

Cornell University

M.Eng. Project Report

OpenComm Design Documentation & Developer Handover 2011 - 2012

Rahul Arora, *Computer Science Spring 2012*

Najla Elmachoub, *Engineering Management Fall 2012*

Flavian Hautbois, *Computer Science, Centrale Paris*

Vinay Maloo, *Computer Science Spring 2012*

Risa Naka, *Computer Science Fall 2012*

Jonathan Pullano, *Computer Science Spring 2012*

Graeme Bailey, *Computer Science Department*

Introduction

OpenComm is an independent research/project team under the department of computer science and advised by Professor Graeme Bailey. The team is developing an application for advanced audio-conferencing system for Android smartphone devices. Prior to this academic year, the team primarily had a research focus. This year, the OpenComm team has continued implementing the findings done by members of previous years while adding human usability. They have also focused on the packaging and professionalism of the application. A large portion of this project is focused on group and individual development of soft skills, understanding of the software development cycle, and development of technical skills, including programming and understanding different technologies. Throughout this paper, we will discuss both the team and the technical nature of our application.

Goals for the Academic Year

The goals of OpenComm are both interpersonal and technical in nature. Since members are all assigned different roles, their individual goals may vary over the course of the semesters. The goals are grouped into categories and described below.

Personal goals:

- To learn to work as a team in a manner similar to a small company.
- To take on leadership positions in order to delegate responsibility, understand members' strengths and weaknesses, solicit feedback, and evaluate work.
- To work with people from different fields and backgrounds to complete tasks.
- To enhance communication and organizational skills.
- To keep updated on new technologies, competition, and practices.
- To showcase work in BOOM 2012 and speak to professionals and students in the industry.

Product goals:

- To develop a minimal viable product that includes the necessary features of the application: sound spatialization, side conferences, a user interface, and conference scheduling.
- To create a presentable application that is usable by the public.
- To create a website that directs awareness about the product.
- To develop a business recommendation based on the market and competitors.
- To get familiarized with the software development cycle and Agile methodology.
- To obtain good programming and integration practices.
- To use a repository.

Application

The audio-conferencing system developed by Android has two features not seen in similar applications: sound spatialization and side conversations. The goal is to tackle the problem: how do we make an audio-conference more like a face-to-face conference? In a real conference, if person A has person B on his right and C on his left, then he will hear B on his right and C on his left. Using the OpenComm application, each user is represented by an icon. If A is using the application, he can drag B's icon to his right and C to his left, and he will hear them accordingly in his headphones as if they were in the room with him. This sound is therefore spatialized. The

user can also move icons farther and closer, which causes their volumes to decrease and increase.

In physical conferences, A could whisper to B without C being able to hear. This is not the case in traditional audio-conferencing. However, OpenComm allows the user to make simultaneous side conversations. In this case, A could make a side conversation with B; C would have no idea this occurring, and the two people in the side conversation would still be able to hear everything going on in the main conversation.

The application also has several standard features that make it incredibly useful: the ability to schedule conferences, the creation of user accounts, and contact lists.

Team Structure

OpenComm is led by Risa Naka with four subteams. Each team has a different lead, and management changed between the two semesters. For a complete list of team members and information about them, please see appendix A.

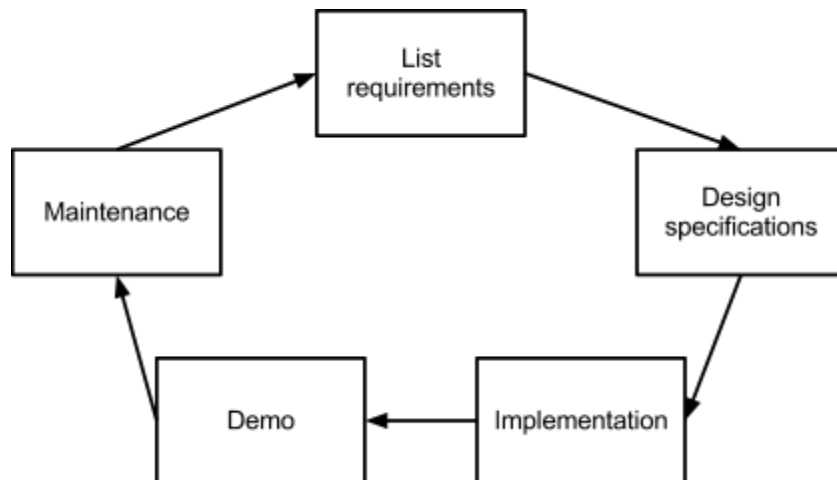
Subteam	Fall 2011	Spring 2012
Design	Najla Elmachtoub, lead Rahul Arora Chris Liu Ashley Sams Joey Triska	Ashley Sams, lead Justin Bard Najla Elmachtoub Vinay Maloo Joey Triska
Publication	Najla Elmachtoub, lead Chris Liu Ashley Sams	Najla Elmachtoub, lead Rahul Arora Brandon Frankel Natalie Lin
Front-End Development	Nora Ng-Quinn, lead Rahul Arora Justin Bard Vinay Maloo Crystal Qin	Vinay Maloo, lead Chris Liu Jonathan Pullano Crystal Qin Joey Triska
Back-End Development	Risa Naka, lead Annie Edmundson Kris Kooi Jonathan Pullano	Rahul Arora, lead Flavian Hautbois Kris Kooi Risa Naka Jonathan Pullano

Recruitment

Members were formally recruited in the beginning of the academic year. We advertised the team to the Computer and Information Science departments. Candidates submitted resumes and were asked to interview with previous team members. M.Eng. students were recruited with particular caution, as we made them aware of the workload expected of them. Throughout the year, a few members were added to the team on a case-by-case basis.

Practices

OpenComm uses a modified version of the Agile methodology to develop. This involves going through quick iterations of the software development cycle to create minimal viable products. Thus functionality is limited, but all implemented aspects should be working and should allow for an end-to-end product.



The long-term requirements for OpenComm are established at the beginning of each semester by the advisor and team leads with consent from the team as a whole. The design team sets requirements for the short-term. These short-term periods are usually two-week cycles, with the exception of some one- and three-week cycles. Upon determining the requirements, the design team creates a design specification document to guide the front-end and back-end development teams for the corresponding cycles. These documents are presented to the entire team during weekly meetings. Team members are encouraged to ask questions.

Throughout the cycle, the front-end and back-end teams work on implementation. This is explained below in detail in the section Code Structure.

Upon completion of the cycle, the development teams demo the application during the weekly meetings. The design team verifies that the tasks were completed accurately and that there is nothing lacking. In the case that there is, the development teams must perform maintenance as quickly as possible. If the design team finds something unsuitable, that is in alignment with the design specification, it is their role to formally put it into another specification.

During weekly meetings, team members were also required to discuss critiques of current events related to technology. We frequently did team exercises and brainstorming activities.

Communication and Sharing

Git and GitHub

For source code management, we utilized the Git revision control system through GitHub. Git is a distributed revision control and source code management (SCM) system. A Git repository is created on the local server, whose changes--creation, modification, deletion--do not impact the original repository. The history of the project is stored in a commit object within the repository. Every time a member makes a modification she or he has to commit it. The commit file keeps the author, committer, comment and any parent commits that directly precede it. This snapshot is used to compare the files when merging the local repository with the shared repository.

This year, we tried the subversion-style workflow, in which all developers clone from the shared repository and can push back to the shared repository. For our project, we used GitHub, a web-based hosting service for software development projects that use the Git revision control system.

Basecamp

Basecamp is a web-based project management tool that contains a calendar, message boards, to-do lists, chat rooms, contact list, and file storage. Thanks to its various tools, we were able to contain all communication relevant to our project in one place.

For each meeting, we created an event and added meeting minutes to it at the end of each meeting. If anyone in the team wanted to know what the other teams were doing or wanted to attend a meeting, they simply had to look at the project calendar. Events and messages are viewable by all team members, so there is complete transparency.

A To-Do list was created for each cycle, and each team lead assigned various tasks to their members along with a deadline; whenever a task was assigned to a member, they received an email notification. Throughout the cycle, members could communicate with other members on their tasks as needed. Team leads were able to keep track of its members' progress as the members would check the tasks as completed once they were completed and integrated into the project.

While files were generally stored in GitHub for consistency, some files that we were uncomfortable having public on our Git repository were uploaded to Basecamp.

Design Team

The design team is meant to define the user interface of the application and the direction of the team. During the first semester the team focused more on actual the conference space. The second semester was concentrated on the steps users need to take to be in a conference. This

includes application basics, such as setting up an account and scheduling conferences in advance.

The team met weekly to discuss components to include in design specifications or results from user testing. Before the meeting, all members were assigned design tasks, such as brainstorming or testing, to be completed prior to the meeting. During the meeting results would be discussed, and then the team would decide what to include in the specifications. These decisions would be based on new features to add in the application, adjustments to previous cycles, and lingering tasks. Team members wrote the design specification for developers in a rotation basis so that everyone could write the document at least once during the semester.

Concept images were usually made by the team lead and would then be sent to the design team for feedback before final specs were released to the general team. When ideas were generated but not added to the specifications, they were added to an ongoing backlog list. The complete backlog list can be seen in appendix B.

Iterative Design Process

The design team followed an iterative design process that includes ideation, idea selection, paper-prototyping, implementation, and interviews. OpenComm is ultimately a project in human-computer interaction, so these methods are crucial. It is highly recommended to include students who have taken INFO/COMM 3450: Introduction to Human-Computer Interaction Design, as students in related fields develop empathy and a non-technological view of product creation.

Publication Team

The publication team was responsible for the OpenComm website, technical research papers, business proposals, talking to clients, video demos, and preparation for BOOM 2012. Although the work on each component was largely independent, the team met once a week to discuss new ideas, provide critiques, and to determine next steps.

Website

Our website is hosted at <http://opencomm.github.com/>. The website is meant to be a professional landing page for someone interested in our application. It should also display information about the team. This is not meant to be a place for internal project management. Instead, it should serve as the public face of our product. We also included new promotional material, such as our logo and a video demo. The website has several versions can have been updated. These can be located in the repository.

Website is built using CSS, Javascript and HTML5. It uses Twitter's Bootstrap framework for scaffolding. The link to the Bootstrap's website : <http://twitter.github.com/bootstrap/>
The javascript framework used for our website is jQuery.

Technical Research Papers

Our goal is to have two research papers written. One will be completely technical in nature to be submitted to a human-computer interaction conference, and the other will be for more general audiences. Drafts of these have been written, but this work was put aside since the conference deadlines are generally in the fall and to allow more time for business development. The papers can be located in the repository.

Business Development

In the second half of the year, the team recruited two MBA students to research and discuss commercialization of the product. The pair looked into similar products, possible users, and strategic recommendations. They ultimately suggested that OpenComm seek strategic partnership through companies like Microsoft and Google. A copy of their findings can be found in the repository.

BOOM 2012

The publication team was responsible for BOOM (Bits On Our Mind) preparation. BOOM is an annual showcase for projects in technology that is open to students, faculty, and corporate representatives. Aside from making sure the product worked, we needed a poster, t-shirts, practice presentations, a logo, and an appropriate demo. This year, OpenComm won three awards: Bank of America Merrill Lynch Technology Award, Goldman Sachs Award, and Cisco Innovation Award.

Front-End Development Team

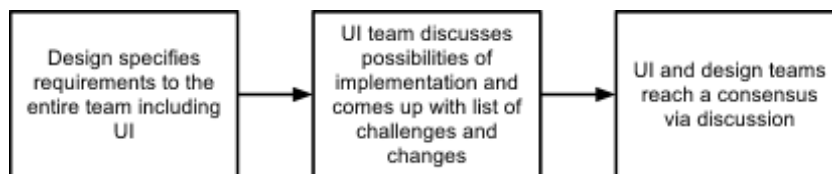
Goals

The user interface (UI) development for this academic year was aimed at serving a two-fold purpose:

1. To meet the requirements of the design specifications as much as possible while maintaining the standards of development allowed within the purview of the application.
2. To develop rich UI features that make the application easily usable, transferring the rigmarole of dealing with complexity to the developer rather than user.
3. To provide varied experience for the UI team members, so as to develop different kinds of features and learn new styles of development.

Meeting Design Requirements

The UI team faces the unique challenge of implementing the rich and elaborate feature designs provided by the design team. Given that the two teams are largely distinct, one does not necessarily see the challenges of each other directly. This turns out to be a good thing, for we are able to isolate *what the application should look like versus what can actually be implemented*. Many a times a compromise needs to be reached between the two teams as to what is possible given the limitations of the framework, and this necessitates new styles of development.



Development of rich UI features transferring complexity to developer and not the user

While using the application one might find that many of the rich features like lasso, ghosting, ability to drag and drop the user into a private chat, etc. are relatively simple to use from a user's perspective but have an inherent level of complexity in development: tracking where the user moves his hand on the touch screen while drawing a continuous lasso, understanding the subtle differences in touch gestures, and making sure that the ghost appears a little above the finger so that the user can see where he drags it amongst others.

One other example is the development of the vertical volume bars within the application, which was not actually used in the final product. The android development framework only supports horizontal slider bars. So when the design specified that vertical bars were requirement, the initial reaction from the front-end team was to change the design to make them horizontal. But looking from a user's perspective, horizontal volume bars for vertically aligned chat rooms is counter-intuitive. A user may stop using application since these small things add up. As a result,

a custom vertical volume design was created by the UI team to accommodate the specification of the design and ease of use for the user.

Providing varied development experience and learning new styles of development

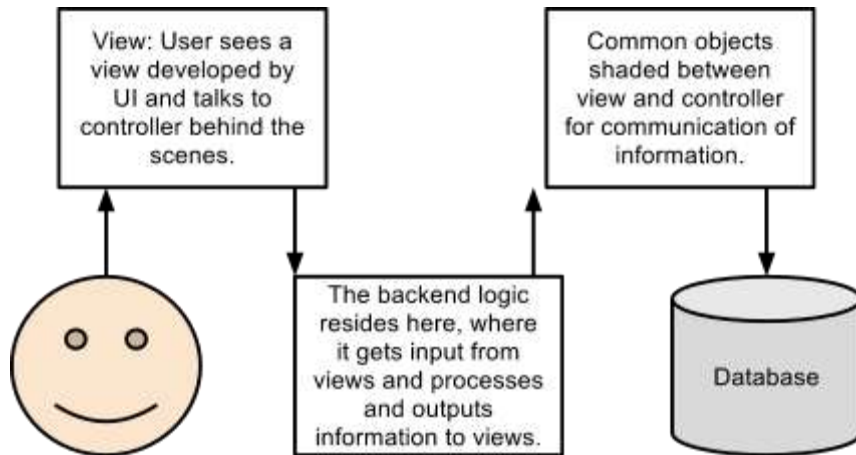
The UI development on the UI side roughly represents three different types of development-- developing UI specific features, developing the modules that integrate the front end technology to the backend, enhancing the structure of development for ease of use for the developers. (the MVC model under practices) Given that there are five front-end development members, every developer at some point has worked with each of these, learning new techniques and styles of along the way. Developers who started with no android development experience are now able to implement design features with relative ease.

Practices

Given the unique structure of our team where we have almost distinct teams for UI, Design, backend and publication development, we needed a framework where it was possible for the different teams to work almost independently yet in a manner that the different features and mechanisms that connect the teams are in place implicitly. Hence the Model-View-Controller (MVC) model of development was chosen. MVC is a design pattern for computer user interfaces that divides an application into three areas of responsibility:

1. **Model**, the domain objects or data structures that represent the application's state.
2. **View**, which observes the state and generates output to the users
3. **Controller**, which translates user input into operations on the model.

Most of the UI development relates to creating the various user views and using the models for passing the created objects to the backend. Most of the backend development happens in the controller, where all the logic relating to talking to the backend APIs and databases happens. The UI and backend meet somewhere within a few functions in the controller, where UI passes in all the information required by the backend to make the specific calls. Hence this model of development lets independent development while still working on a unified integrated framework.



Some of the best UI-specific practices followed by the team are as follows:

1. Model should only contain object specifications and properties, its getters, and its setters. It should not contain any controller or application logic.
2. Every view that is seen on the UI should have its own layout, a corresponding View class and a corresponding Controller class. This isolates the logic for every view, and it is easier to locate and debug within the code.
3. Views should not have any application logic. Each view has a handle to its controller, and every time an event is triggered it calls the appropriate function within the Controller to handle the same.
4. The Controller in turn has a handle to its corresponding view, so that changes to be made on the view can be delegated to it with this handle.

Please see Appendix C for information on specific views.

Back-End Development Team

Goals

We had various goals for this semester and we were successful in achieving most of them:

1. Refactoring network code
2. Moving to a self-hosted server (Openfire) from a shared server (Jabber)
3. Sound spatialization integrated in the final project
4. User account management which involves account creation and forgotten password
5. Scheduling conferences in the future
6. Database plugin for XMPP server which was not possible before (as we were using Jabber)

In all, Backend's team goals for this semester were to make a robust backend which we were able to achieve probably because of moving to a self-hosted server.

Technologies

Android

Android provides a number of building blocks that are reused frequently in our code base. The first is the Activity class, which represents a single task that the user can do. In our application, the user always interacts with a single Activity, which passes data on to subsequent Activities as necessary. Android maintains an Activity stack, which tracks all the Activities that the user has passed through since the start of the application. In this way, we can implement operations such as the back button, which allows us to return to the previous state. Activities in OpenComm include the Login, Dashboard, Conference screens.

It is crucial for blocking or long running operations to execute off of the UI thread, as Android periodically pings this thread to ensure the application has not crashed. The AsyncTask class is used to represent such an operation that needs to be run in the background. Several tasks that involve network communication use AsyncTasks, such as the login and signup button presses. Since AsyncTasks were discovered late in the project, some tasks still need to be refactored to use AsyncTask.

Android Views represent UI elements on the screen, and correspond to the Views discussed in the MVC section. Android also provides a number of different Layouts used in the design of various UI elements. Layouts are containers that hold a number of child Views, and control how those children are rendered on the screen. There are a number of Layouts that we have used frequently in OpenComm. LinearLayouts are used to position their children next to each other, either horizontally or vertically. Relative layouts give you more fine grained control over the positioning of elements by allowing you to specify the position of an element with respect to a previous one. A ScrollView is a wrapper class that may be placed around a layout in order to make its children scrollable.

User input is processed by means of click handlers, which are defined within a view, and call the appropriate method within the views controller. It is the responsibility of the view to extract user input from the UI elements on the screen and then pass this to the controller. In our application, we idiomatically create accessor methods within each view for UI elements that are responsible for user input (i.e. getPasswordBox()), and query them for the relevant data. In particular we use TouchListeners, and LongTouchListeners, for different types of user input. Additionally, in some situations we need to respond to particular types touch events, such as ACTION_MOVE, ACTION_UP, and ACTION_DOWN. These are used in implementation of the lasso, and for dragging user icons around the space.

Finally, Android also provides a few odds and ends:

- TextView: Represents text in the UI
- ImageButton: Used for most buttons in our app, and for UserIcons
- DatePicker: Used for selecting dates during conference creation and editing
- ProgressBar: Used to display "Reconnecting in..." message on timeout

XMPP

Throughout the project, we used the Extensible Messaging and Presence Protocol (XMPP). For a more detailed description, please see Appendix D.

Openfire

Openfire is a real time collaboration (RTC) server licensed under the Open Source Apache License. It uses the only widely adopted open protocol for instant messaging, XMPP (also called Jabber). Openfire is incredibly easy to setup and administer, but offers rock-solid security and performance.

Advantages of using Openfire over Jabber:

1. Web-based administration panel which was missing in Jabber.
2. Support for debugging user info and MUC info
3. Database support for storing messages and user details. It can also be used for fetching information about conferences.
4. Plugin support which makes it more customizable. Database interactivity would have been difficult if we were not able to install custom plugins on the server.
5. LDAP connectivity can be used in case our application is targeted to be used inside a specific organization.
6. Debugging possible on local computer which is not the case with Jabber.
7. Easy installation via:

<http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/install-guide.html>

Network.java

All the hard coded values reside in this class. So, all the information about Openfire server (IP address, port number, hostname, default username, default password) are in Network.java. However, if one has to switch from Openfire to Jabber, they can do so by changing the boolean variable to true.

```
// Currently, to use Openfire
static boolean isJabber = false;
```

Scheduling Conferences

Conferences are scheduled using SchedulingService.java. SchedulingService creates a chat using chatManager when the constructor is called. The chat is used to communicate with the scheduling plugin which is residing on the openfire server.

```
/** Sends conference info to the database. */
public void pushConference(String owner, String date, long start, long end,
                          String recurring, String[] participants){
... }

/** Update a conferences on the database. */
public void updateConference(Conference toUpdate) {
... }
```

```
/** Sends a message to the server asking for conference data. */  
public void pullConferences() {  
... }
```

Scheduling Plugin

The plugin resides at the server but it is customizable and generic to any database. This primarily does pulling of conference data from the server's database and pushing the conference data to the server's database.

Documentation to write a plugin for Openfire:

<http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/plugin-dev-guide.html>

The java files necessary for creating the plugin are in 2011_2012/Apr26May3cycle folder. Steps to build the plugin using the java files are also mentioned in the documentation given above. There are two java classes in this folder, DatabaseService.java and SchedulingPlugin.java. We can use DatabaseService.java as a base for other custom plugins. SchedulingPlugin does the major task of scheduling conferences.

Code Structure

Class Descriptions

Invitations, and Confirmations

In a conference, the owner of the room may invite whoever they wish. However, other users must send an invitation request to the owner. This functionality is not directly supported by XMPP, so it is implemented as follows: A client sends an invitation request, in the form of a message to the MultiUserChat. Each user receives notification that a message was posted to the chat via smack MessageListeners. If they do not own the conference, it is ignored. The owner receives the message, and a ConfirmationView popup is triggered allowing the user to decide whether to accept or decline the request. If the request is accepted, then the user is invited via a smack invite() call. Each client has an InvitationListener which will be notified in the event of an invitation, triggering an InvitationView popup. If the user accepts the invitation, they will be added to the conference, otherwise it is ignored. Kickouts are handled similarly.

Join Leave / Notifications

Smack also handles user presence notifications. Our ParticipantStatusController class implements smack listeners to handle these events. If a user joins or leaves a chat, everyone in the room is notified, so that the UI can display or remove their icon. Users may be connected to XMPP without being logged in to our app. If we detect that a user has suddenly completely disconnected from XMPP, this is likely due to a connectivity problem, and so we show the user as “disconnected” in chat.

Conference List

The Conference List Screen is implemented as an ExpandableListView, which differs in implementation from other more widely used android views. Each item in the list is a custom view called a ConferenceListing. Each conference listing contains a Conference object, which is populated by the backend. These ConferenceListings are then passed to an ExpandableListViewAdaptor, which provides display information to the ExpandableListView while maintaining a loose coupling between the ConferenceListings and the ExpandableListView itself.

Space View

Each space is handled by the MainApplication activity. When a user changes space, a variable called MainApplication.screen is updated which tracks the space the user is currently part of. Each space has an associated SpaceView, which handles rendering of the appropriate UserIcons in the SpaceView's draw function. The lasso mechanism is also handled inside the SpaceView. The lasso is implemented inside the onTouchClickHandler. The lasso maintains a list of points to be rendered onscreen via the draw() function. When the user clicks once in the background, the lasso is cancelled. A sensitivity parameter was added to the lasso in order to maintain correct behavior on phones, since even involuntary finger movement would trigger a move event, making it impossible to cancel the lasso. This sensitivity parameter indicates how

far you must drag your finger before a second point is added to the lasso. Each time a new point is added to the lasso, it tests intersection on screen with each of the `UserIconViews`. This is implemented via an optimized version of the Liang-Barsky algorithm discussed on stackoverflow here: <http://stackoverflow.com/questions/99353/how-to-test-if-a-line-segment-intersects-an-axis-aligned-rectangle-in-2d>

When a user clicks on a lassoed icon, a “ghost” of the `UserIcon` is generated via the `getGhost()` function, which may then be dragged onto a private space. On release, each user associated with a lasso icon is sent an invitation to that private space via the `Smack invite()` function.

Signup

The `UserAccountManager` allows us to register a new user and reset the password in case it has been forgotten. It does not use the API on the Android phone for two reasons: security and feasibility. An Android phone is not secure at all; we have to rely on an external server in order to do sensitive operations, like handling passwords. Moreover, the API we use does not allow for any modification operation, we can only read from the database. We have set up a Virtual Private Server to be our intermediate remote server between the Android application and the OpenFire server. The `UserAccountManager` should also be used to edit a profile, but it has not been implemented yet.

Application side

The User Account Manager

Signature and input of `userChange`

The `UserAccountManager` only provides one public working function, whose name is `userChange`. Its signature is: `public boolean userChange(ArrayList<NameValuePair> nameValuePairs)` where the `nameValuePairs` is a dictionary. The possible values for the keys are:

- ‘userEmail’: the email address
- ‘action’: at the moment ‘forgot’ or add
- ‘password’: the password
- ‘id’: the name of the user (which is not the username) e.g ‘Pr. Mohamed Jackson’

How the code works

`userChange` opens an HTTP connection between the application and the `userChange.php` script on the intermediate server using an `HttpClient` object instantiated as `httpClient` from the Apache library (already included in Android). An instance of an `HttpPost` object called `httpPost` is used to set the target URL and the POST data is set by calling `httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs))`. An `HttpReponse` object is instantiated as `response = httpClient.execute(httpPost)`. It is cast into a string using the private `InputStreamToString` method defined in the same class because the response can be cast into an `InputStream` object.

Error handling

The response is what would be shown on a web browser, but it is not an HTML page, it only consists in a string ‘ok’ or ‘PB1’ for custom problems.

The `UserAccountManager` handles errors in multiple ways, but returns only a boolean value, so it is not possible at the moment to have a custom message describing the error to the user. The error is shown in the logcat, and any possible case should already be handled, except a ‘no

internet connection' type error. If the error is of the type: 'Openfire error. String: <html>...' then the intermediate server is not working properly. If the error looks like: 'OpenFire error. String: [something meaningful]', check this page: [http://www.igniterealtime.org/projects/openfire/plugins/User Service/readme.html](http://www.igniterealtime.org/projects/openfire/plugins/User%20Service/readme.html) to interpret the error. A last case that can happen is a consequence of forgetting to adapt the string to the URL format.

Here is a list of the errors associated to the response:

- 'PB1': There was a problem sending an email;
- 'PB2': Couldn't connect to the OpenFire server;
- 'PB3': The reponse from the OpenFire is unkown (which is very unlikely);
- 'PB4': Wrong parameters for the post;
- 'PB5': The email address is not valid.

Sign Up and Reset Password controllers

The SignUpController handles a registration by:

1. Calling createUser;
2. Starting an AsyncTask passing the values instanciated by createUser to the userChange method.

Please note that the values to be sent have to be URL-friendly. For example, the title, first name and last name are concatenated into one string with '+' signs, e.g 'Pr.+Mohamed+Jackson', where the plus is coded as '%2B'. You don't need to replace the spaces though, but letters like 'é' must be replaced for example (this is common with foreign names). We have not coded anything automatically make a string URL-friendly.

The ResetPasswordController works in the same way but calls validEmail instead of createUser. If you have to use the userChange method in UserManager for another task, you should always call userChange in an AsyncTask. A simple way to do it is to copy-paste the content of the already existing controllers.

Testing

The process of testing these codes was simple, but the actual testing was more painful. The Openfire admin console doesn't show a user's password, but it shows the user's name, so I was changing the name of the user along with the password. If the name had changed, then the password had also changed. I later erased the part changing the name. Testing the registration was more painful as I had to file the whole form several time.

Future work

The Android code to edit an user is not written yet for Openfire, and the data transfer is not secure. Encrypting the information sent by the phone should be useless, because a pirate would have access to the key or to the encryption algorithm. The connection between the two has to be secure, which is not the case at the moment. Some future work should involve working on this security issue.

Server side

Setting up the server

The documentation to set up the server is in the 'server' folder. We chose to install nginx which is a free, open-source HTTP server. We also installed PHP on the server to run our scripts.

The PHP scripts

There are two scripts files: userChange.php and confirmation.php. The first one handles: updating the OpenFire database using the User Service plugin, and sending an email. The value 'edit' for the key 'action' is already handled and it does not send an email. This script is then ready for the implementation of a user edit for Openfire.

Emails

We use a Gmail account to send an email, using a PHP plugin named phpmailer; see <http://phpmailer.worxware.com/>. At the moment the main problem is that the emails go into the spam folder. This has to be addressed by setting up a proper SMTP server on the intermediate server. The body of the emails is very simple, some future work could involve designing a nicer one.

HTTP connection

The script opens an HTTP connection to the Openfire server using the fopen function and updates the information using a get method. You cannot read data using HTTP, which means that it is impossible to get the email address knowing the username (which is the identifier of an user). As a consequence, the username must be deduced from the email address if we want to be able to do a script to reset the password. Thus we defined the username as being the user's email address.

Let us now focus on how to add a user to Openfire. The userChange page receives all the correct information and sends an email containing an url which is: 'servername.extension/confirmation.php?email=[...]&password=[...]&name=[...]' with the email address, password and name encoded with the RSA algorithm (it could be any encryption algorithm with a key) whose key is stored on the server. Then the confirmation.php script decodes those three values and adds them to the Openfire database thanks to the User Service plugin.

Handling the server response

The response we get from the server is a string which is not exactly what we want; it is "<response>[The Openfire response]</response>". We used a regular expression to get rid of the tag, which was enough.

Future work: improving on security

The intermediate server and the Openfire server should run on the same server, and the User Service plugin should restrict the authorized IP addresses to that of the server (Server -> Server Settings -> User Service).

The last security issue concerns the intermediate server and how the critical data (User Service secret key, SMTP server, SMTP password, encryption key) should be stored. A .htpasswd file should be one solution.

Audio

Theory of Sound Spatialization

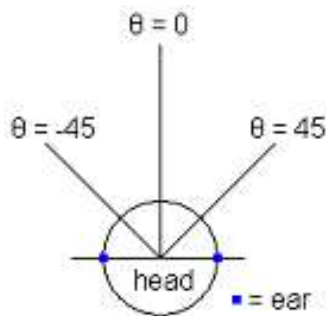
Psychoacoustics is the scientific study of sound perception, specifically the psychological and physiological responses associated with sound. While humans have two ears, they are able to determine the direction and distance of a sound source in a three dimensional (3-D) environment. Sound spatialization is the manipulation of an audio source to give the listener the illusion that the sound source is coming from a distinct point in a virtual 3-D environment. Unfortunately, complete 3-D sound simulation is difficult, as it depends on a lot of factors such as individual physical difference (i.e. ear shape), temperature, humidity, reverberation of the location, and the pitch of the sound. To simplify the environment, we made the following assumptions:

- Radius of a human head: 8.5cm
- Ears are points lying on the opposite ends of the sphere (no outer ears)
- Room temperature is assumed to be 20°C
- Speed of sound is 343.42m/s
- There is no reverberation in the location
- The ear perceives all sound in the same fashion regardless of the pitch

Due to the limit of Dalvik and mobile devices in terms of OS support, CPU speed, and memory amount, we could not fully simulate a 3-D environment. In a typical conference, participants are often sitting and facing each other; thus, we limited the possible range of sound source to the same horizontal plane as the ears and to the front of the user. Humans localize sound on a horizontal plane through interaural time delay (ITD) and volume difference.

Interaural Time Delay (ITD)

ITD is the difference in the time it takes for the two ears to hear the sound. When the angle is defined from the center of the face to the right, it becomes positive to the right side of the nose, and negative to the left side of the nose. For example, when the angle completely lies on a half line from the center of the head to the left ear, becomes -90°.



The interaural time between two ears is derived using the formula :

$$\text{Interaural Time Delay (s)} = \frac{\text{radius}_{\text{head}}}{\text{speed}_{\text{sound}}} \times (\theta + \sin \theta)$$

where θ is expressed in radians.

Negative delay indicates that the left ear first hears the sound.

Volume Difference

Volume difference is the difference in volume level perceived by the two ears. According to the distance law, the sound pressure (volume) emitted by a source is inversely proportional to its distance from the listener.

$$\Delta_{distance} = ITD(s) \times speed_{sound}$$

$$distance_{left} = distance_{center} + \frac{\Delta_{distance}}{2}$$

$$distance_{right} = distance_{center} - \frac{\Delta_{distance}}{2}$$

where $distance_{center}$ is the distance from the sound source to the center of the head

Volume is set for each ear accordingly.

Implementation

Code Structure

VoIP or Voice over IP refers to the protocol or methods involved in sending voice and multimedia communications of Internet Protocol (IP) networks. Jingle is an extension to XMPP that adds P2P session control for multimedia communication. It defines how a software sending messages based on XMPP should send the multimedia.

For our application, we use the implementation of Session Initiation Protocol (SIP) (protocol that defines the creation, modification, and deletion of VoIP sessions) and Real-Time Transport Protocol (RTP) (packet format for delivering audio and video over IP networks) written by the project team in 2010 which is based on the project Sipdroid. Please see Appendix E regarding the details of the audio, RTP, and Jingle protocol.

When the protocol code was created in 2010, there was a class called *MUCBuddy* which included both the information of the User as well as the variables and methods required to implement Jingle. When we integrated the protocol codes into our application, we filtered from *MUCBuddy* the methods and variables relevant to Jingle and created a new class called *JingleController* that was affiliated with an instance of *User*.

In addition, we created a class called *SoundSpatializer* which manipulates the incoming audio source in accordance to its location to spatialize the sound. Both classes are stored in the package *edu.cornell.opencomm.audio*.

Each Jingle connection uses two *AudioTrack* objects. Each object handles one of left or right sides only; the side it is not handling is set to 0.0f. Every audio packet that comes in is processed as follows:

1. Extract audio source from RTP packet and convert to 8kHz 16-bit linear PCM data using G.711
2. Update stereo volume of the two *AudioTrack* objects based on the source location
3. Duplicate the audio source and implement ITD for left and right channel
4. Write the modified audio sources to its corresponding *AudioTrack* objects
5. Play the *AudioTrack* objects (we can write to an *AudioTrack* object while playing)

Determining Location of Sound Source

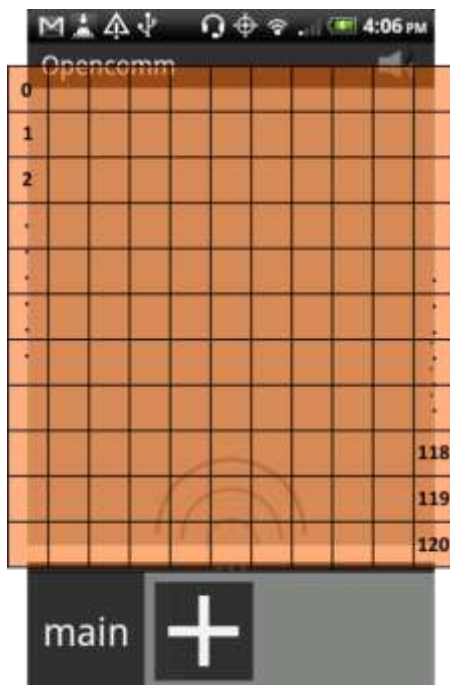
In the application, the source of sound from a user is determined by the location of its icon on the virtual conference space. To alleviate the mobile phone from calculating the interaural time delay and volume difference at every point, we calculated the values beforehand and store them within *SoundSpatializer*. Whenever a user's icon is moved, once the application user stops dragging the icon, the x and y-coordinate of the icon (the center of the icon) is updated as the user's location in the virtual conference space.

If every possible location were precalculated and stored, there would be over 250,000 tuples of $\langle x, y, \text{itd}, \text{volLeft}, \text{volRight} \rangle$, assuming that the coordinates are expressed in integers. Thus, we divided conference space is divided up into 121 different regions, identified as Region 0 – 120. The region covers 110% of the available conference space.

```
/** update region, ITD, and vol based on new position */
public void moveTo(int nx, int ny) {
.... }

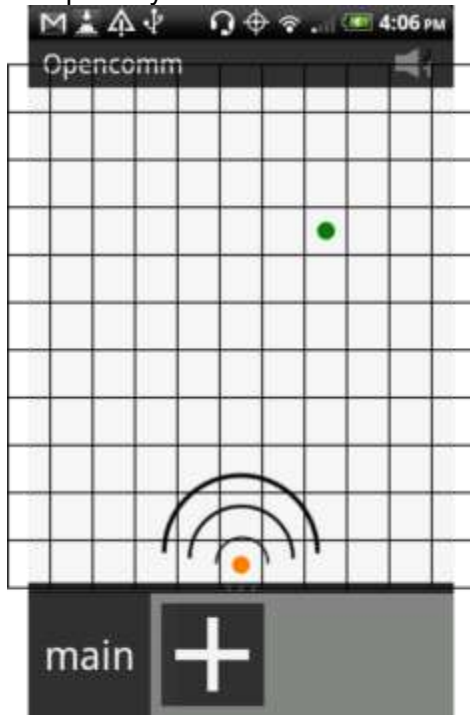
/** private:: set region of sound source based on position */
private void setRegion(int x, int y) {
.... }

/** private:: set interaural time delay (short) and volume [left, right] based on the
region of sound source */
private void setSS() {
.... }
```



The ITD and volume difference for each region is calculated based on the region's center point (green for Region 80) with the primary user's location (orange) as the bottom center edge of the

conference space. The primary user's location is calculated as the origin (0, 0). In the case of a user located in Region 80, the point used to calculate the spatialization variables is (336, 161). The primary user's location is calculated as the origin (0, 0).



Due to user feedback, when converting the virtual conference space into meters, we doubled the value of the x-coordinate to exaggerate the direction of the source.

Calculating ITD and Volume Difference: Example for Region 80

$coordinate_{primaryUser} = (0, 0)$

$coordinate_{userReg80} = (336, 161)$

$Conference\ space\ width = 480px$

$Conference\ space\ height = 537px$

$1px = 0.0025m$

$radius_{head} = 0.085m$

$speed_{sound} = 343.42m$

Sound: 8kHz 16 – bit linear PCM, mono channel (8 shorts/ms)

Coordinate of User at Region 80, Converted with Origin as (0, 0):

$x_{userReg80C} = 2 * (x_{userReg80} - conference\ space\ width/2)$

$y_{userReg80C} = conference\ space\ height - y_{userReg80}$

ITD in shorts

$$\theta(\text{radian}) = \frac{\pi}{2} - \cos^{-1} \left(\frac{x_{\text{userReg80C}}}{\sqrt{(x_{\text{userReg80C}})^2 + (y_{\text{userReg80C}})^2}} \right) = 0.472121982$$

$$ITD(s) = \frac{\text{radius}_{\text{head}}}{\text{speed}_{\text{sound}}} \times (\theta + \sin \theta) = 0.000229417 \text{ seconds}$$

$$ITD(\text{short}) = \left\lfloor \frac{\text{radius}_{\text{head}}}{\text{speed}_{\text{sound}}} \times (\theta + \sin \theta) \times \frac{1000\text{ms}}{1\text{s}} \times \frac{8 \text{ shorts}}{1\text{ms}} \right\rfloor = 1 \text{ short}$$

Volume for Left and Right channel ($0 \leq \text{vol} \leq 1$)

$$\Delta_{\text{distance}} = ITD(s) \times \text{speed}_{\text{sound}} = 0.078786427\text{m}$$

$$\text{distance}_{\text{left}} = \left(\sqrt{(x_{\text{userReg80C}})^2 + (y_{\text{userReg80C}})^2} \right) \times \frac{0.0025\text{m}}{1\text{px}} + \frac{\Delta_{\text{distance}}}{2} = 1.094855198\text{m}$$

$$\text{distance}_{\text{right}} = \left(\sqrt{(x_{\text{userReg80C}})^2 + (y_{\text{userReg80C}})^2} \right) \times \frac{0.0025\text{m}}{1\text{px}} - \frac{\Delta_{\text{distance}}}{2} = 1.016068771\text{m}$$

$$\text{distance}_{\text{min}} = 0.130741\text{m}$$

$$\text{vol}_{\text{left}} = \frac{1}{\text{distance}_{\text{min}} \times \text{distance}_{\text{left}}} = 0.0980$$

$$\text{vol}_{\text{right}} = \frac{1}{\text{distance}_{\text{min}} \times \text{distance}_{\text{right}}} = 0.1029$$

Implementing ITD

To implement the ITD, we perform the following steps:

1. Read in short array of sound
2. If the sound source is coming from the left:
 - a. Left channel: add silence to the end of the sound source
 - b. Right channel: add silence to the front of the sound source

If the sound source is coming from the right:

- a. Left channel: add silence to the front of the sound source
- b. Right channel: add silence to the end of the sound source

```
/** return a SS-modified source */
public short[][] spatializeSource(short[] source) {
.... }

/** = private:: a short array with the itd added before the source */
private short[] addITDBefore(short[] source) {
.... }

/** = private:: a short array with the itd added after the source */
private short[] addITDAfter(short[] source) {
.... }
```


Appendix A - Team Members

Name	Field of Study	Graduating Semester/Year
Arora, Rahul	Computer Science (MEng)	Spring 2012
Bailey, Graeme	Computer Science (Faculty)	n/a
Bard, Justin	Computer Science	Spring 2015
Edmundson, Annie	Computer Science	Spring 2014
Elmachtoub, Najla	Computer Science Engineering Management (MEng)	Fall 2011 Fall 2012
Frankel, Brandon	MBA Candidate	Spring 2013
Hautbois, Flavian	Computer Science (visiting graduate student from <i>Centrale Paris</i>)	n/a
Kooi, Kris	College Scholar Music	Spring 2011
Lin, Natalie	MBA Candidate	Spring 2013
Liu, Christopher	Computer Science	Spring 2015
Naka, Risa	Computer Science (M. Eng.)	Fall 2012
Ng-Quinn, Nora	Computer Science Film	Spring 2013
Maloo, Vinay	Computer Science (MEng)	Spring 2012
Pullano, Jonathan	Computer Science (MEng)	Spring 2012
Qin, Crystal	Computer Science	Spring 2014
Sams, Ashley	Information Science	Spring 2013
Triska, Joseph	Independent Studies in Computer Science and Film	Spring 2014

Appendix B - Design Backlog

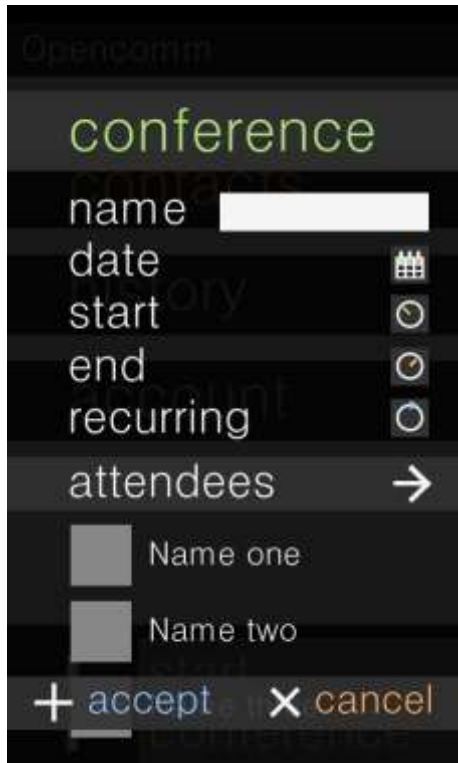
The following are ideas not implemented into the application. These ideas will not necessarily be in the final application, but they can serve as a starting point for brainstorming.

- Designing for different screen sizes
- Changes with moderator rights for each chat
 - Loss of connectivity could be an issue
- User testing
 - New conference screen with sound
 - Test in group setting
 - Setting up an account
 - Creating and going to conferences
- Accounts
 - Allowing users to add multiple email addresses for an account
- Notifications
 - Conferences
 - Contact requests
 - Changes during the conference (users entering/leaving room)
 - Global notifications
- Calendar view of schedule
- Tutorials in app
- Grouping users
 - Email addresses
 - User-defined groups
 - Auto-spatialization
- Contacts screen
- How to add new users into contact list
- Keep a list of bugs
- Audio feedback
- Tactile feedback
- Portrait mode
- Text chat within action bar
- Enhanced error handling
- Tutorial
 - Option not to display tips
- Scalable icon sizes
- Menu screens and more specific menu items
- Where people end up when they get kicked out of a conference

Appendix B – View Management

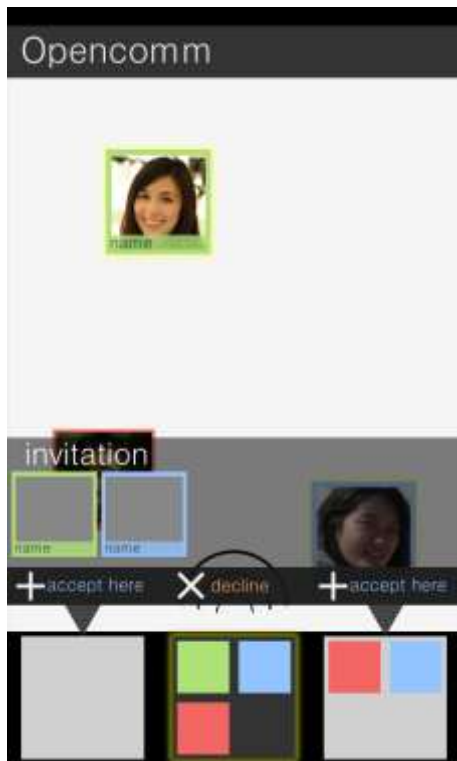
As explained earlier, the MVC model required every screen the user sees to basically have a separate view defined, which talks to its corresponding Controller and models if required. Below is a detail of every view, the challenges and uniqueness it contains, and what they mean.

ConferencePlannerView



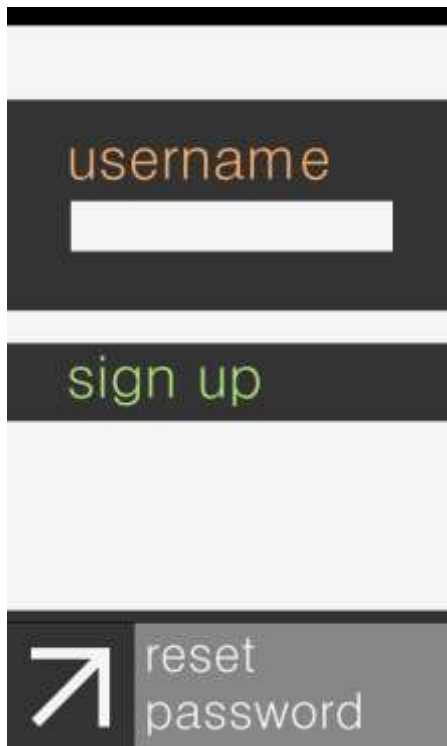
The objective of this screen is to let the users create conferences to be scheduled for future. This screen can be launched by clicking on “Conferences” under dashboard. It contains basic attributes such as name, date, start and end time, recurrence and most importantly the list of invitees much like the “Add users” on the main conference screen. When the create button is clicked, a call is made to scheduling service’s pushconference function. Pushconference function connects with Openfire’s scheduling plugin to schedule the conference and goes back to the dashboard. If the conference has to be created in real-time, pushconference function creates the conference and goes to to the MainApplication screen. Scheduling Service is explained in detail in the backend section.

InvitationPopupPreviewView



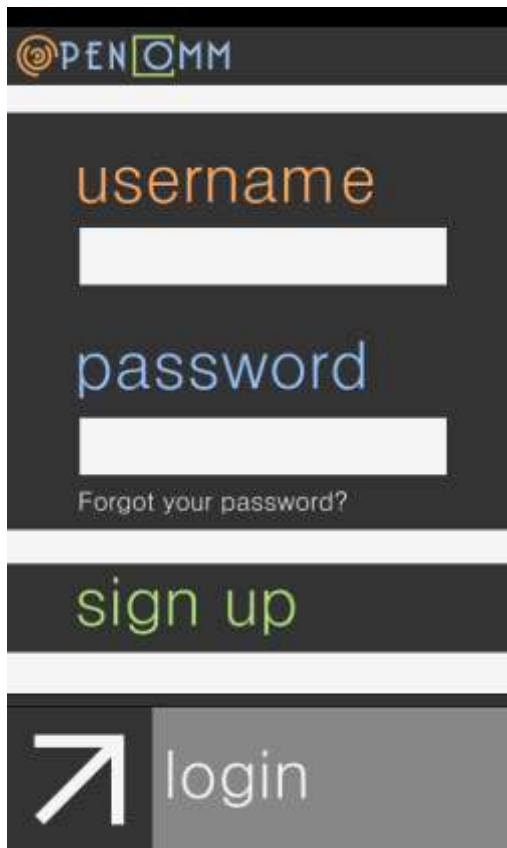
This view is initialized when the user receives an invitation to a side chat. The `NetworkService` class provides the listener that launches this view in the event of an invitation, and we check for the case where the inviter is a member of the current main chat, which signifies a side chat invitation. The user is allowed to click on the accept button on either of the two side chats, which upon clicking will accept the invitation and place the user there. With regard to network, this initializes a new `Space` upon the user's invitation acceptance, and takes the appropriate measure of dismissing the old room. This will have the effect of putting the user into a space corresponding to the invitation's room.

ResetPasswordView

A vertical UI mockup for the ResetPasswordView. It features a dark grey header bar at the top. Below it is a light grey bar. The main content area has a dark grey background. It contains a text input field with the label 'username' in orange text above it. Below the input field is a light grey bar. At the bottom is a dark grey bar with a white arrow icon pointing up and to the right, followed by the text 'reset password' in white.

This view depends on the network to check if an email is valid. There is a Regex check prior to, but upon the user pressing "Reset Password", we query the server regarding the username entered, and the UI provides the suitable notifications for success or failure of the user's request. Thus, if the request is successful, the user will be notified that a new password is sent to the registered email, while if the username does not exist, then the user request will not pass through.

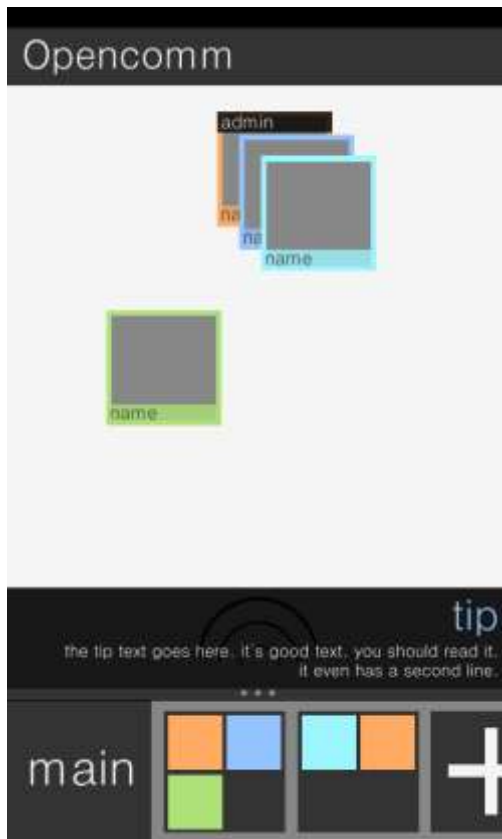
LoginView



The image shows a mobile application login screen. At the top is a dark header with the 'OPENCOMM' logo in white and green. Below the header is a dark gray area containing the text 'username' in orange, followed by a white input field. Underneath is the text 'password' in blue, followed by another white input field. Below the password field is a link that says 'Forgot your password?'. Below this is a dark gray button with the text 'sign up' in green. At the bottom is a dark gray button with a white right-pointing arrow icon and the text 'login' in white.

This view depends on the network to check if a username and password pairing is valid. Upon the user pressing "login", we query the server regarding the username entered, and the UI provides the suitable notifications for success or failure of the user's request. Thus, if the request is successful, the user enters the application dashboard.

PopupNotificationView



```
public PopupNotificationView(Context context, String[] args, String head, String
line1, String line2, int type) {
.....
}
```

This is a standardized view for any kind of error notifications and tip to be popped up to the user. As seen from the constructor, it is simple to use as all that needs to be passed in are the appropriate Context, header text, first line, second line and the type of notification and one can use this view anywhere they need. There is also a **NotificationView** in the framework, which is used to display quick error messages on one line popup. For example, invalid email in the signup screen.

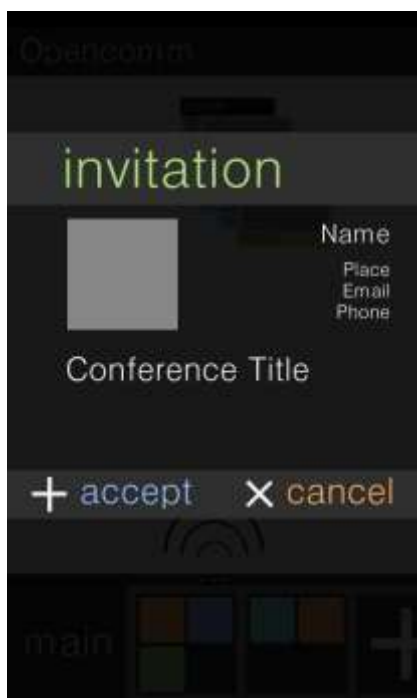
SettingsView / Accounts Edit Screen



A screenshot of a mobile application's 'my account' settings screen. The screen has a dark background. At the top, the text 'my account' is displayed in a light green font. Below this, there are several input fields: 'first name', 'last name', 'email', and 'photo'. The 'photo' field contains a white right-pointing arrow icon. Below the 'photo' field is a 'title' field. At the bottom of the screen, there are two buttons: a blue '+ save' button and an orange 'X cancel' button.

This view is very similar to the account creation/signup page, except the email cannot be changed. Also note that as the screen is launched, all the details of the account are auto populated with a corresponding call to the backend.

Invitation/Confirmation View



A screenshot of a mobile application's 'invitation' confirmation screen. The screen has a dark background. At the top, the text 'invitation' is displayed in a light green font. Below this, there is a grey square placeholder for a profile picture. To the right of the placeholder, the text 'Name', 'Place', 'Email', and 'Phone' is listed vertically. Below the placeholder, the text 'Conference Title' is displayed. At the bottom of the screen, there are two buttons: a blue '+ accept' button and an orange 'X cancel' button. At the very bottom, there is a 'main' label and a small icon of a mobile phone.

Signup View



my account

first name

last name

email

photo

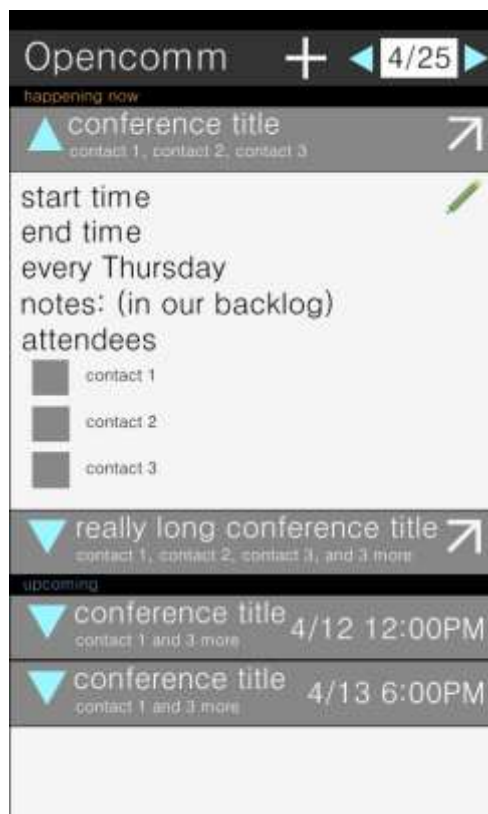
→

title

+ create x cancel

The signup view is implemented as a `LinearLayout` inside a scroll view, with many text fields inside of it. When the user submits the form, a script is called which create a new account on the server.

Conference List View



Appendix D - The XMPP Protocol

1.1 Introduction

This section requires basic knowledge of the XMPP protocol and the three packet types: message, iq and error. In addition, knowledge of the Smack library will be helpful since most of the signaling through XMPP Protocol is implemented using the smack library.

Smack Documentation:

<http://www.igniterealtime.org/builds/smack/docs/latest/documentation/>

1.2 Resource

The XMPP Protocol is one of the two backbones of the OpenComm project: XMPP for signal channel and RTP for payload/media (audio, at present) channel. Server connections, Reconnection management, Buddy-list, Multi-user chat room (MUC) management, session negotiation are all done through the XMPP protocol. We rely on the asmack library to handle the grunt XMPP work for us.

1.3 Implementation

1.3.1 XMPP connection management

All OpenComm clients are connected to Openfire Server and this requires a XMPP connection to manage connections, logins and creating conferences on the server.

Asmack has connection management to handle these tasks. To connect to a server, all we need is the server's DNS or IP address. Below is a straightforward example:

```
XMPPConnection xmppConn = new XMPPConnection("cuopencomm.no-ip.org");
xmppConn.connect();
```

This example shows how to connect to a server with IP = cuopencomm.no-ip.org. XMPPConnection can take more parameter, such as port. When these parameters are unspecified, it will take default values. In this case, port = 5552.

Logging in to the server using simple authentication mechanism is straightforward:

```
xmppConn.login(userName, password);
```

1.3.2 Reconnection Management

ReconnectionManager is enabled by default, and as far as I can see is not meant to be used explicitly (the constructor is private).

If you'd like to verify that it's actually working, or e.g. do something upon reconnection, you might add a custom ConnectionListener to your XMPPConnection like this:

```
connection.addConnectionListener(new ConnectionListener() {
    @Override
    public void reconnectionSuccessful() {
        logger.info("Successfully reconnected to the XMPPserver.");
    }
    @Override
```

```

apublic void reconnectionFailed(Exception arg0) {
    logger.info("Failed to reconnect to the XMPP server.");
}
@Override
public void reconnectingIn(int seconds) {
    logger.info("Reconnecting in " + seconds + " seconds.");
}
@Override
public void connectionClosedOnError(Exception arg0) {
    logger.error("Connection to XMPP server was lost.");
}
@Override
public void connectionClosed() {
    logger.info("XMPP connection was closed.");
}
});

```

Every time a connection is dropped without the application explicitly closing it, the manager automatically tries to reconnect to the server.

The reconnection mechanism will try to reconnect periodically:

1. For the first minute it will attempt to connect once every ten seconds.
2. For the next five minutes it will attempt to connect once a minute.
3. If that fails it will indefinitely try to connect once every five minutes.

1.3.3 Buddy-list

The connection management also provides method to get a specific user's roster (a.k.a the buddy list).

```

Roster xmppRoster = xmppConn.getRoster();
Collection<RosterEntry> entryCollection = xmppRoster.getEntries();
for (RosterEntry r:entryCollection){
    System.out.println(r.getName() + "(" + r.getUser() +") is: " );
    Presence p = roster.getPresence(r.getUser());
    System.out.println("Presence: " + p);
    System.out.println("Presence type: " + p.getType());
    System.out.println("Presence mode: " + p.getMode());
}

```

1.3.4 Multi-User-Chat (MUC) Room

The Smack/Asmack library allows you to create MUC room on a server as well. To create a MUC room, you must first login to a server and then simply create one by:

```

String roomname = "test_room@conference.localhost.localdomain";
MultiUserChat muc = new MultiUserChat(connection, roomname);

```

The connection parameter is just the XMPPConnection object that we have shown earlier.

To invite somebody to the chat room, simply do muc.invite(JID).

To join a chat room, just do muc.join(JID).

While creating the room and sending out invites are pretty straightforward, listening for packet that is relevant to the ongoing chat room (such as chat message, invitation) is a little bit complicated. The way to do it is by adding various packetlisteners to the current connection.

Asmack has already got these packetlisteners implemented. To add a listener to receive room invitation, just do:

```
MultiUserChat.addInvitationListener(connection, new InvitationListener() {  
    public void invitationReceived(XMPPConnection conn, String room,  
                                   String inviter, String reason, String password,  
                                   Message message) {  
        //Whatever needs to be done goes here  
    }  
})
```

Basically, while all the XMPP packets are transmitted in pure XML format, the Smack library has some methods to intercept specific packets according to the Namespace attribute. These methods are called providers. A provider will further parse the XML and extract the information in this packet and return this information. For the example above, there is a provider that intercepts the invitation packet, parses through it and finds out the attributes *connection*, *room name*, *inviter's JID*, *reason*, *room password* and *messages* if there are any.

Now you can imagine that same thing will apply to messages exchanged during a chat session, i.e. we add a packet listener for all the chat messages. Inside this listener, we can just output the message that has been received.

Another important subject of a MUC room is the affiliation. There can be owners, administrators, members, and participants in a chat room. Owners will have the privilege to change any JID's affiliation in his chat room. The owner can even set somebody else as another owner of the chat room.

Appendix E - Audio, Jingle Protocol, and RTP from Spring 2010 M.Eng. Report

6 Audio on Android

One major issue with the Android emulator (as of May 2010) is that the emulated microphone cannot be instantiated; the emulator will crash if this is attempted. This means that most audio work from within the emulator will be done with pre-recorded files. On the development phone, the microphone can be instantiated as you would expect. We support both means of audio streaming in our Test XMPP Client with a few key lines that need to be commented out or in to switch between microphone use and file use.

On a high-level, audio playback and recording is accomplished by the MediaPlayer and MediaRecorder classes, Google documentation at (<http://developer.android.com/guide/topics/media/index.html>). However, there's an overhead with these classes that is unacceptable for real-time streaming.

For low-level audio playback and recording, we use the AudioTrack and AudioRecord classes, which allow direct write of the Linear PCM data going to the speakers, and direct read from the Linear PCM data coming from the microphone. Our code that uses these classes comes from the Sipdroid codebase (sipdroid.org), which is licensed under the GPL v3.

6.1 AudioTrack

AudioTrack is instantiated as:

```
AudioTrack track = new AudioTrack(AudioManager.STREAM_MUSIC, 8000,
AudioFormat.CHANNEL_CONFIGURATION_MONO, AudioFormat.ENCODING_PCM_16BIT,
    BUFFER_SIZE*2*2, AudioTrack.MODE_STREAM);
```

Where BUFFER_SIZE is defined as 1024 bytes.

This sets up an AudioTrack using the Music stream for playback, and allows us to write audio to the device as it plays. We use 8000 kHz 16-bit mono for the audio parameters. After starting the AudioTrack, writing a sequence of shorts (16-bit samples) to the AudioTrack will cause them to be played back according to the audio parameters.

6.2 AudioRecord

AudioRecord is instantiated as:

```
AudioRecord record = new AudioRecord(MediaRecorder.AudioSource.MIC, 8000,
AudioFormat.CHANNEL_CONFIGURATION_MONO, AudioFormat.ENCODING_PCM_16BIT,
AudioRecord.getMinBufferSize(8000, AudioFormat.CHANNEL_CONFIGURATION_MONO,
AudioFormat.ENCODING_PCM_16BIT));
```

This sets up an AudioRecord using the microphone recording in 8000kHz 16-bit mono. After starting the AudioRecord, we read into a section of a buffer, perform some amplification and noise calculation based on previous portions of the buffer, and package that section into a sequence of shorts (16-bit samples).

7.3.3 Jingle Negotiation

This section discusses the construction and parsing of Jingle packets. The asmack library handles the sending and receiving of Jingle packets, but as of now, mechanisms to parse jingle packets do not exist in asmack.

To send a packet, just simply call `connection.send(packet)`. However, listening to a specific kind of packet is not trivial. Because asmack does not implement the jingle protocol, we had to define our own Jingle packet, `JingleIQPacket`. The asmack library obviously has no existing listeners to intercept our packets.

The way to handle any new packet is:

Write a Provider to intercept the packet, according to its namespace. The provider then parses the XML, extracts the information and constructs a `JingleIQPacket` with the information. A packet listener must be added to the connection, similar to what we did for the invitationlistener. We can also add `PacketFilter` to the `PacketListener` so that it only “listens” for certain types of packet.

Remember, it is always the Provider that intercepts the packet first, and parses the XML to construct the corresponding packet. Then the `PacketListener` will receive the packet, and according to the `PacketFilter`, the Listener will decide whether to process the packet or ignore it.

Thus for our Jingle negotiation session, we are only interested in the `JingleIQPacket` and the follow up acknowledgements in the form of regular IQ packets. So we created a `PacketListener` with `PacketFilter(packet.TYPE.IQ)`. This way, only the IQ packet (`JingleIQPacket` is an extension of IQ) will go into the listener and be processed.

8 The Jingle Protocol in OpenComm

8.1 Pre-requisite

Before diving into this section and working on the `com.cornell.opencomm.jingleimpl` package of the OpenComm, please make sure you have a decent understanding of the XMPP protocol and the Jingle protocol. Good reference materials are:

The XMPP Protocol general: O'REILLY's XMPP: The Definitive Guide

The Jingle Protocol intro: O'REILLY's XMPP: The Definitive Guide, Chapter 9

Jingle Negotiation Flow: XEP-0166 <http://xmpp.org/extensions/xep-0166.html>

Jingle RTP: XEP-0167 <http://xmpp.org/extensions/xep-0167.html>

Jingle raw-udp: XEP-0177 <http://xmpp.org/extensions/xep-0177.html>

Jingle ice-dup: XEP-0176 <http://xmpp.org/extensions/xep-0176.html>

What follows is a brief discussion on the implementation of the Jingle protocol in OpenComm.

8.2 Vision

Jingle is an extension of the XMPP protocol and as such is a protocol for multimedia session (audio/video) negotiation. Since the OpenComm project uses XMPP as its signaling channel and RTP as its audio channel, we must have Jingle for our audio chat negotiation.

The purpose of adopting a standard protocol (Jingle) for our session negotiation is to allow compatibility with other chat clients that support Jingle for their signaling and negotiation.

8.3 Resource and Challenges

There are two existing library for the Jingle Protocol. One is called `libjingle`, which is developed by Google and used in GTalk. The library however employs certain extensions that are not fully

incorporated into XMPP Jingle yet and as such does not comply completely with the XMPP Jingle protocol. This library is written in C++ and thus we were not able to use the library for this project. The other is smack-jingle, which is part of the Smack library for the full XMPP protocol. This library was written in Java and Smack is also able on Android platform as asmack. However, smack-jingle is not yet available on the Android platform. If one were to hope for an Android platform jingle library, the best bet is asmack-jingle. But for now, we were left with no choice other than to develop our own Jingle library.

The challenge of implementing the full Jingle protocol is that the protocol itself is well, rather complex! Without first defining a proper high level architecture of the implementation, the implementation gets very messy and very hard to maintain. We learnt this lesson the hard way. We started off implementing a couple of negotiation scenarios, but as we encountered more and more scenarios we figured the problem is just out of the scope to handle. Thus we decided to restart from scratch to do define a proper high level framework. As for now, we have not implemented the full Jingle protocol. But after all the hard work, the high level architecture of the protocol implementation is fully defined along with a decent number of Jingle packet types. Future developers should not have a hard time to continue developing the protocol on top of the current architecture.

8.4 Jingle Implementation

There are three major things we have implemented for the Jingle protocol:

1. Constructing a Jingle packet according to the XMPP standard Jingle packet schema
2. Parsing a Jingle packet according to the XMPP standard Jingle packet schema
3. Determining events based on incoming IQ/JingleIQPackets and doing appropriate transitions in the jingle negotiation state.

8.4.1 Constructing a Jingle Packet

To construct a specific jingle packet, first construct a JingleIQPacket by calling constructor: JingleIQPacket() or JingleIQPacket(String from, String to, String action). The first one is a dummy constructor while the second one is more formal and can be used when you know the source and destination at the time of packet creation.

Then we can construct the contentElementType, define the necessary attribute and add it to the JingleIQPacket by calling addContentType(ContentElementType content).

In general, the JingleIQPacket is constructed in a nested fashion. An outer element (such as *jingle*) will contain the inner element (*content* or *reason*).

Let's work through an example real quick to demonstrate the creation a JingleIQPacket.


```

<iq from="tester2@ming8832/yate" type="set" to="tester1@ming8832/minglaptop" id="JG4_1563246179_1">
<jingle xmlns="urn:xmpp:jingle:1" initiator="tester2@ming8832/yate" responder="tester1@ming8832/minglaptop"
sid="JG4_1563246179" type="session-initiate">
<content creator="initiator" name="jingle/4_content_1894869935" senders="both">
<description xmlns="urn:xmpp:jingle:apps:rtp:1" media="audio">
<payload-type id="0" name="PCMU" clockrate="8000"/>
<payload-type id="3" name="GSM" clockrate="8000"/>
<payload-type id="8" name="PCMA" clockrate="8000"/>
<payload-type id="11" name="L16" clockrate="8000"/>
<payload-type id="98" name="iLBC" clockrate="8000"/>
<payload-type id="98" name="iLBC" clockrate="8000"/>
<payload-type id="106" name="telephone-event" clockrate="8000"/>
</description>
<transport xmlns="urn:xmpp:jingle:transports:raw-udp:1">
<candidate generation="0" port="21648" component="1" ip="192.168.1.6" id="jingle/4_candidate_95770219"/>
</transport>
</content>
<content creator="initiator" name="jingle/4_content_1383165658" senders="both">
<description xmlns="urn:xmpp:jingle:apps:rtp:1" media="audio">
<payload-type id="0" name="PCMU" clockrate="8000"/>
<payload-type id="3" name="GSM" clockrate="8000"/>
<payload-type id="8" name="PCMA" clockrate="8000"/>
<payload-type id="11" name="L16" clockrate="8000"/>
<payload-type id="98" name="iLBC" clockrate="8000"/>
<payload-type id="98" name="iLBC" clockrate="8000"/>
<payload-type id="106" name="telephone-event" clockrate="8000"/>
</description>
<transport xmlns="urn:xmpp:jingle:transports:ice-udp:1" pwd="1748506183172527191316" ufrag="2042"/>
</content>
</jingle>
</iq>

```

Figure 2: An example Jingle packet in raw xml

1. The procedure to create this Jingle IQ Packet is:
2. Construct a JingleIQPacket by calling the constructor, set the attribute values that are highlighted in light green. Since JingleIQPacket is an extension of IQ, constructing a JingleIQPacket will also take care of the construction of the IQ header (the part that is highlighted in dark green)
3. Construct a ContentElementType, define the attributes (the first pink highlight) by calling the setter functions
4. Construct a DescriptionElementType, define the attributes (first yellow highlight)
5. Construct a bunch of PayloadElementType, define the attributes of each one, add them to the DescriptionElementType created in step 3:
6. DescriptionElementType.addPayload(PayloadElementType payload)
7. Set the descriptionElementType to the ContentElementType created in step 2:
8. ContentElementType.
9. setElementDescription(DescriptionElementType elementDescription)
10. Construct a TransportElementType, define the attributes (first brown high lighter)
11. Construct a CandidateElementType, define the attributes (red high lighter), add it to the TransportElementType created in step 6:
12. TransportElementType.addCandidate(CandidateElementType candidate)

13. Set TransportElementType to ContentElementType created in Step 2:
14. ContentElementType.
15. setElementTransport(TransportElementType elementTransport)
16. Now the first ContentElementType is properly defined, add it to the JingleIQPacket:
JingleIQPacket.addContentType(ContentElementType content)
17. Create another ContentElementType, define the attribute and nested-element by repeating step 3-9. At the end, add it to JingleIQPacket.
18. Now the JingleIQPacket should be properly constructed. To check the validity of the XML, use JingleIQPacket.getChildElementXML() method.
19. To send the packet, use XMPPConnection connection.send(JingleIQPacket)

At this point, you should have a good understanding for the procedure of creating the packet. Some points worth noting:

1. Some of the attributes which have well defined values (like jingle action) have interfaces implemented for them that contain all possible values. Use those to avoid simple errors like sending out a “session-initiat” instead of a “session-initiate”. As an example, all possible values for the attributeAction are defined in the interface:
JingleIQPacket.AttributeActionValues
2. There are many different Jingle packet schemas, according to the Action attribute. As an example, the jingle packet with action set to “session-terminate” does not contain the ContentElementType but contains a reasonElementType. If one would like to construct a termination packet, the procedure is:
3. Construct a JingleIQPacket with actionAttribute ==
4. JingleIQPacket.AttributeActionValues.SESSION_TERMINATE
5. Construct an object of ReasonElementType
6. Add the ReasonElementType object to the JingleIQPacket

8.4.2 Parsing a Jingle Packet

The first step towards parsing any packet in the Smack framework is to register a sub-class of an org.jivesoftware.smack.provider.IQProvider. In our implementation, the class JingleIQProvider extends IQProvider and the registration happens in the method JingleIQBuddyPacketRouter.setup() as demonstrated by the following snippet:

```
JingleIQProvider jiqProvider = new JingleIQProvider();
ProviderManager.getInstance().addIQProvider(JingleIQPacket.ELEMENT_NAME_JINGLE,
JingleIQPacket.NAMESPACE, jiqProvider);
```

The above piece of code merely tells the ProviderManager that all IQ packets which have their child elements falling in the namespace JingleIQPacket.NAMESPACE and are named JingleIQPacket.ELEMENT_NAME_JINGLE are to be parsed by an instance of JingleIQProvider, namely jiqProvider.

To understand the functioning of the JingleIQProvider.parseIQ method, the best resource is the code itself along with the various XMPP Extension Protocols outlined in section 7.1. Although it is hard to clearly demarcate which specific XMPP Extension Protocol specification was used to come up with the rules for parsing a particular element type, the following list serves as a good starting point:

- XEP-0166: *jingle, content, reason*
- XEP-0167: *description, payload*
- XEP-0177: *transport, candidate*

In this section, we will not explain the parsing rules, which can be understood by reading the above specifications. We will instead try to explain the structure of the JingleIQProvider and its related classes. Each element in the XML is represented by a different *ElementType class defined in the com.cornell.opencomm.jingleimpl package. In the provider, there are separate methods to parse each of these elements. A method while parsing its designated element will encounter the start-tag of a child element and will in turn call the corresponding method to handle that particular child XML element. As an example, when in parseJingleElement, a <content> tag is encountered; a call to parseContentElement is made which returns an object of ContentElementType. Similar logic applies recursively to handling each element defined in the XML.

8.4.3 Handling a new ElementType in our Jingle Implementation

In the future, when there would be plans to implement SRTP (Secure RTP) or ICE-UDP, new ElementTypes will need to be handled and parsed accordingly. To do this, two things need to be done.

1. Declare a new *ElementType which extends the BaseElementType in com.cornell.opencomm.jingleimpl and declare variables for each attribute of the XML element that this new class will represent. Also declare variables that are references to objects that represent child-elements of this XML element.
2. In the JingleIQProvider, add appropriate methods to parse the new ElementType and its sub-types that return a constructed *ElementType representing the new ElementType.

8.4.4 Session Management

The Session Management is again based on the XMPP Extension Protocols mentioned in earlier sections. The key to understanding a state change is to understand what type of IQ packet was received in what state. The code that deals with the session management in our implementation is largely confined to two areas – the class SessionCallStateMachine and the method MUCBuddy.processPacket. The former is an implementation of the state machine presented in section 5.1 Session Management of the XEP-0166 specification and checks what state transitions are valid. The code in MUCBuddy.processPacket is responsible for taking suitable action based on the type and content of an incoming IQ or JingleIQPacket.

9 RTP on Android

RTP (Real-time Transport Protocol) is a media delivery protocol built on top of UDP and lies at the core of most VoIP systems. However, there is no standard library for RTP (Real-time Transport Protocol) in Android. The RTP implementation we use comes from the Sipsdroid codebase (sipsdroid.org), which is licensed under the GPL v3.

9.1 RTP Sending

Sending RTP packets simply involves setting various fields of the RTP packet, setting the payload to the encoded frame of audio, and sending over UDP to the destination. Receiving RTP packets is similar: the packet is received over UDP, fields are checked, and the payload is decoded. In a 2-way chat, each client has its own sender and receiver thread.

9.2 Multi-User RTP

In standard RTP implementations, multi-user chat is handled by putting RTP on top of multicast UDP. However, Android doesn't support multicast addressing. To get around this, we split off

the audio recording from the RTP stream. We have a MicrophonePusher (or AudioPusher) class that grabs input once, then puts it on a queue belonging to each active RTP stream. A single RTP streaming thread only goes to one recipient, so a streaming thread for each recipient is necessary. Likewise, on the receiver end, an RTP receiving thread is needed for each sender. We do not separate playback from the RTP packet reception, though, since the audio mixer on Android is able to understand simultaneous writes to multiple AudioTracks.

9.3 Codec Used

G.711 is a very basic codec whose strengths lie in the speed of computation and simplicity of implementation. We use G.711 A-law encoding for the audio we send over RTP. The G.711 codec implementation we use also comes from Sipdroid.

9.4 NAT Traversal

When a client announces its IP address as a candidate for an RTP connection, it has obtained its public-facing IP address by using STUN for NAT traversal. We used Sipdroid's implementation of STUN, which was itself based off of jStun.