Daisy Barbanel, Davaajav Ganzorig,
Miranda Grisa, Connor Hopcraft, Troy Hu,
Ryan Li, Gray Mackall, Zené Sekou

Milestone 3a Explanation Document

Submission Location:

https://github.com/ze-ne/cs220-smart-and-secure-messging

Tests Files Under:
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/clientBackendUnitTests/
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/frontendUnitTests/
cs220-smart-and-secure-messging/app/src/androidTest/java/com/cs220/ssmessaging/UIEspressoTests/

Source Files Under:
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/clientBackend
cs220-smart-and-secure-messging/app/src/test/java/com/cs220/ssmessaging/frontend

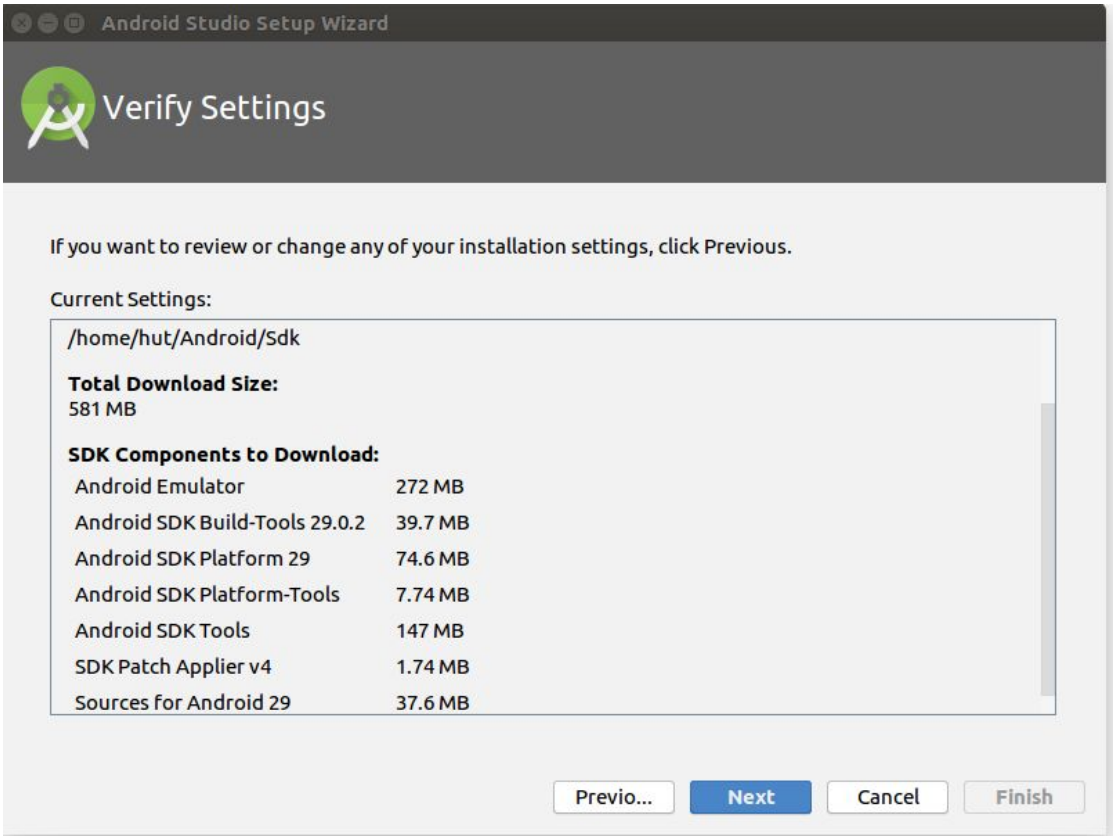
How to Run Tests:

IMPORTANT: If possible run on CSIL Machines (It worked for us in person. No SSH)
It also runs on our own linux/MacOs computers but sometimes it requires additional tuning. Try it at your own risk.

Setup

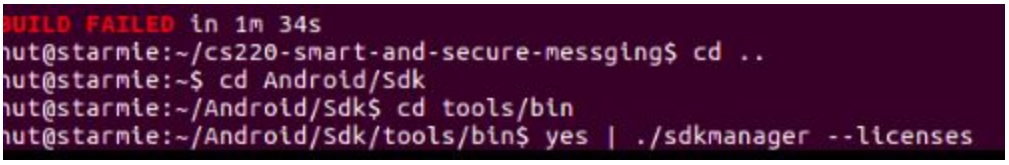
1. First download Android Studio: https://developer.android.com/studio
    a. Note: We will use command line to run the tests. But installing the studio also installs the Android SDK and relevant packages (which we need).
2. Then extract it someplace and go to android-studio/bin and run ./studio.sh
3. You will now be presented with options.

a. Choose standard setup keep clicking next until you get to this page

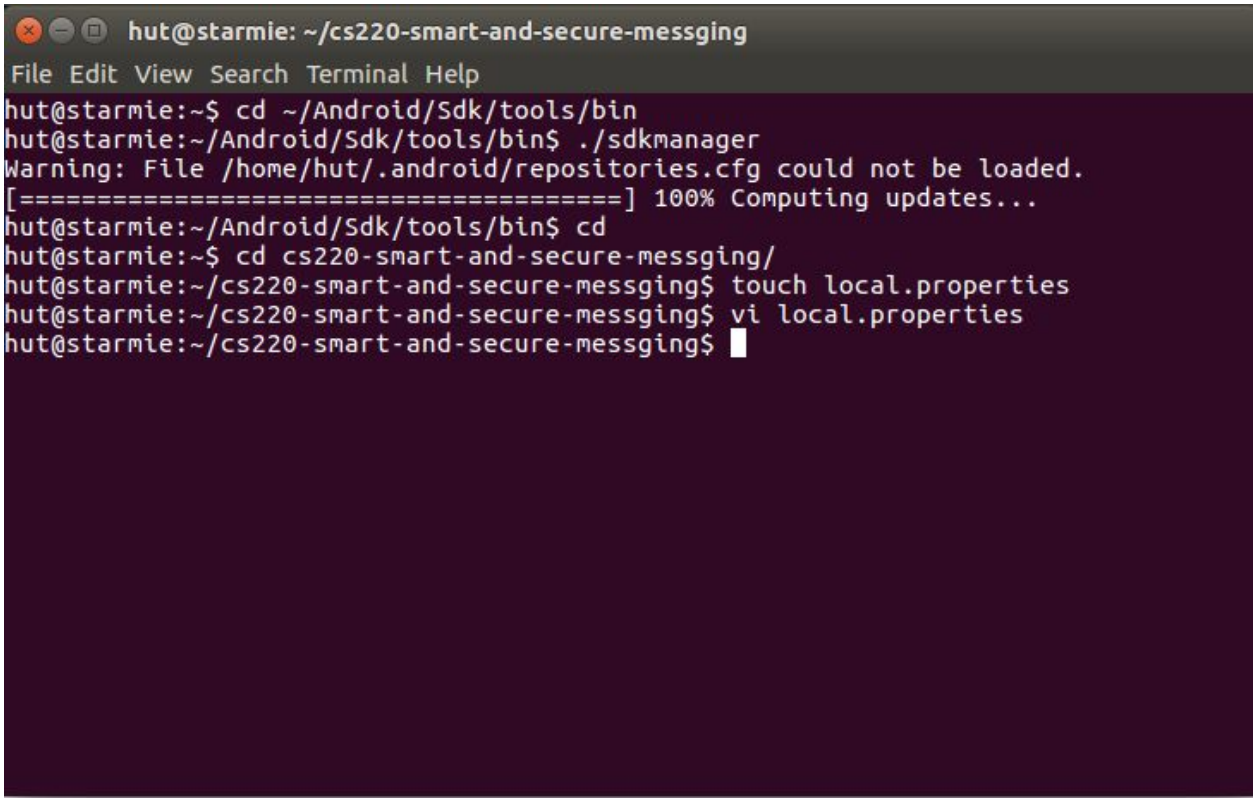


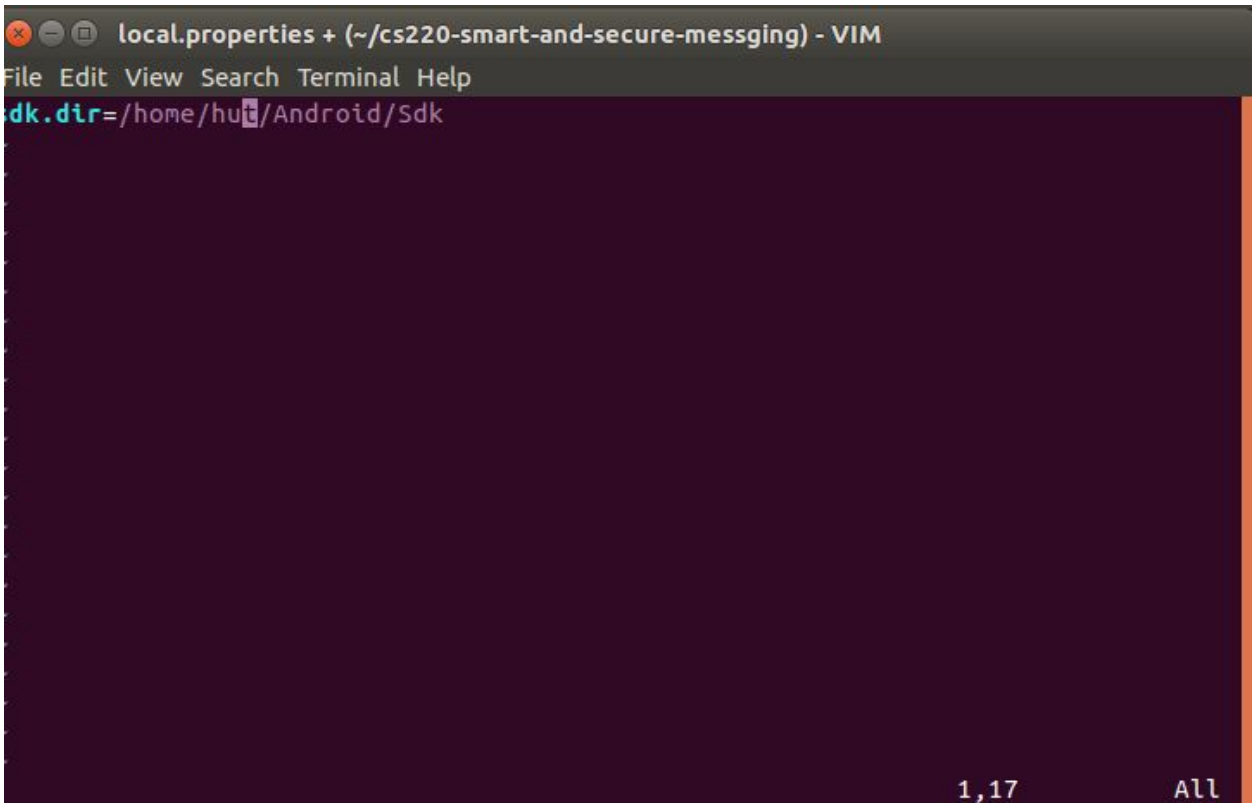
b. Keep a note of the path given there! It is the path to your Android SDK folder.
c. Keep clicking next, install, and finish.
d. Exit the pop up for the studio. We will use the command line from here on.

4. Now you have to accept the license for Android SDK for our tests to work. What you'll want to do is:
    a. Go to the path shown in the image in 3a. In my case it is /home/hut/Android/Sdk
    b. cd into tools/bin
    c. run the command: yes | ./sdkmanager --licenses



5. Find a place to git clone the files from the repository.
    a. Go to top level directory. That is cs220-smart-and-secure-messaging.
    b. Create a file called local.properties
    c. within the file write ONLY:
        i. sdk.dir = the path you noted for android SDK
        ii. For example: sdk.dir = /home/hut/Android/Sdk





You are now done with set up!

Running the Tests

Do everything in the setup first.

1. Go to the top level of the git repository you cloned.
2. Run the command: ./gradlew test --info
    a. It will take some time to configure and download necessary packages
3. You should see all tests run and fail.

NOTE: Even though it says build failed, in reality, our tests compile and work. You can see that all of the tests ran and failed. Build only fails because the tests fail.

Please contact us if you have trouble getting this to work.

Private Variables, Getters, and Setters In Kotlin

All accesses (except for the constructor) in Kotlin must go through the getter and setter. This is because variables are properties and all properties have a private field. Therefore we achieve encapsulation Kotlin by defining custom getters and setters. This style is standard Kotlin convention called "Backing Fields". That is why in some cases we don't need to declare variables with private modifiers. In the cases that we do need private modifiers, it is when we need "Backing Properties" OR when we want the variable getter and setter to be private. Please go here for more information: https://kotlinlang.org/docs/reference/properties.html.

For getters and setters, instead of object.get_var() or object.set_var(), Kotlin uses the syntax “object.var” to call the getter and “object.var = value” to call the setter.

Deviations from Previous Design Goals:

Moving Some Use Cases to Iteration 2:

We talked with the TAs and decided that our first iteration was a little too ambitious. We are moving some less important or non-essential use cases to iteration 2. The following use cases are not essential for building a workable/basic messenger:

- **Search For Contact/User**
    - This is not needed for a basic messenger since this use case is more for quality of life.
- **Search For Conversation**
    - Again, this use case is more about quality of life.
- **Manage Block List**

    - What is more important is that we manage the contacts list to allow for conversations. Block list is not needed for a basic messenger.
- **Delete Sent Message**
    - We are moving this use case to iteration because it is more in line with extra security and privace features. It is not as important as encryption/decryption.

As a result, these are the remaining substantial use cases we are implementing for iteration 1:

- Manage Contacts
- Login
- Logout
- Register
- Manage Account
- Start Conversation (Between 2 People)
- View/Leave a Conversation
- Send/Receive Message Without Image
    - Message Encrypted
- Send/Receive Message With Viewable Image
    - Message Encrypted

In our opinion, this will be a more manageable amount of work for the first iteration and allow us to focus on the central building blocks for messenger functionality.

Changing Backend Architecture

Originally, we planned to have an Elixir server that would facilitate relaying messages back and forth between clients using the Channels API in its Phoenix framework (with PostgreSQL as the database due to Phoenix's native support for it). However, after more research, we realized that the Channels API was limited in the sense that it was at most amenable to a chat room, and we needed a solution that would persist messages across clients and could address synchronization issues among clients (i.e. offline mode). For these reasons, we decided to go with Firebase Firestore for the backend for its high scalability and availability, and its native support for database change synchronization across clients using the Android SDK's snapshot listeners. Additionally, it provides cloud functions that replace the need for a server in our use case (i.e. functions for some registration flow that react to HTTP requests or specific changes in the database, sort of like SQL triggers). In the case that users need to be notified of messages when the app isn't running (and therefore the snapshotListener isn't working), we have the Google Cloud Messaging Service that could be called by cloud functions to send clients specific activity alerts. Because basically all of the backend functionality would therefore be done in the cloud on

the Google Firebase side, we could not do local unit tests to invoke the Firestore database for the purposes of testing.

User Interface Unit Tests:

Unfortunately, there are not many ways to test the user interface without having at least some of the UI elements implemented as elements must be referred to specifically by id in the Espresso testing framework. That being said, we have endeavored to provide as complete of a testing suite as possible. Sample tests were written for Login, ConversationsList, Settings, and Conversation screens and are commented out so that the app will still compile. While these tests may not be comprehensive currently, they represent our best attempt to test UI elements that don't currently exist. These tests will be updated with the iteration 1 code to reflect our actual implementation.

Server Tests:

Several of our functions rely on communicating with the server and updating/retrieving data in the database. We found that the functionality related to the server was better suited for integration tests rather than unit tests because it relies on asynchronous communication with the server which would be very difficult to mock. For functions that updated both local classes and the server we only tested the local side-effects of those functions. These functions are listed under "Partial Tests." Under "No Test" we listed functions that had no local side-effects and therefore could only be tested using integration tests.

Partial Tests

- addConversation
- sendTextMsg
- receiveMsg

No Test

- addPublicKey
- getUserPublicKey