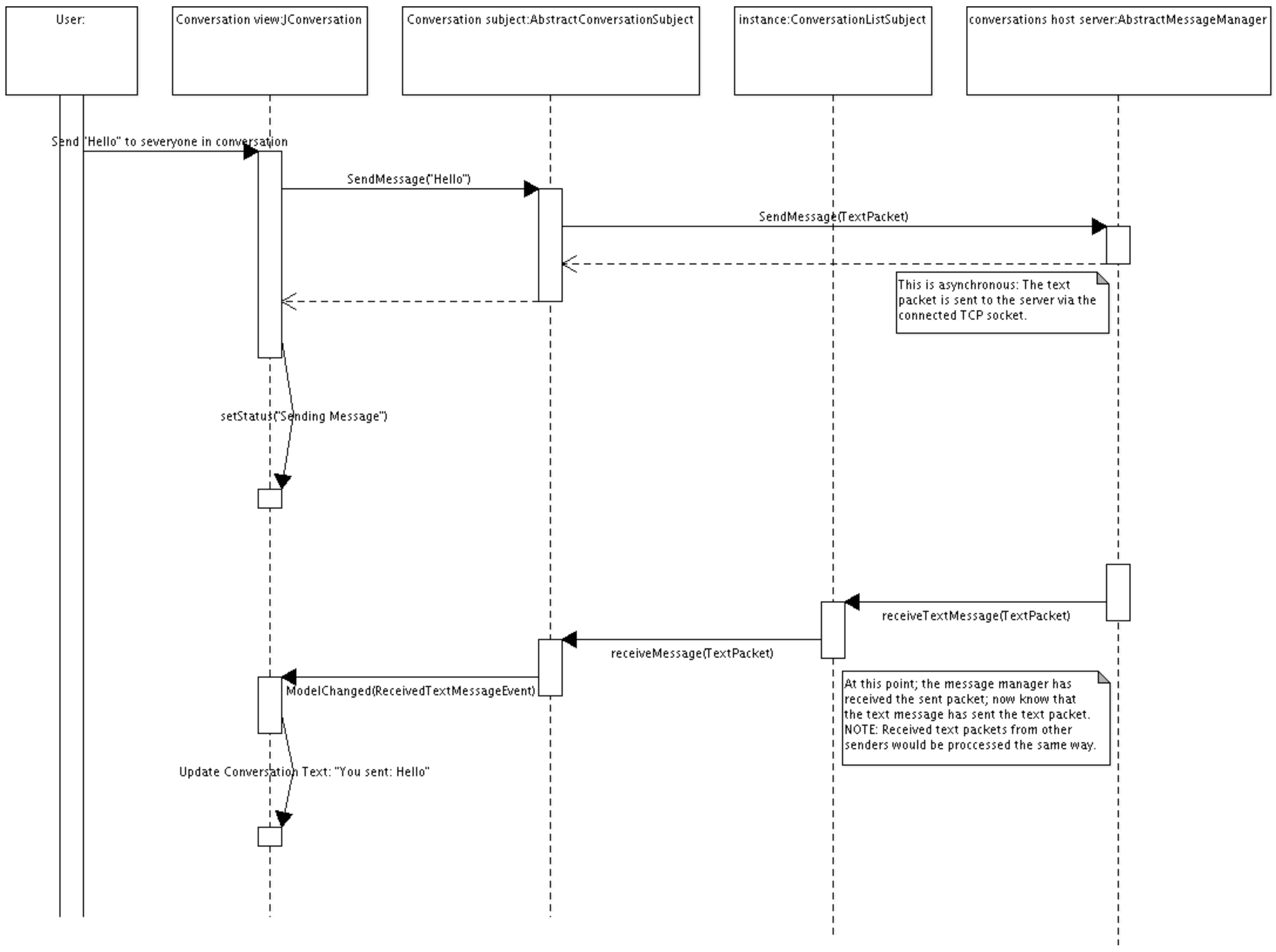


This is the actual class diagrams for the chatroom-listing models:



Client
AbstractSubject
1
1
ChatroomListSubject
serviceRunning : boolean
showPacketTransfers : boolean
RESEND_REQUEST_TIME : long
sHasGivenInstance : boolean
ChatroomListSubject()
giveChatroomListSubjectToClient(clientSingleton : Client) : void
fireChatroomListChanged() : void
startService() : void
stopService() : void
refresh(cr : Chatroom) : void
refreshAll() : void
getChatroomEntries() : Collection
removeUnavailableChatrooms() : boolean
isShowPacketTransfers() : boolean
isServiceRunning() : boolean
getDSChatroomList(ds : DedicatedServer) : DedicatedServerChatroomList
updateFromPacket(p : ChatroomListPacket) : void
integrity() : boolean
Innerclass
1
1
DatagramListener
Thread
Innerclass
1
1
RefreshManager
1
0..*
ChatroomEntry
lastUpdateTime : long
Chatroom
MINIMUM_MAX_USER_COUNT : int
name : String
genre : String
description : String
maxUsersCount : int
connectedUsersCount : int

Here is a sequence diagram to show the interaction within the client models; this sequence illustrates what happens when a user wants to send a public message to other users in a conversation. Note that the user is already connected and joined to the conversation:


User:
Conversation view:JConversation
Conversation subject:AbstractConversationSubject
instance:ConversationListSubject
conversations host server:AbstractMessageManager
Send "Hello" to severyone in conversation
SendMessage("Hello")
SendMessage(TextPacket)
This is asynchronous: The text packet is sent to the server via the connected TCP socket.
setStatus("Sending Message")
receiveTextMessage(TextPacket)
receiveMessage(TextPacket)
ModelChanged(ReceivedTextMessageEvent)
At this point; the message manager has received the sent packet; now know that the text message has sent the text packet. NOTE: Received text packets from other senders would be proccessed the same way.
Update Conversation Text: "You sent: Hello"

Sharing Sub-system

Purpose

This section describes the important design aspects of the sub-system that provides sharing compositions between users via the Jammin network. The sharing sub-system addresses requirements 6.x.

General Priorities

Decisions for this sub-system are made based on the following priorities (most important first):

1. Reliability
2. Maintainability
3. Simplicity
4. Consistency (with rest of systems core architecture).
5. Security
6. Efficiency

Outline of design

We decide on what sort of connection should be used for transmitting compositions for in order to meet requirements 6.x. It briefly outlines the protocol used for transferring compositions from an uploader to a downloader.

Major Design Issues

Issue 1:
What connection should composition transfers be sent from? (There are limited choices since we have chosen a client-server architecture).

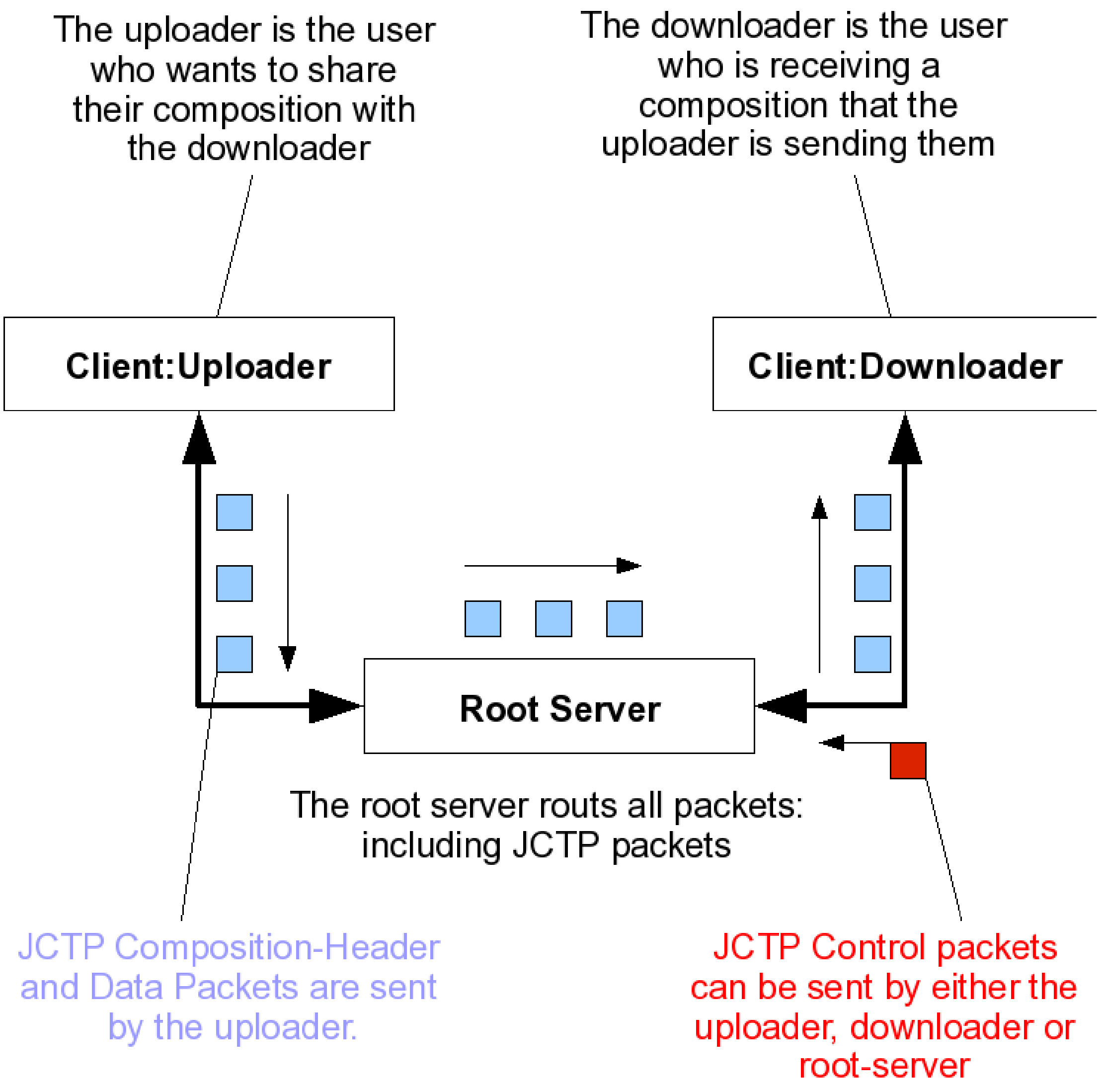
Option 1.1:
Via the Root Server TCP connection: The uploaders sends compositions to the root server to be re-routed to the downloader. The option would put a lot of stress on the root-servers' networks resources; especially when there would be a lot of file transfers occurring at the same time. This design however is very simplistic because we know that all logged-in clients will always have a connection with the root server so there wouldn't need to be any work involved in ensuring that both-clients are connected to the same server that will rout the compositions.

Option 1.2:
Via a dedicated server TCP connection. The uploaders sends compositions to a root dedicated to be re-routed to the downloader. Both clients must establish a connection before the transfer. This option dramatically reduces the load-factor on the root servers host and network; thus improving efficiency. The trade off is that it would be more complex.

Decision 1:
1.1 due to simplicity. Ideally we would have direct TCP connections between clients for file transfers but this would resemble a peer-to-peer network; we have chosen a client-server approach so this is not possible. Consequently we have no choice but to choose an option that would not suit a commercial standard (i.e. Wouldn't handle a large amount of users). We made an early design decision for the network to be simple and straight-forward so by taking that line we chose option 1.1 over 1.2. option 1.2 would improve performance but still wouldn't be adequate for a commercial standard, it just adds complexity which defies one of our main priorities (simplicity).

Details of design
The compositions are sent via the root-server connection in a series of packets using a protocol called JCTP (Jammin Composition Transfer Protocol) which is a basic protocol that is designed to run over TCP.

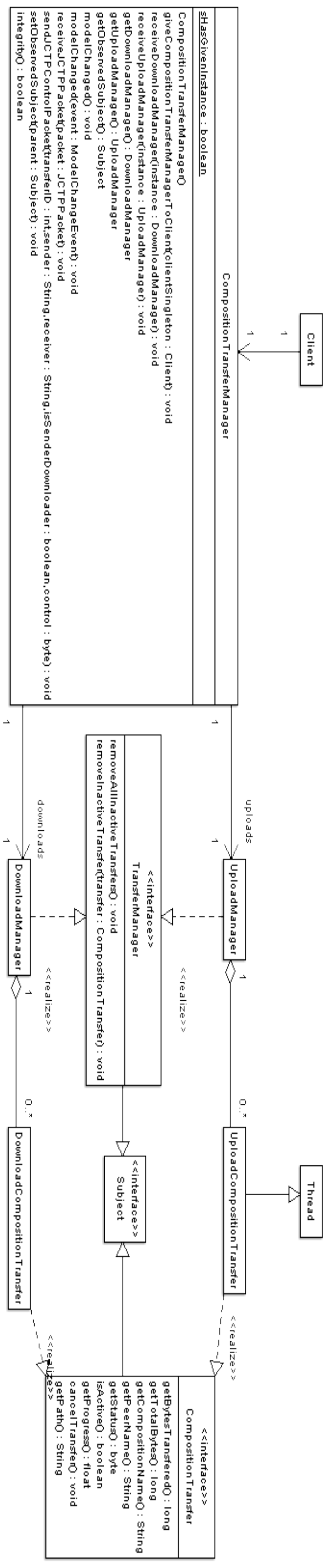
Protocol Details
JCTP is a very simple protocol. Here is a graphical overview of JCTP:


The uploader is the user who wants to share their composition with the downloader
The downloader is the user who is receiving a composition that the uploader is sending them
Client:Uploader
Client:Downloader
Root Server
The root server routs all packets: including JCTP packets
JCTP Composition-Header and Data Packets are sent by the uploader.
JCTP Control packets can be sent by either the uploader, downloader or root-server

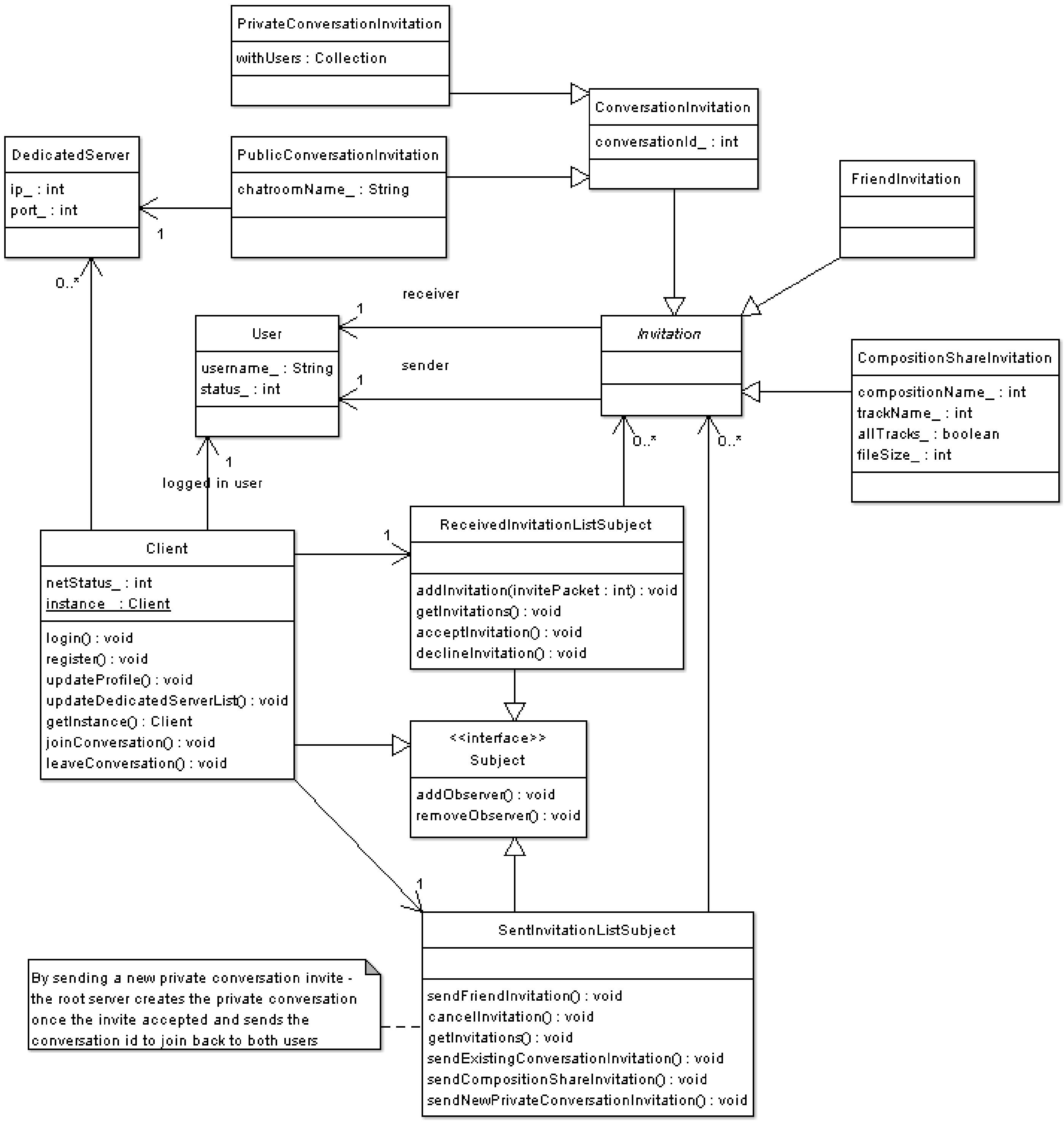
Initially, the uploader sends a composition-header packet to the downloader (who is always listening for header packets) via the root server, the root-server creates an outstanding-connection between the two users so if one of the users disconnect during the transfer the other user is notified that the disconnected user lost their connection (by the root-server sending a JCTP “lost connection” control packet). The uploader then sends the composition file in separate ordered data-chunks that are encapsulated in JCTP data packets. The downloader expects these and writes them to a temp file as soon as a data-packet is received. Once the last data-packet is sent then the downloader ends the transferred (the downloader knows that transfer is complete by counting the downloaded bytes and matching the amount with the JCTP Header-packets file size member. The root-server doesn't count the sent bytes because it would impose too much load factor on the root server; so the uploader sends a JCTP “finished” control packet to the root-server to tell it that the transfer is finished (it then does not need to keep track of the transfer anymore). During the transfer both the uploader and downloader has the option of canceling the transfer by sending a JCTP “cancel” control packet; when sent, the root-server detects this and stops tracking the transfer and routes the packet to the other user who then ends the transfer.

Sharing Sub-system UML Diagrams

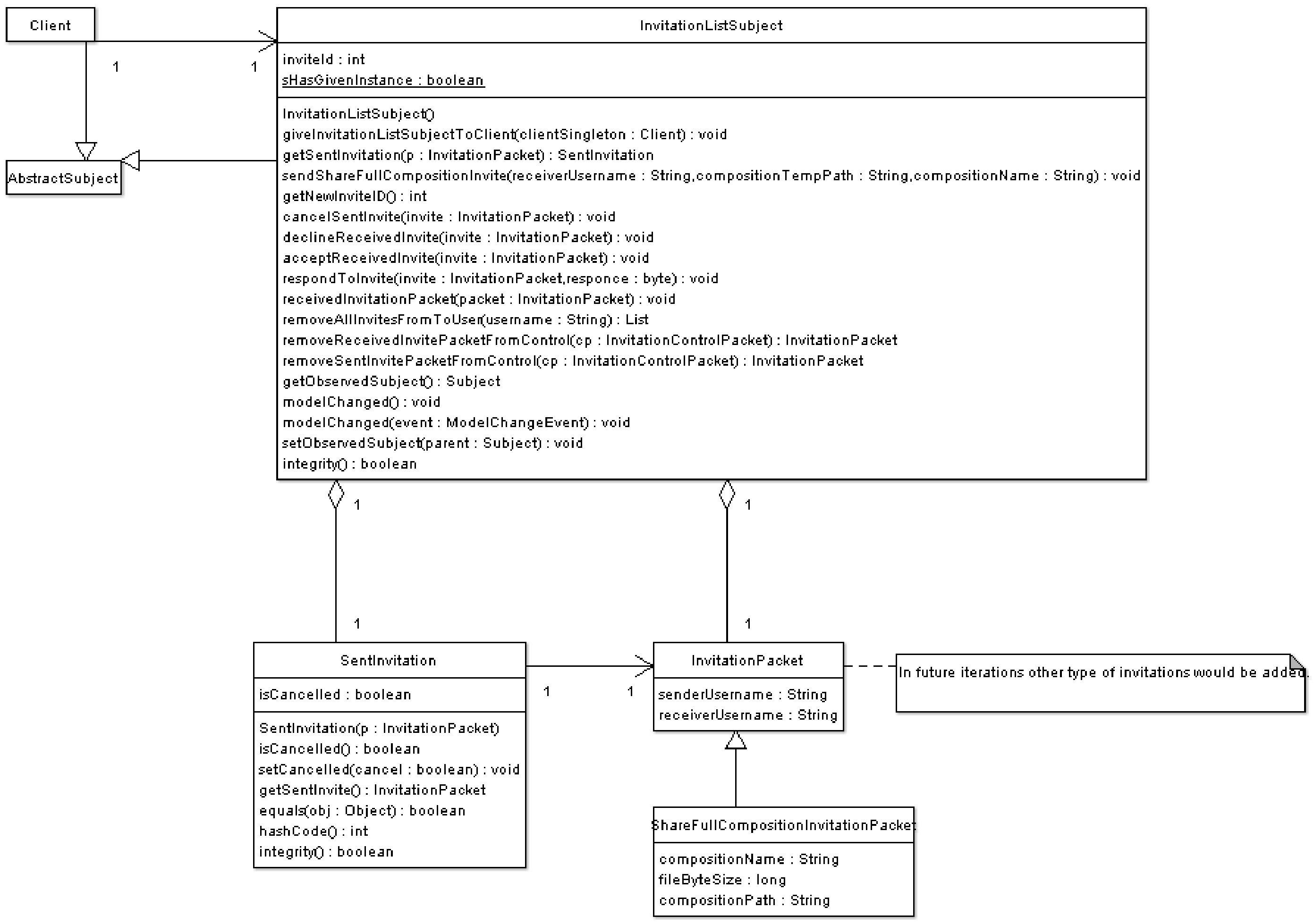
This is the class diagrams for the client transfer-models. Note that the server-side classes remain the same.


Client
1
1
CompositionTransferManager
isHasGivenInstance : boolean
CompositionTransferManager()
giveCompositionTransferManagerToClient(clientSingleton : Client) : void
receiveDownloadManager(instance : DownloadManager) : void
receiveUploadManager(instance : UploadManager) : void
getDownloadManager() : DownloadManager
getUploadManager() : UploadManager
getObservedSubject() : Subject
modelChanged() : void
modelChanged(event : ModelChangeEvent) : void
receiveJCTPPacket(packet : JCTPPacket) : void
sendJCTPControlPacket(transferID : int,sender : String,receiver : String,isSenderDownloader : boolean,control : byte) : void
setObservedSubject(parent : Subject) : void
integrity() : boolean
1
uploads
1
UploadManager
1
downloads
1
DownloadManager
<<interface>>
TransferManager
removeAllInactiveTransfers() : void
removeInactiveTransfer(transfer : CompositionTransfer) : void
<<realize>>
<<realize>>
1
1
0..*
0..*
UploadCompositionTransfer
Thread
<<interface>>
Subject
DownloadCompositionTransfer
<<realize>>
<<realize>>
<<interface>>
CompositionTransfer
getBytesTransfered() : long
getTotalBytes() : long
getCompositionName() : String
getPeerName() : String
getStatus() : byte
isActive() : boolean
getProgress() : float
cancelTransfer() : void
getPath() : String

Invitations were used for inviting users to download a composition; we wanted many different types of invitations so we designed the invitation models such that it would be easy to implement new types of invites such as invites to a conversation.
This is the initial design for invitations:


PrivateConversationInvitation
withUsers : Collection
ConversationInvitation
conversationId_ : int
DedicatedServer
ip_ : int
port_ : int
PublicConversationInvitation
chatroomName_ : String
1
FriendInvitation
0..*
receiver
1
User
username_ : String
status_ : int
Invitation
sender
1
CompositionShareInvitation
compositionName_ : int
trackName_ : int
allTracks_ : boolean
fileSize_ : int
0..*
0..*
1
logged in user
Client
netStatus_ : int
instance_ : Client
login() : void
register() : void
updateProfile() : void
updateDedicatedServerList() : void
getInstance() : Client
joinConversation() : void
leaveConversation() : void
1
ReceivedInvitationListSubject
addInvitation(invitePacket : int) : void
getInvitations() : void
acceptInvitation() : void
declineInvitation() : void
<<interface>>
Subject
addObserver() : void
removeObserver() : void
1
SentInvitationListSubject
sendFriendInvitation() : void
cancelInvitation() : void
getInvitations() : void
sendExistingConversationInvitation() : void
sendCompositionShareInvitation() : void
sendNewPrivateConversationInvitation() : void
By sending a new private conversation invite - the root server creates the private conversation once the invite accepted and sends the conversation id to join back to both users

This is the actual class-diagram for the invitation client models:


Client
1
1
AbstractSubject
InvitationListSubject
inviteId : int
sHasGivenInstance : boolean
InvitationListSubject()
giveInvitationListSubjectToClient(clientSingleton : Client) : void
getSentInvitation(p : InvitationPacket) : SentInvitation
sendShareFullCompositionInvite(receiverUsername : String,compositionTempPath : String,compositionName : String) : void
getNewInviteID() : int
cancelSentInvite(invite : InvitationPacket) : void
declineReceivedInvite(invite : InvitationPacket) : void
acceptReceivedInvite(invite : InvitationPacket) : void
respondToInvite(invite : InvitationPacket,responce : byte) : void
receivedInvitationPacket(packet : InvitationPacket) : void
removeAllInvitesFromToUser(username : String) : List
removeReceivedInvitePacketFromControl(cp : InvitationControlPacket) : InvitationPacket
removeSentInvitePacketFromControl(cp : InvitationControlPacket) : InvitationPacket
getObservedSubject() : Subject
modelChanged() : void
modelChanged(event : ModelChangeEvent) : void
setObservedSubject(parent : Subject) : void
integrity() : boolean
1
1
1
1
SentInvitation
isCancelled : boolean
SentInvitation(p : InvitationPacket)
isCancelled() : boolean
setCancelled(cancel : boolean) : void
getSentInvite() : InvitationPacket
equals(obj : Object) : boolean
hashCode() : int
integrity() : boolean
1
1
InvitationPacket
senderUsername : String
receiverUsername : String
In future iterations other type of invitations would be added.
ShareFullCompositionInvitationPacket
compositionName : String
fileByteSize : long
compositionPath : String