GSoC 2021: Library for Messaging App Store

Mentors: Hasan HASAN, Furkan KAMACI

Organization: Apache Software Foundation

Student: Yusuf KARADAĞ

Project Description:

The purpose of this task is to define an ontology for conversations and to create a Java library based on Apache Clerezza to store them. Signal (https://signal.org/docs/) is one of many messaging apps which is open source. This task recommends integrating the library to be developed into Signal (https://github.com/signalapp) for testing.

Key Takeaways:

- RDF
- Apache Clerezza
- Signal API

This report consists of 3 sections:

1. Conversation Ontology
2. Clerezza Signal Library
3. RESTful web service for Clerezza Signal Library
4. Debugging in Signal Android

1. Conversation Ontology

An ontology is a formal description of knowledge as a set of concepts within a domain and the relationships that hold between them. To enable such a description, we need to formally specify components such as individuals (instances of objects), classes, attributes and relations as well as restrictions, rules and axioms. As a result, ontologies do not only introduce a sharable and reusable knowledge representation but can also add new knowledge about the domain. [1]

The *Conversation Ontology* consists of following nodes and attributes:

Nodes:

- Message node, to indicate sent messages
- Person node (FOAF), to indicate person that sends the messages
- Conversation, to indicate conversation between 2 people

Attributes:

- phone attribute, to indicate *Person'*s phone number
- username attribute, to indicate *Person'*s username
- member attribute, to indicate outgoing relationship from *Person* to *Conversation* (*Person* is a member of *Conversation*)
- createdBy attribute, to indicate to indicate outgoing relationship from *Conversation* to *Person* (*Conversation* is created by *Person*)
- startDate attribute, to indicate starting date of the conversation
- conversationName attribute, to indicate name of the conversation
- timestamp attribute, to indicate timestamp of the text message
- text attribute, to indicate incoming message text
- post attribute, to indicate outgoing relationship from *Person* to *Message* (*Person* posts *Message*)
- consistOf attribute, to indicate outgoing relationship from *Conversation* to *Message (Conversation* is consist of *Message*s*)*

Below we can see the visualization of the created ontology:

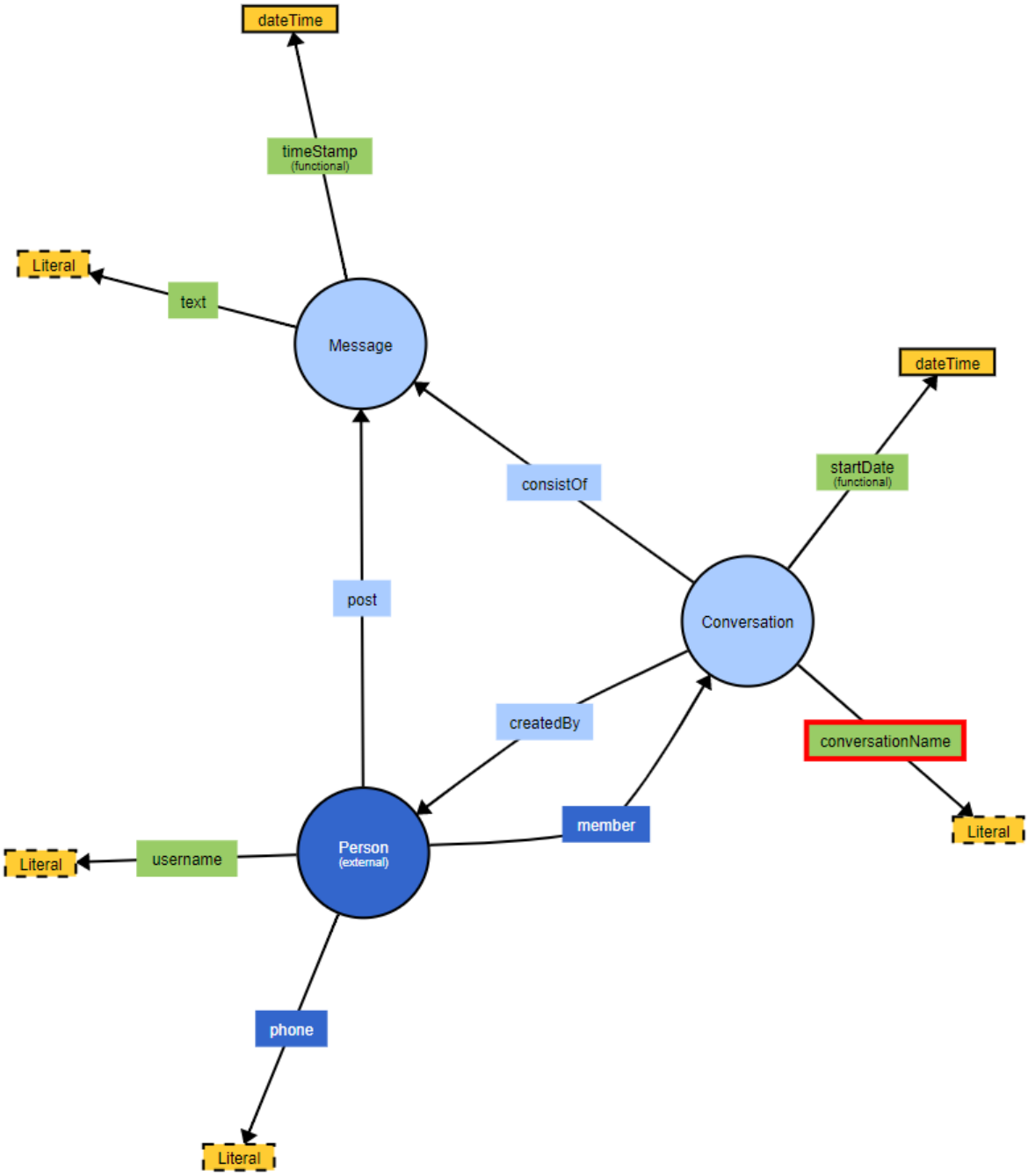


Image 1: Visualization of Signal RDF Ontology

This is the randomly generated data based on *Conversation Ontology:*

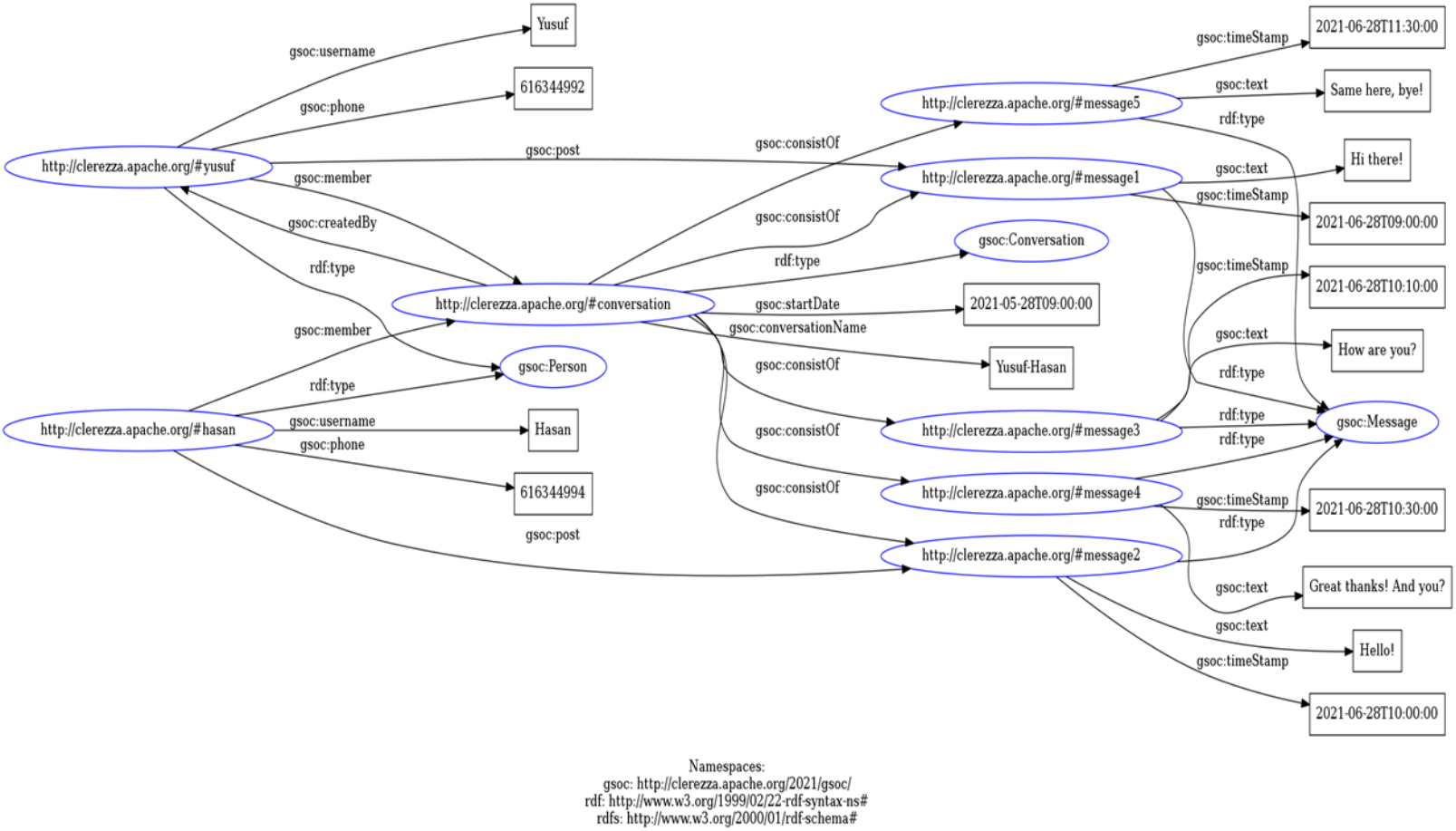


Image *2:* Visualization of randomly generated data based on Signal RDF Ontology

2. Implementation Architecture

Below we can see simplified Signal Messenger messaging architecture with 2 clients, Android(Java) and Desktop(Typescript):

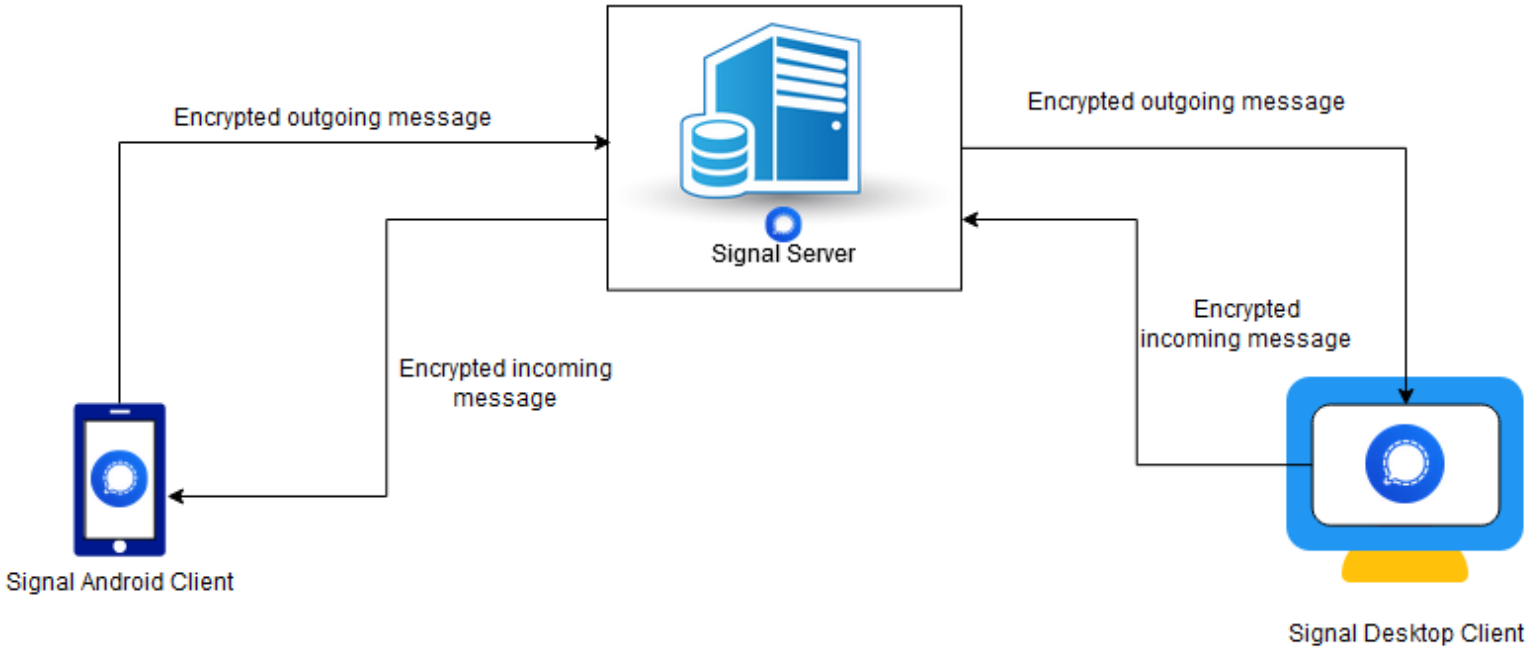


Image 3: Simplified Signal Messenger messaging architecture

- Signal Android Client:
  - Types a message
  - Encrypts the message
  - Push over to Signal Public Server.
- Signal Server:
  - Receive encrypted message.
  - Store the encrypted message.
  - Transfer encrypted message over to other client.
- Signal Desktop:
  - Types a reply message
  - Encrypts the message
  - Push over to Signal Public Server.

By interfering the one of the Signal clients(in this case Signal Android), we store the message into an RDF graph with Clerezza Signal Library.

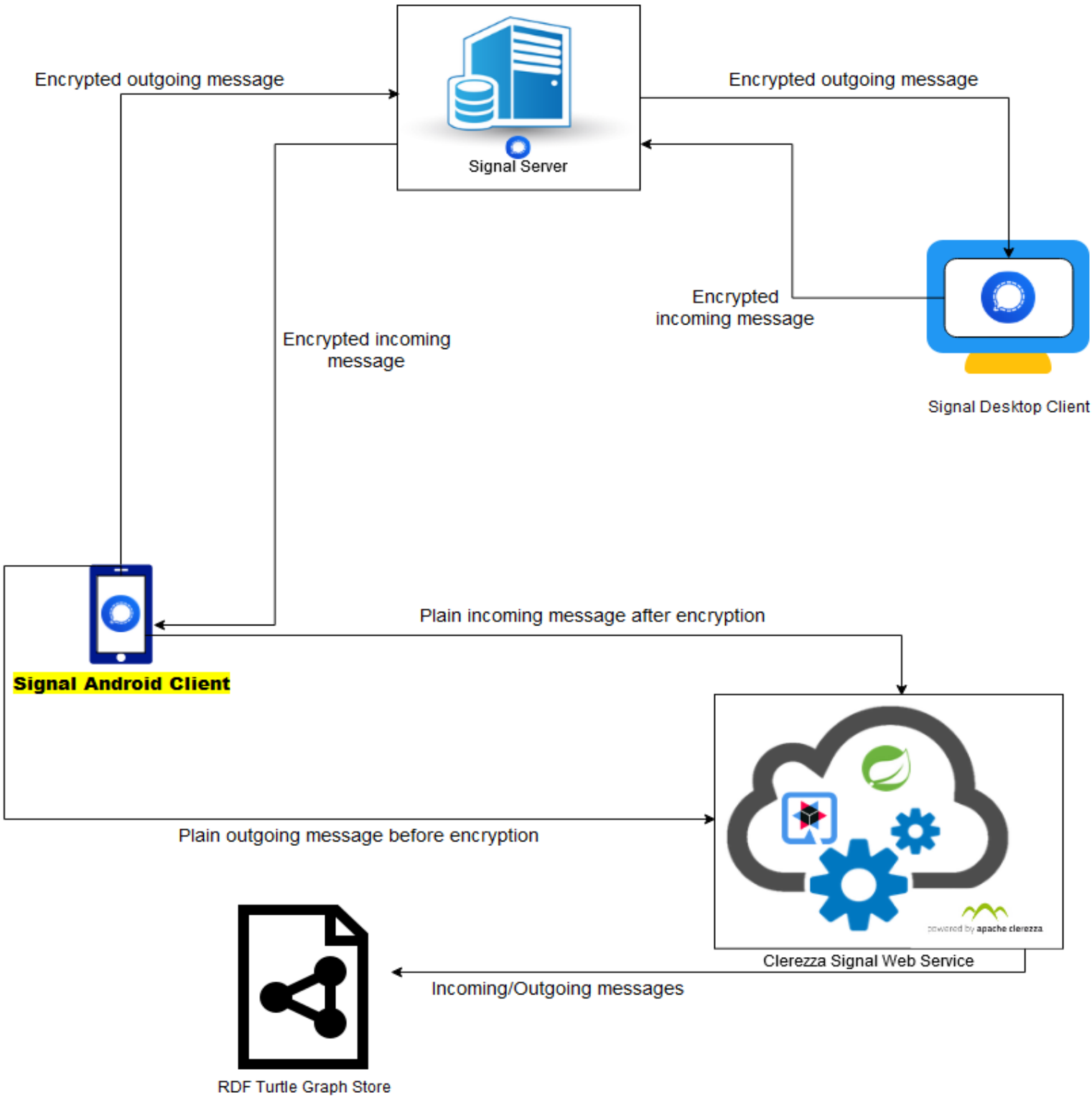


Image 4: Apache Clerezza based Messaging Graph Store Web Service and Architecture for Signal Messenger

- **Signal Android Client**(Interceptor Signal client):
    - Types a message
    - Sends message to the Messaging Graph Store Web Service. (With the client code in page 12)
    - Encrypts the message
    - Pushes over to Signal Public Server.
- Signal Server:
    - Receive encrypted message.
    - Store the encrypted message.
    - Transfer encrypted message over to other client.
- Signal Desktop:
    - Types a reply message
    - Encrypts the message
    - Push over to Signal Public Server.
- **Signal Android Client**(After receives message):
    - Encrypts the incoming message.
    - Sends the decrypted message to the Messaging Graph Store Web Service. (With the client code in page 12)

3. Clerezza Signal Library

Signal Clerezza Library consists of 3 packages, 3 classes.

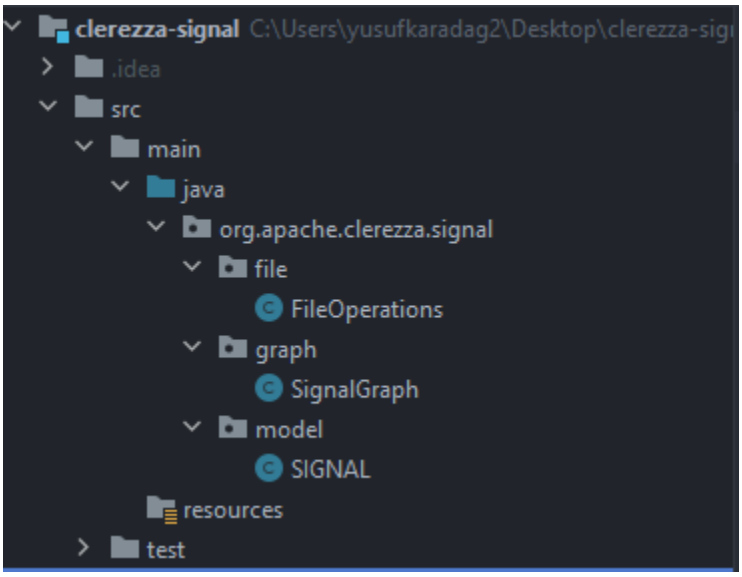


Image 3: Clerezza Signal Class Structure

- *SignalGraph* class has one method to retrieve incoming and overwrite to the existing graph.
- *FileOperations* class has the methods for file operations such as create file with the given name if not exist, parse, serialize and print the graph from the file.
- Classes in org.apache.clerezza.*model* package has the ontology model classes used by the library.
- The library uses 2 more external models RDF and FOAF, they are retrieved from *clerezza-ontologies* plugin.

4. RESTful Web Service for the Clerezza Signal Library

- Built with Quarkus(Java 11) and Spring Boot(Java 8).
    - Quarkus is a full-stack, Kubernetes-native Java framework made for Java virtual machines (JVMs) and native compilation, optimizing Java specifically for containers and enabling it to become an effective platform for serverless, cloud, and Kubernetes environments.[2]
    - Java Spring Boot (Spring Boot) is a tool that makes developing web application and microservices with Spring Framework faster and easier through three core capabilities: Autoconfiguration. An opinionated approach to configuration. The ability to create standalone applications.[3]

- Web services consist of one GET and one POST endpoint. GET is used to print the existing graph and POST is used to save messages from Signal.
- The idea behind the POST is:
    - Create file with the requested name. If file exist, do nothing.
    - Check if there's data ready to read. If not, save incoming graph.
        - If there's data to read, that means there are existing messages. Iterate over the nodes to check whether conversation is existing or not. If the conversation exists, then overwrite the graph.
- Both web services are using *Jackson-Databind* library to serialize and deserialize JSON from the string.

5. Debugging in Signal Android with Android Studio

The purpose of this section is to give an idea about where to put client code and integrate Signal Clerezza Library into Signal Android.

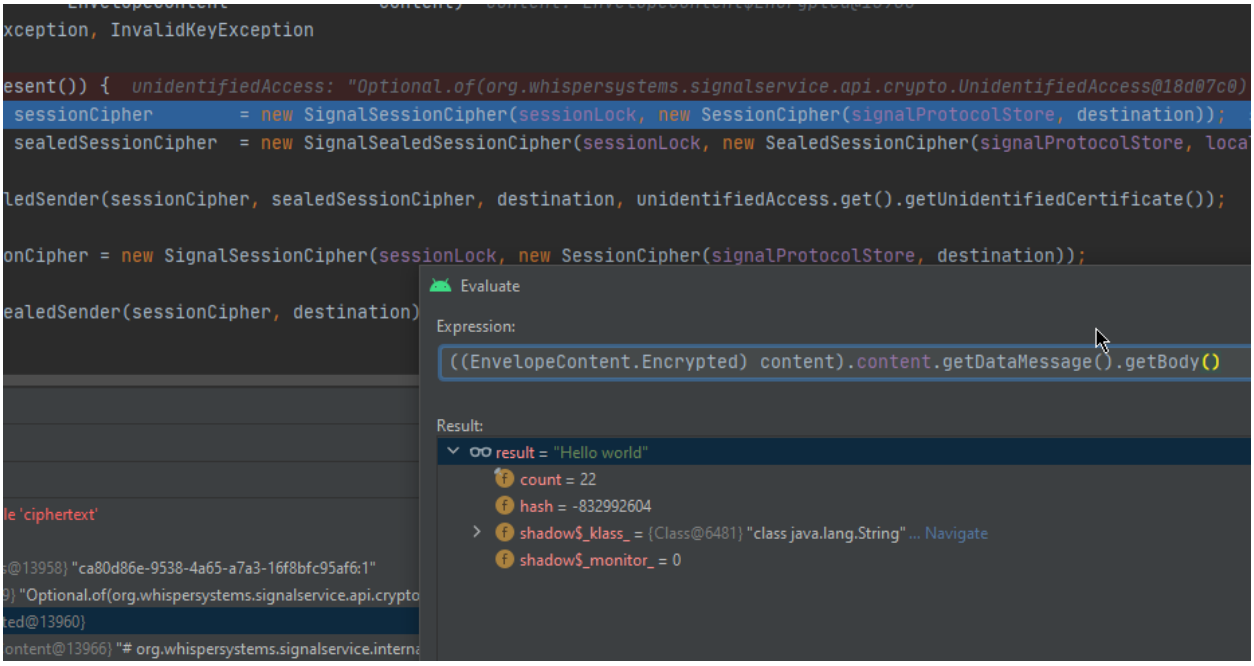
To properly debug the Android apps, Android Studio IDE should be installed beforehand.

To install Android Studio for all operating systems, the instructions from the official webpage can be followed: https://developer.android.com/studio/install

After Successfully installing Android Studio

1. Go to: https://github.com/signalapp/Signal-Android and copy git URL.
2. Open a terminal and clone the codebase: *git clone https://github.com/signalapp/Signal-Android.git*
3. Open Android Studio, from the tabs menu click File -> Open. Find Signal Android app and open.
4. Build project.
5. You can use either your phone or an emulator to debug the application.

6. To debug from the phone, go to Settings -> About -> Build number -> Click 3 times to enable developer settings -> Go to Developer Settings -> Enable USB Debugging. To debug from an emulator, click AVD Manager and add a virtual device.
7. After everything set up, put breakpoint to line 113 in *SignalServiceCipher.java* (*libsignal/service/src/main/java/org/whispersystems/signalservice/api/crypto/SignalServiceCipher.java)* to see message before it's encrypted. And put another breakpoint to line 156 in same class to see the message after it's decrypted.
8. Click debug. Wait for the emulator/phone to install and run the app.
9. Send a message through the app. When debugger attach, it will first show empty messages because it tracks things like when person start writing, waiting etc. In debug page follow *content* variable carefully. After few iterations you will see the content as:



10. When someone sends you a message, follow the same procedure and wait for the debugger, when you evaluate *content,* you can see the decrypted message as:

6. Few Notes

- The library is built for *libsignal-service-java* and *signal-android* client.
- In Signal Android client, the request code can be added to *SignalServiceCipher* class from where we can retrieve messaging information.
- If we create a project based on *libsignal-service-java,* we can also use our library to store messages.
- The workflow of the storing process as:

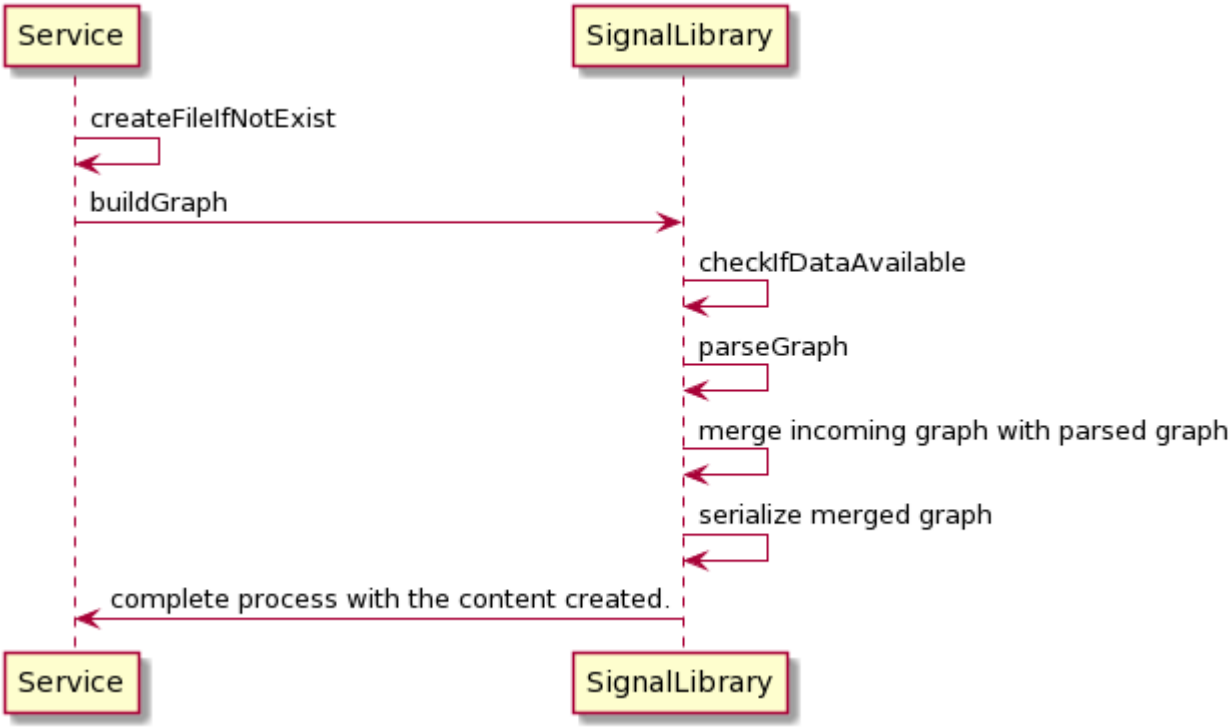


Image 4: Storing messages sequence diagram

- To use Clerezza Signal with *libsignal-service-java,* recommended usage is to create a separate project and after sending message to signal server, send it to Signal web service as well.
  NOTE: To be able to use library and serialize messages into RDF graphs, we don't need to use a Java client necessarily. For instance, if the user wants to integrate the functionality into Signal Desktop client which is written in Typescript, the only thing to be done is to construct a POST request with the related information in JSON.

- For requesting to server, following code snippet can be used:

```
public void postMessage( String text, String timeStamp, String conversationName,
String username ) throws IOException {
    //Change the URL with any other publicly accessible POST resource, which
accepts JSON request body
    URL url = new URL( "http://gsoc.test.com:8080/signal/messages" );

    HttpURLConnection con = ( HttpURLConnection ) url.openConnection();

    con.setRequestMethod( "POST" );
    con.setRequestProperty( "Content-Type", "application/json; utf-8" );
    con.setRequestProperty( "Accept", "application/json" );
    con.setDoOutput( true );

    //JSON String need to be constructed for the specific resource.
    ObjectNode o = new ObjectNode( JsonNodeFactory.instance );
    o.with( "Message" ).put( "text", text );
    o.with( "Message" ).put( "timeStamp", timeStamp );
    o.with( "Person" ).put( "username", username );
    o.with( "Conversation" ).put( "conversationName", conversationName );
    String jsonInputString = new ObjectMapper().writeValueAsString( o );

    try ( OutputStream os = con.getOutputStream() ) {
        byte[] input = jsonInputString.getBytes( StandardCharsets.UTF_8 );
        os.write( input, 0, input.length );
    } catch ( IOException e ) {
        e.printStackTrace();
    }

    int code = con.getResponseCode();
    System.out.println( code );

    try ( BufferedReader br = new BufferedReader( new InputStreamReader(
con.getInputStream(), StandardCharsets.UTF_8 ) ) ) {
        StringBuilder response = new StringBuilder();
        String responseLine;
        while ( ( responseLine = br.readLine() ) != null ) {
            response.append( responseLine.trim() );
        }
        System.out.println( response );
    }
}
```

7. Created Pull Requests and Commits for the Project and Apache Clerezza

1. Integrating Schemagen into master branch from legacy branch:
    - The purpose of schemagen is to generate Java clasess from RDF ontologies. https://github.com/apache/clerezza/pull/21#event-4985268757

2. Implementing tutorial module and integrating into master branch:
    - Tutorial module consists of three tutorial classes to make developers familiar with Apache Clerezza. https://github.com/apache/clerezza/pull/22

3. Quarkus Web Service:
    - Quarkus web service is built to accept incoming messages from the Signal client and serialize them into file. It uses Quarkus microservices and Java 11 with Maven. https://github.com/miador/clerezza-gsoc-quarkus

4. Spring Boot Web Service:
    - Spring Boot web service is built to accept incoming messages from the Signal client and serialize them into file. It uses Spring Boot microservices and Java 8 with Maven. https://github.com/miador/clerezza-gsoc-spring-boot

5. Clerezza Signal:
    - Clerezza Signal Library consists of classes and methods to be able to store messages into RDF graphs and overwrite them. https://github.com/miador/clerezza-signal

8. What needs to be done?

The integration of library into Signal clients is not done yet because of the limited knowledge about Android development and Gradle. The report gives few ideas about integration into Signal clients. There's enough detail to debug Signal Android app and find where message is encoded & decoded. But due to limited Android knowledge, I couldn't find few attributes such as username, phone and conversation name & start date.

9. References

- [1] What are ontologies? (https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/)
- [2] What is Quarkus? (https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus)
- [3] What is Spring Boot? (https://www.ibm.com/cloud/learn/java-spring-boot)
- Tools used to visualize&convert RDF graphs(http://www.visualdataweb.de/webvowl/) & (https://www.easyrdf.org/converter)
- FOAF Vocabulary Specification (http://xmlns.com/foaf/spec/)
- Apache Clerezza API Documentation (http://clerezza.apache.org/apidocs/)