

CS5023 – Practical 1

Mobile chat application

Architecture of the application

Quick overview

The application starts by displaying a contact list. The contacts are saved in a database, so that the user won't have to add the same contacts every time the application restarts. When the application starts, it tries to read any saved contacts from the database and display them in a Listview. It also displays a button which creates an AlertDialog in order to add a new contact.

When the user clicks on any of the contacts, a new SimpleMessenger activity starts and it displays a chat window with that contact. Before it starts, by using `Intent.putExtra()` I pass the selected contact's IP address, port number and username to the newly started Activity. When it's created it also tries to read any saved messages history that is saved to the database and display it in the chat window in a greyed out colour. This class has a handler which lets it communicate with the server so that it gets the received messages that are directed to that particular contact and display them in the UI.

The server is quite a simple class which extends Thread and constantly listens for new messages. In the beginning that's all I was using and it was working fine, but later I decided to create a Service, in order to be able to display notifications about missed received messages. When the service is created, it starts that server thread. When the server receives a message, it checks whom it is from. If the received packet is from the same contact with whom a chat window is currently open, it simply uses a handler to communicate with that chat window and display the message in the UI. If the message is from a different contact, or if the user is currently on another screen, like looking at the contacts list or doing something else on their phone, then the server uses another handler in order to communicate with the ChatService and create a status bar notification about the incoming message. When the notification is clicked, then it is dismissed and a chat window with that contact opens up, displaying any missed messages. If that message is from a contact that didn't exist in the contact list, then it is automatically added to the database.

Class descriptions:

1. **ContactsActivity:**

This is the main Activity. Its purpose is to display a contact list, the add a new contact button, and provide the ability to start a new chat window with any of the contacts by clicking on them.

When this activity is created a few things get initialised: The ChatService is created and started, the Database helper is initialised, so are the various views, and the contact's list view is created and populated by retrieving saved contacts from the database. The contacts Listview also has a context menu which appears on long-press and gives the option to delete a contact, and there's a normal menu with the options to quit (which just brings the app to the background) and to bring up a dialog which informs the user of their own IP address and port, which might be useful when adding new contacts.

2. UDPServerThread

As the name suggests, this class is the server and extends Thread. It gets started at the beginning and then keeps running. In its run() method it constantly keeps listening for received packets.

It has two handler references: one for delivering messages to a chat window (that is a SimpleMessenger activity), and one for delivering messages to the ChatService.

Every time a chat window is created, the server's handler is set to that particular chat window's handler, and when that chat window is destroyed, it is set back to null. We also save the current window's IP address in a field in the server class, so that we can compare it to the addresses of other incoming packets. This handler delivers the received chat messages from the server to the chat window activity so that they can be updated in the UI.

The other handler delivers the messages to the chat service, so that a notification is created and shown.

In order to decide which of the handlers to use, the packet's IP is checked. If it's the same as the currently open chat window (if there's one), then the first handler is used to display the received chat message in the chat window, otherwise a notification is created.

If the packet's IP doesn't exist in the database, then it is added to it, and a new contact is added to the contact list.

3. ChatService

This class extends Service. When it's created it starts the server thread, initialises the notification manager, and sets the server's serviceHandler. We don't use any binding because this is just a local service and it's not intended for other applications to use, so the class is kept pretty simple. We also create a handler which receives messages from the server. The received messages are MessageToService objects and have information about the received chat message, the sender's name, IP address and port. When these are received, a notification is created displaying the contents of the message, and providing a PendingIntent so that a SimpleMessenger activity with the given username, IP address and port information is created by clicking the notification. Also the default sound and led light pattern are set for the notification, and the auto cancelling behaviour after clicking on it.

4. MessageToService

This is a simple helper class used to encapsulate the information of the messages sent between the server and the service, and holds information about the received chat message's content, sender IP, port and name.

5. SimpleMessenger

This activity displays a chat window with a certain contact. Before it is started, we pass the contact's IP address, name and port to the Intent using `Intent.putExtra()`. It uses a `ListView` and an `ArrayAdapter<Spanned>` for this `ListView` in order to display the conversation's messages, stacking from the bottom of the list. It also creates a menu with the options to go back to the contact list, and to quit the application. The reason we use `Spanned` instead of `String` for the chat messages is to be able to have formatting, like having the contact's name in bold, and the rest of the message in normal text, to make the conversation easier to read.

Before the activity is destroyed, `onPause()` is called, where we save the conversation history to the database, and `onResume()` we reload it and display it in the `ListView` in grey colour. The conversation history is saved as a single `String` using the new line as a delimiter between messages.

6. ContactDataSqlHelper

This class extends `SQLiteOpenHelper` and is used to create our database. We define four columns: name, IP address, port and history for every contact, and the `BaseColumns._ID` field as a primary auto incremented key. The methods for creating the database table and upgrading (dropping and recreating it) are specified as well. The IP address is set as a unique field, so that we can't have two contacts with the same address.

7. Contact

This is a simple class that holds the information about a contact. That is their name, IP address, port, chat history, information on whether a chat window has been opened (we use that to update the icon to a chat bubble to differentiate between contacts with whom we have on-going conversations from the rest of them) and online status (online / offline).

8. ContactsAdapter

This class extends `ArrayAdapter` and I used it to create a custom adapter that holds `Contact` objects. This is used by the main `ContactsActivity` in order to display the contact list. This class specifies how each element of the contact list `ListView` is going to be displayed. We use the `contact.xml` file to specify the layout of every contact in that list and in the `getView()` method we initialise the views for the contact name, contact picture, online status icon, IP address.

To decide whether a contact is online or not we use `Contact.isAlive()` method, which does a `InetAddress.isReachable()` check with a small timeout and accordingly displays either a green or a grey dot.

Some problems encountered

One of the problems I encountered was that because the Android Emulator runs behind a virtual router that isolates it from the machine's network, it only uses the 10.0.2/24 network address space. So for example when I was trying to find a received packet's IP address in my server code, it was always 10.0.2.2, making it impossible to determine who the sender was. This made testing and debugging more complicated. So, if you run the application on an emulator, there is a problem with receiving messages as the app sees that the packets come from 10.0.2.2 instead of the current contact's IP, so it thinks it is someone else and creates a new conversation window for them. But I tested that on a real device and it was working properly.

I was planning to implement a protocol for checking whether a contact is online, in the server. I was thinking of implementing something like sending a ping, waiting for a pong back, and then sending the actual message, and possibly doing that before sending each message. However because of the difficulty with determining packets IPs and the added complexity of having a finite state machine and time limitations, I didn't implement this. I thought of using `InetAddress.isReachable()` with a small timeout every time a message is sent, which is a much simpler solution instead. This method however, is only working half of the time for some reason, and sometimes says that online hosts are unreachable, so I decided to not give much weight to this result, and use it only to give a small warning when someone tries to speak to an "offline" contact.

Testing and features of interest

I tested quite extensively the application, both between two emulators, and between an android phone and an emulator. I tried different scenarios like receiving messages while the conversation window is not active to see if they are saved successfully and displayed after clicking on the notification. I tried receiving messages from multiple senders to check if each message appears in its correct place and that messages are not getting lost. I also tested scenarios of switching the orientation of the device on various view and checking that the application keeps responding in this case. Also I checked what happens when the phone is offline, for example in airplane mode to make sure that the app wouldn't crash. To avoid erroneous input, I check the input fields so that no empty messages are accepted, for both entering a contact and sending messages. If the user enters an invalid IP, then this is detected messages cannot be sent.

Screenshots

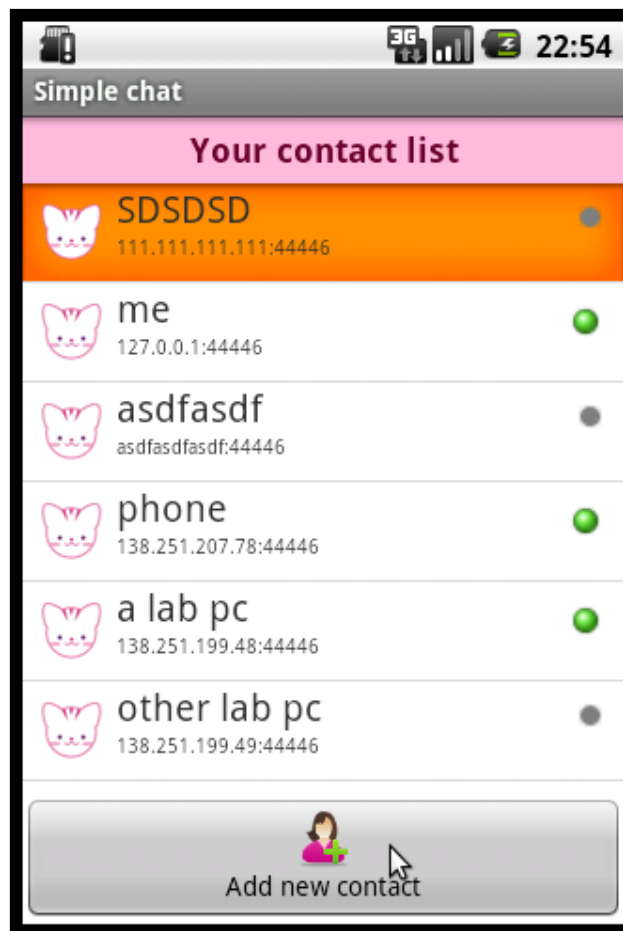


FIGURE 1: INITIAL VIEW OF THE CONTACTS LIST WHEN THE APPLICATION STARTS AND THE ADD BUTTON.

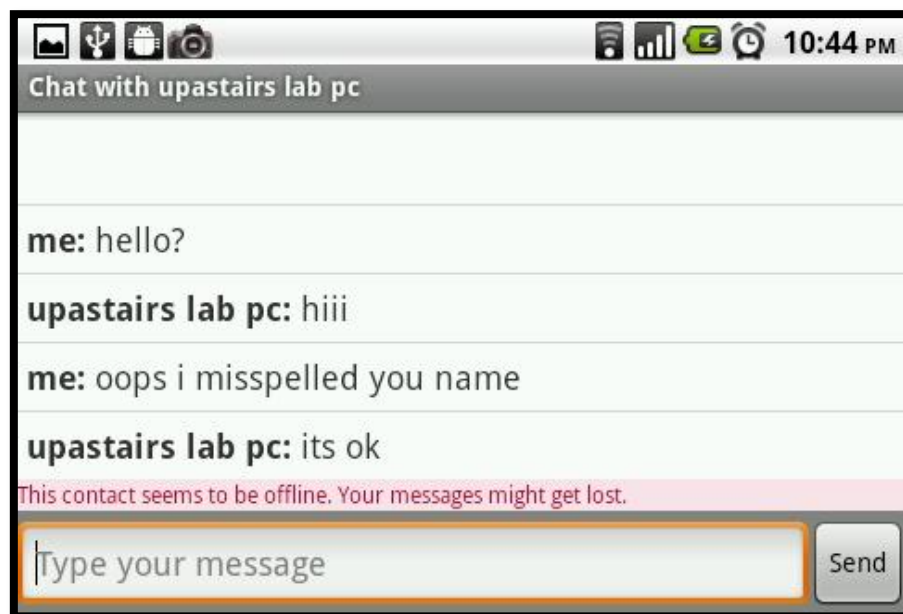


FIGURE 2: A CHAT WINDOW WITH A CONTACT. THERE'S A WARNING THAT THE CONTACT IS UNREACHABLE. HOWEVER WE STILL GIVE THE ABILITY FOR MESSAGES TO BE SENT BECAUSE SOME TIMES THE WARNING IS WRONG.

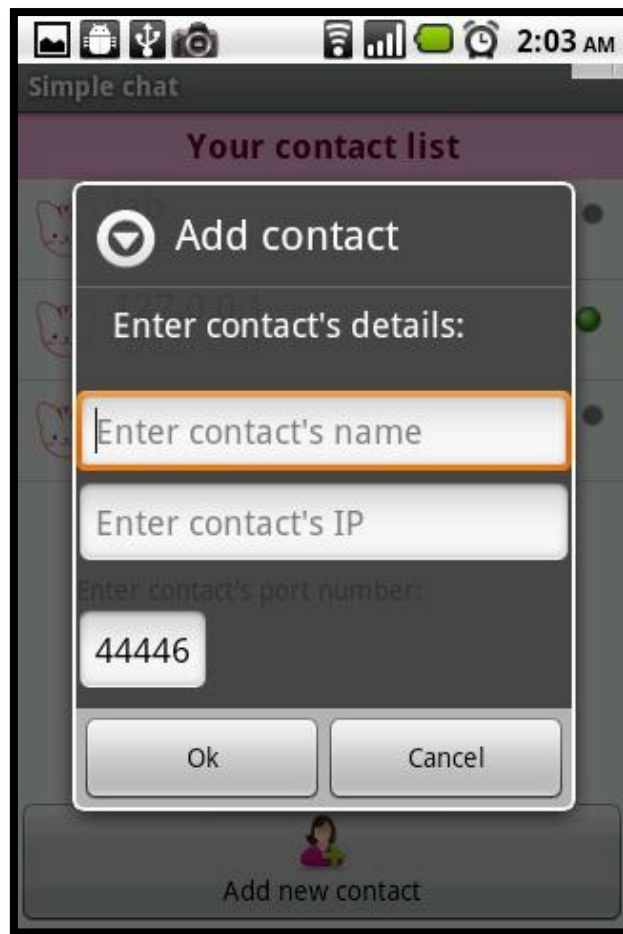


FIGURE 3: THE DIALOG TO ADD A NEW CONTACT AFTER PRESSING THE BUTTON.

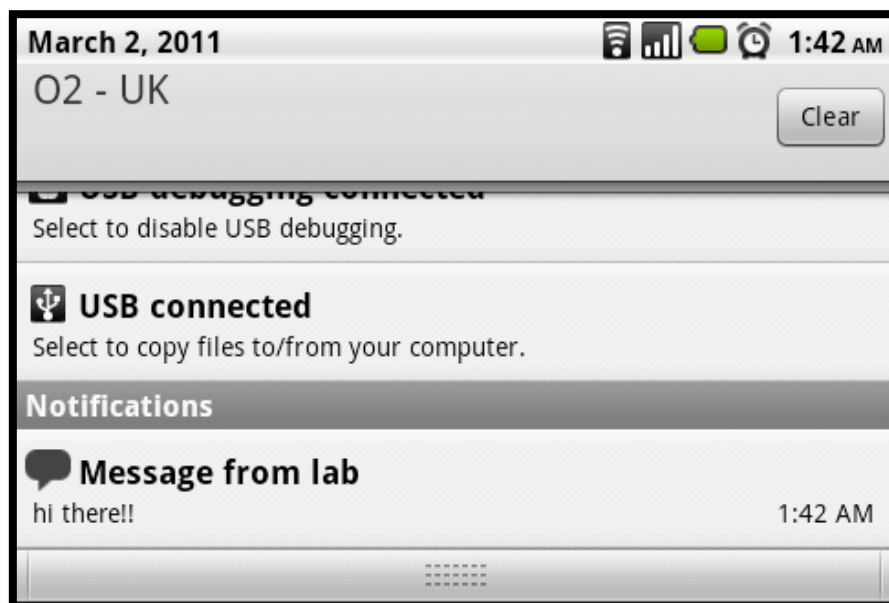


FIGURE 4: A NOTIFICATION INFORMING THE USER OF AN INCOMING CHAT WHILE THEY WERE AT ANOTHER VIEW THAN THE CONVERSATION WINDOW

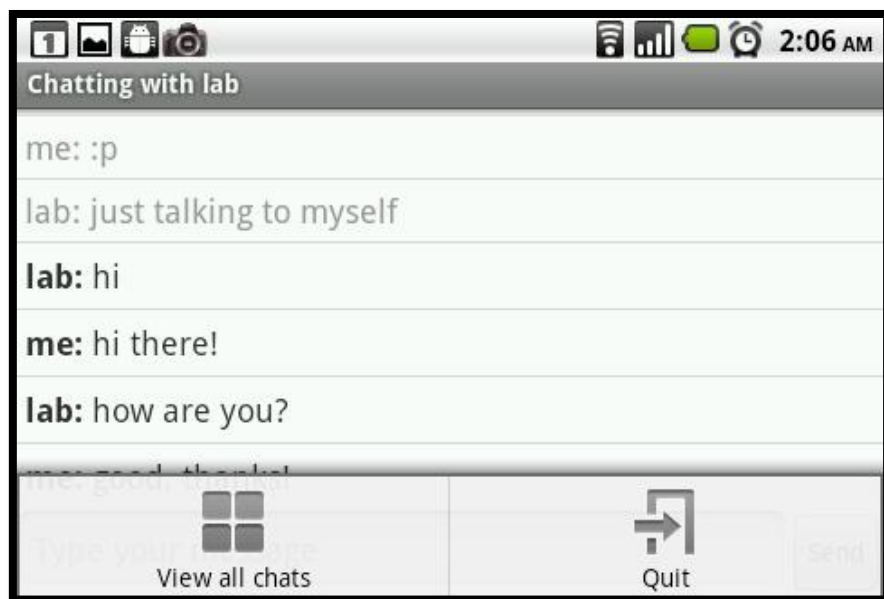


FIGURE 5: THE CHAT WINDOW'S MENU



FIGURE 6: THE CONTACTSACTIVITY MENU



FIGURE 7: THIS WAS NOT A VALID IP SO THERE CAN'T BE A CONVERSATION