**Documentation on VRapid**

**Modules in project**

* Types of Users
* VRapid Server Logging (**LoginActivity**)
* Creating, refreshing, deleting Dialogs and Starting Conference calls (**DialogActivity & SelectUsersActivity**)
* Conference Call (**CallActivity**)
* BaseConversationFragment
* VideoConversationFragment
* Chat (**ChatActivity**)
* **Types of Users**
* Moderator
* Respondent
* Client
* Translator
* Admin
* TRecorder
* **VRapid Server Logging (LoginActivity)**

Each user is first registered in our server from backend.

There is no way a user can register themselves.

The credentials (Mobile Number and Password) which we provide to the user will be used by the user to login in our application.

We send UserID of the user logging in.

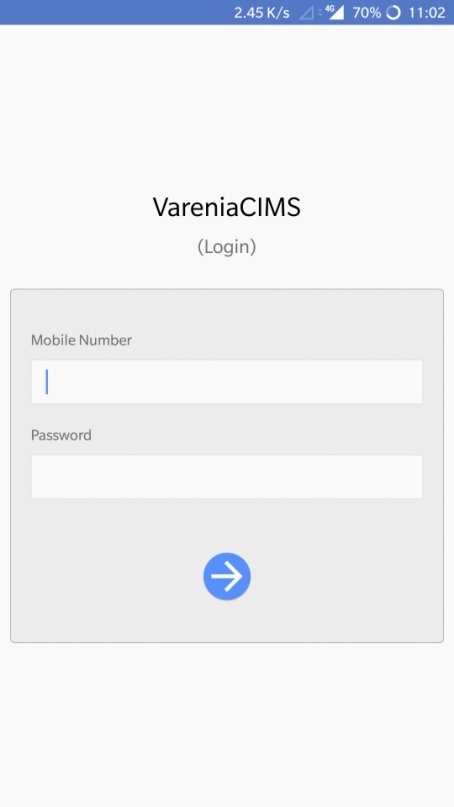
These credentials are stored in the SharedPreferences.

Operations being performed in this activity:

* **(a)** Current user sends a UserId to the VRapid server as a parameter. On return we get details of all the users that corresponds to the same “Room” as the current user does.
* **(b)** Current user also gets the details on which of the users present in the same room they are allowed to See, Talk and Chat with and the Schedule of that group.

(All of these details are stored in the Android SQLiteDatabase)

* **(c)** Once all these details are stored we login the current user to QuickBlox.
* **(d)** After successful login to QuickBlox, we login the user to Chat (this is important for calling or chatting with other users)



* **Creating, refreshing, deleting Dialogs and Starting Conference calls (DialogActivity & SelectUsersActivity)**

What is a dialog?

Suppose there are 10 users present in the “Room”.

Now a dialog can consist of 5 or 8 or 2 users from the group of 10 users.

We can start a conference call using a dialog only. As dialog tells us about the number of participants that there are going to be in a call.

**Steps involed to start a call –**

1. Creating a dialog. This functionality is only provided to Moderator.

* There is a button on the bottom right corner of the screen only visible to moderator.
* Clicking that moderator navigates to **SelectUsersActivity** where he can choose the users out of 10 users (can be any number) to be included in the dialog.
* On creation of the dialog user navigates back to the **DialogActivity**.

1. Refreshing a dialog. This functionality is provided to all the users.

* This will help the user to get the most recent dialog that is created by the moderator.

1. Deleting a dialog. This functionality is provided to all the users.

* Because we only let the user see the most recent dialog doesn’t mean there is only 1 dialog. Suppose moderator creates 2 dialogs and in both of them the current user is present. This will mean corresponding to the current user there are 2 dialogs now.

To unnecessarily have a lot of dialogs is not a good practise. So its better to delete the dialog once the user is completed with the call.

1. Once a dialog has been created, next step is to start the call.

* First step, initialize ConferenceClient.
* Second step, prepare chat service.
* Third step, create a session with ConferenceClient which takes up ArrayList of Integer type containing the QB ids of all the participants.
* On success the user is sent to **CallActivity** where the conference takes place.

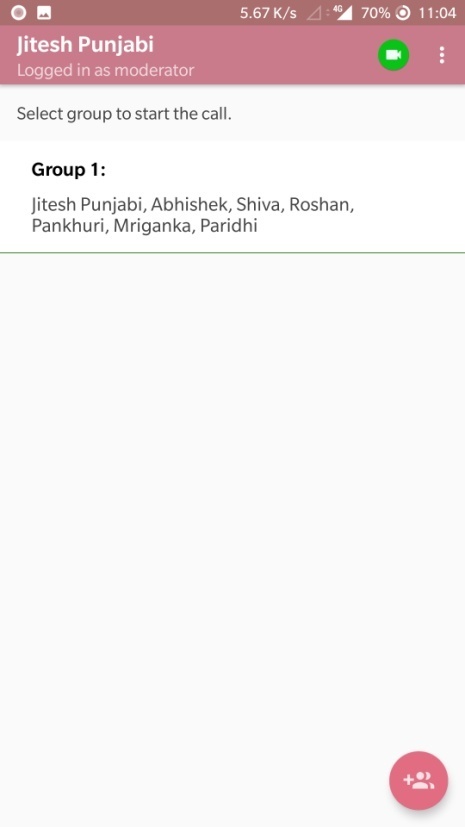
**Schedule**

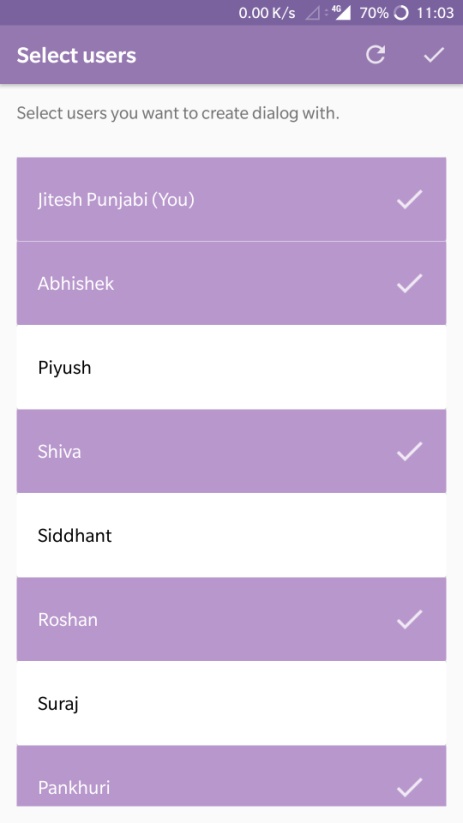
Each group has a schedule for it, i.e. date and time of when it will start.

Call dialogs are only visible before and during the schedule allotted. After that they are not available to the users.

**Add to Google Calendar**

We have the functionality of adding the schedule to google calendar, so user is notified before conference is about to start.





* **Conference Call (CallActivity)**

Activity where the conference takes place.

We perform multiple operations here:

1. Initialize current quickblox session, add callback listeners.
2. Initialize confernce client which gives us a set of overrided methods which handle our camera. Anything which happens to the current user’s camera (opened,closed,freezed,disconnected,error,firstframe) accordingly that method will be called.
3. Set QBRTCMediaConfig parameters such as – video width, video height, video HW Acceleration, videocodec.
4. Initialize audio manager. This is basically setting whether you want your audio to come out from speaker or earpiece.
5. After this, mainly all the work in done on this class. Now we have to navigate to **BaseConverstationFragment** and **VideoConversationFragment**.
6. Before starting the fragment we check if the user is logged in to chat. If yes then start the fragment, if not then first login user to chat and then start the fragment.

**🡪 BaseConversationFragment and VideoConversationFragment**

First step in this class is to register listeners. This class is mainly for handling callbacks.

This is where users will be in call with each other.

Step-wise explanation of what is going on in this activity:

1. If the currentSessions state is “Connected”, we start the

* conversationFragmentCallBackListener.onStartJoinConference();

This invokes a callback on **CallActivity** class. Where the current user joins the dialog.

Eg. of code:

**int** userID = **currentSession**.getCurrentUserID();  
QBConferenceRole conferenceRole = **asListenerRole** ? QBConferenceRole.***LISTENER*** : QBConferenceRole.***PUBLISHER***;  
**currentSession**.joinDialog(**dialogID**, conferenceRole, **new** JoinedCallback(userID));

1. Once the user joins the call, we add Client Connection Callback.

* conversationFragmentCallBackListener.addClientConnectionCallback(this);

This invoked a callback on **CallActivity** class.

Eg. of code:

**if** (**currentSession** != **null**) {  
 **currentSession**.addSessionCallbacksListener(clientConnectionCallbacks);  
}

1. Next thing is to register video track listeners and audio track listeners.

* This enables us to get local (current user) and remote (other users on the conference call) video tracks and audio tracks.

currentSession.addVideoTrackCallbacksListeners(this);

This enables two overriden methods to get callbacks.

1. onLocalVideoTrackListener
2. onRemoteVideoTrackListener

currentSession.addAudioTrackCallbacksListener(this);

This enables two overriden methods to get callbacks.

1. onLocalAudioTrackListener
2. onRemoteAudioTrackListener

In **BaseConversationFragment**, only the returned audio and video tracks are stored for releasing them at the time of closing the fragment to clean up memory.

Controlling of user’s video and audio tracks is done in **VideoConversationFragment**.

This is the most important part in the class. As here is where we decide which user will be shown or heard to which other user.

This detail we get from our backend on **ConnectActivity** which we stored on the database at that time. We retrieve that data and check where the user id given in the callback consists in the database for see/hear/chat data. If yes, then we enable the audio or video track accordingly otherwise we disable it.

Eg. of code is provided as under:

1. **Video Listeners**

@Override  
**public void** onLocalVideoTrackReceive(ConferenceSession conferenceSession, QBRTCVideoTrack qbrtcVideoTrack) {  
 *//For translator and client don't show the local video track* **if**(**currentUserType** == 1 || **currentUserType** == 2) {  
 **cameraState** = CameraState.***NONE***;  
 VideoTrackModel model = **new** VideoTrackModel();  
 model.setQb\_id(-1);  
 model.setVideoTrack(qbrtcVideoTrack);  
  
 **videoTrackArrayList**.add(model);  
  
 **adapter** = **new** OpponentsVideoList(getActivity(), **videoTrackArrayList**);  
 **opponentsVideoList**.setAdapter(**adapter**);  
 GridLayoutManager manager;  
 **if** (**videoTrackArrayList**.size() <= 4)  
 manager = **new** GridLayoutManager(getActivity(), 2, GridLayoutManager.***VERTICAL***, **false**);  
 **else** manager = **new** GridLayoutManager(getActivity(), 3, GridLayoutManager.***VERTICAL***, **false**);  
 **opponentsVideoList**.setLayoutManager(manager);  
 }  
  
}

@Override  
**public void** onRemoteVideoTrackReceive(ConferenceSession conferenceSession, QBRTCVideoTrack qbrtcVideoTrack, Integer integer) {  
 handleVideoTracks(qbrtcVideoTrack,integer);  
 handleTranslatorImage(integer);  
}

**private void** handleVideoTracks(QBRTCVideoTrack videoTrack, Integer integer){  
  
 VideoTrackModel model = **new** VideoTrackModel();  
 model.setQb\_id(integer);  
 model.setVideoTrack(videoTrack);  
  
 **if**(**seeArrayList**.contains((Object)integer)){  
 **videoTrackArrayList**.add(model);  
 }  
 **if**(**videoTrackArrayList**!=**null** && **videoTrackArrayList**.size()>0) {  
  
 **if**(**currentUserType** == 1 || **currentUserType** == 2) {  
 *//For Moderator and Respondent localVideoTrack is already set as the first item in the adapter  
 //So we just notify the adapter* **adapter**.notifyChange(**videoTrackArrayList**);  
 }  
 **else** {  
 *//For Client and Translator localVideoTrack is not set as the first item in the adapter* **adapter** = **new** OpponentsVideoList(getActivity(), **videoTrackArrayList**);  
 **opponentsVideoList**.setAdapter(**adapter**);  
 GridLayoutManager manager;  
 **if** (**videoTrackArrayList**.size() <= 4)  
 manager = **new** GridLayoutManager(getActivity(), 2, GridLayoutManager.***VERTICAL***, **false**);  
 **else** manager = **new** GridLayoutManager(getActivity(), 3, GridLayoutManager.***VERTICAL***, **false**);  
 **opponentsVideoList**.setLayoutManager(manager);  
 }  
 }  
 **else**{  
 Log.*e*(**TAG**,**"No Video for "** + String.*valueOf*(integer) + **" user."**);  
 }  
 }

1. **Audio Listeners:**

@Override  
**public void** onLocalAudioTrackReceive(ConferenceSession conferenceSession, QBRTCAudioTrack qbrtcAudioTrack) {  
  
}  
  
@Override  
**public void** onRemoteAudioTrackReceive(ConferenceSession conferenceSession, QBRTCAudioTrack qbrtcAudioTrack, Integer integer) {  
 handleAudioTracks(qbrtcAudioTrack, integer);  
}

**private void** handleAudioTracks(QBRTCAudioTrack audioTrack, Integer integer){  
  
 **if**(**hearArrayList**.contains((Object)integer))  
 audioTrack.setEnabled(**true**);  
 **else** audioTrack.setEnabled(**false**);  
}

Other callbacks taking place in this fragment:

1. onConnectedToUser:

This is invoked when the opponent user gets connected to the call. (This generally depends on Internet connectivity)

Note: User doesn’t leave the conference call.

@Override  
**public void** onConnectedToUser(ConferenceSession conferenceSession, Integer integer) {  
  
 **if**(**seeArrayList**.contains(integer)) {  
 String name = **db**.getNameFromId(integer);  
 Toast.*makeText*(getActivity(), name + **" joined the conference."**, Toast.***LENGTH\_SHORT***).show();  
 }  
  
}

1. onDisconnectFromUser:

This is invoked when opponent user gets disconnected from the call. (This generally depend on Internet connectivity)

Note: User doesn’t leave the conference call.

@Override  
**public void** onDisconnectedFromUser(ConferenceSession conferenceSession, Integer integer) {  
 *//User got disconnected.* **if**(**seeArrayList**.contains(integer)){  
 String userName = **db**.getNameFromId(integer);  
 Toast.*makeText*(getActivity(), userName + **" "** +getString(R.string.***user\_disconnected***), Toast.***LENGTH\_SHORT***).show();  
 }  
  
}

1. onConnectionClosedForUser:

This is invoked when the current user’s connection is closed.

Note: This is where the user leaves the conference call.

@Override  
**public void** onConnectionClosedForUser(ConferenceSession conferenceSession, Integer integer) {  
 *//Remove the user from view who left the conference.* **if** (**currentSession**.isDestroyed()) {  
 Log.*d*(**TAG**, **"onConnectionClosedForUser isDestroyed userId= "** + integer);  
 **return**;  
 }  
 userLeftConference(integer);  
}

* **Firebase Videos**

For showing users videos (while call is going on) we have implemented firebase.

Firebase features being used:

1. Database
2. Storage

We save the video file on Firebase Storage.

And give the file url in the database.

We fetch this url and show in on our ends.

Following is the json example of database:

"testingroom" : {

"video\_list" : {

"video1" : {

"status" : "stop",

"url" : "http://www.vareniacims.com/vcimsweb/uploads/vid..."

},

"video2" : {

"status" : "stop",

"url" : "https://firebasestorage.googleapis.com/v0/b/fir-8c645.appspot.com/o/RPReplay\_Final1530791162.MP4?alt=media&token=367e6ee7-72ef-4a2c-b0b3-15d7d3d86f31"

}

}

}

Two Parameters being used:

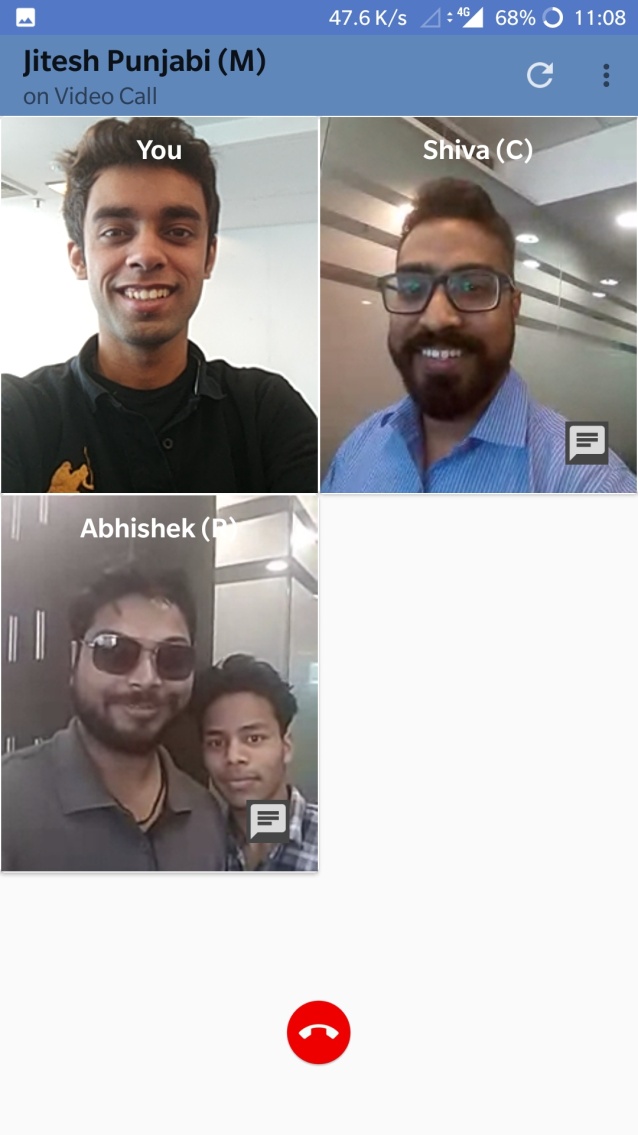
1. Status:

* Play
* Pause
* Stop

All these statuses on change give a callback on Android and only moderator has the functionality to change these status. All other users can only watch the whole video.

1. Url:

On status change we get child callback with status and url. Using url we show video on Android with VideoView



* **Chat (ChatActivity)**

While video conferencing some users have the functionality to chat with others.

This depends on the user ids for chat that we get from backend on **ConnectActivity**.

On video adapter item, there is a button to start chat.

Clicking on that user will be migrated to **ChatActivity**.

Step-wise explanation of how chat works:

1. A private dialog is created between current user and the opponent user.

Eg. of code:

VideoTrackModel m = **arrayList**.get(position);  
**occupantsIds**.add(m.getQb\_id());  
QBChatDialog dialog = DialogUtils.*buildDialog*(**"Dialog"**, QBDialogType.***PRIVATE***, **occupantsIds**);  
  
QBRestChatService.*createChatDialog*(dialog).performAsync(**new** QBEntityCallback<QBChatDialog>() {  
 @Override  
 **public void** onSuccess(QBChatDialog qbChatDialog, Bundle bundle) {  
 **occupantsIds**.clear();  
 holder.**notiBadgeTV**.setText(**""**);  
 holder.**notiBadgeTV**.setVisibility(View.***GONE***);  
 ChatActivity.*startForResult*(**activity**,0,qbChatDialog);  
 }  
  
 @Override  
 **public void** onError(QBResponseException e) {  
 **occupantsIds**.clear();  
 }  
});

1. Load Chat History. If there is any previous chat history with that user then first load that.

Eg. of code:

**private void** loadChatHistory(){  
 loadChatHistory(**qbChatDialog**, **skipPagination**, **new** QBEntityCallback<ArrayList<QBChatMessage>>() {  
 @Override  
 **public void** onSuccess(ArrayList<QBChatMessage> qbChatMessages, Bundle bundle) {  
 Collections.*reverse*(qbChatMessages);  
 **if**(**chatAdapter** == **null**){  
 **chatAdapter** = **new** ChatAdapter(**instance**,qbChatMessages);  
  
 **if** (**unShownMessages** != **null** && !**unShownMessages**.isEmpty()) {  
 List<QBChatMessage> chatList = **chatAdapter**.getList();  
 **for** (QBChatMessage message : **unShownMessages**) {  
 **if** (!chatList.contains(message)) {  
 **chatAdapter**.addMessage(message);  
 }  
 }  
 }  
 **chatMessagesRV**.setAdapter(**chatAdapter**);  
 **chatMessagesRV**.setLayoutManager(**new** LinearLayoutManager(**instance**));  
 }  
 **else**{  
 *//chatAdapter.addList(qbChatMessages);* }  
 **pDialog**.cancel();  
 }  
  
 @Override  
 **public void** onError(QBResponseException e) {  
 **pDialog**.cancel();  
 }  
 });  
}

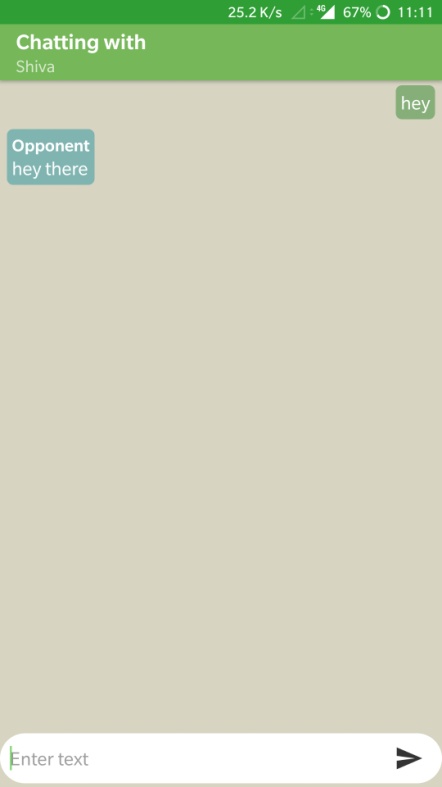
Set the messages you get here in the adapter as shown in the example.

1. For any new message that arrives we have a listener.

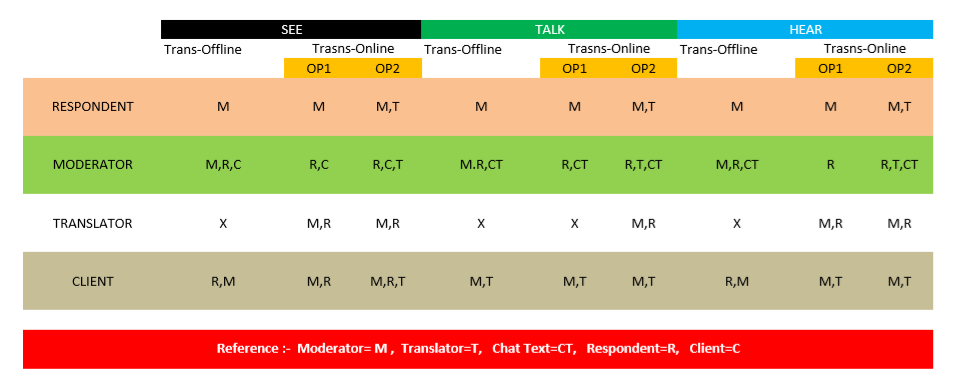
Eg. of code:

**private void** addMessageListener(){  
 QBIncomingMessagesManager incomingMessagesManager = **qbChatService**.getIncomingMessagesManager();  
 incomingMessagesManager.addDialogMessageListener(**new** QBChatDialogMessageListener() {  
 @Override  
 **public void** processMessage(String s, QBChatMessage qbChatMessage, Integer integer) {  
 showMessage(qbChatMessage);  
 }  
  
 @Override  
 **public void** processError(String s, QBChatException e, QBChatMessage qbChatMessage, Integer integer) {  
  
 }  
 });  
}

**private void** showMessage(QBChatMessage chatMessage){  
 **if**(**chatAdapter**!=**null**){  
 **chatAdapter**.addMessage(chatMessage);  
 scrollMessageListDown();  
 }  
 **else**{  
 **if**(**unShownMessages** == **null**){  
 **unShownMessages** = **new** ArrayList<>();  
 }  
 **unShownMessages**.add(chatMessage);  
 }  
}



**Documentation on VRapid Model**



There are three models in which the audio and video tracks depend on

1. **See**
2. **Hear**
3. **Chat**

User on **ConnectActivity** gets a response of array type consisting of **see**, **hear** and **chat** arrays.

Each array consists of Quickblox ids of users with which the current user will be able to **see**, **hear** and **chat**.

Web Service used:

http://live.vareniacims.com/vcimsweb/wsr.php?tasks=vcusertaglist

Example:

Parameters send

1. uid

* This is the auto increment id of each user in the table where each user is registered in our server.

Sending this id returns a list of users which are of the same room as the current user. And an array containing see, hear and chat list.

"msg": [

{

"see": [

"43785090", "43785100", "47712370", "47712634", "47712685",

"47882349", "47890334"

],

"hear": [

"43785090", "43785100", "47712370", "47712634", "47712685", "47882349",

"47890334"

],

"chat": [

"43785090", "47712458", "47712525", "47712580", "47712753", "47712794",

"47882388", "47890163", "47890301", "47890359"

],

"chat": [

"43785090", "47712634"

]

}

]

We store this detail on Android SQLiteDatabase.

According to these ids we enable audio, video tracks and ability to chat with other users for the current user.