5.3 Database application implementation

Based on the database and the database application modeling, a chosen software platform must be operated on mobile devices, and must support features that the designs require. Both Android and iOS operative system are the good candidates for the position. They all support threads, SQLite, TCP/IP connection, Bluetooth, etc. However, there is difficult to say which one is better, it depends on the user favorites. Since the implementation is a proof-of-concept, it is no need to deeply argue why choosing iOS or Android as long as they meet the requirements. Android platform is chosen for the designs in the thesis for some reasons. First of all, according to netmarketshare.com[https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1], there is 66.71% devices installed Android operative system compared to 29.55% devices installed iOS. Moreover, applications for Android are written by using Java programming language, which is very easy to be read and understood by many developers. Nonetheless, it is easy to synchronize and test with CESAR acquisition tool, because the tool is written for Android.

Beside satisfying the function and non-function requirements, the implementation also focuses on the graphic user interface (GUI). It is natural that people have a better feeling when they interact with icons and symbols compared to text. By using GUI, the system can avoid asking input from users, instead it lets the users choose via provided input or default values, because typing text on a mobile platform is a cumbersome task.

Subsection 5.3.1 presents a short discussion on how to manage multi-threaded database accessing in SQLite database system on Android platform. Subsection 5.3.2 presents how to implement the real-time wrapper model on the chosen platform for collecting data from CESAR acquisition tool. Subsection 5.3.3 presents the implementation of non-real-time wrapper, in which the EDF import and export functions are implemented accordingly to the abstract designs in Subsection 5.2.2. Subsection 5.3.4 presents a simple implementation, in which the collected data can be visualized on a graphical graph view.

5.3.1 SQLite and Multi-threaded database accessing

In SQLite, the same database can be shared by multiple processes at the same time. By using read/write locks, SQLite can control the ways processes accessing to the database. The processes can perform read operations at the same time, but only one process can be performed write operations at any moment in time. From version 3.5.0, SQLite manage locks internally to avoid data corruption. Hence, several threads can use a single SQLite connection simultaneously. That is, the application does not need to manage the database accessing between threads. However, balancing database workloads between threads need to be considered, it is because when any thread wants to write to the database, it locks the entire database file for the time it uses for writing.

A statistic on the relative number of devices running a given version of the Android platform from Google presents that more than 99,9% devices running an Android version with API 10 or better [https://developer.android.com/about/dashboards/index.html]. According to the dependence between Android API and SQLite version[https://developer.android.com/reference/android/database/sqlite/package-summary.html], SQLite 3.6 comes with Android API 8; the higher the Android API is, the better SQLite version it has. Therefore, most of current mobile phones have the SQLite version better than 3.6 which supports the internal database-level locks to avoid database corruption. However, database accessing can be failed if each process has its own connection to the database file. It is because the SQLite does support synchronization between multiple connections. When one connection is in use for writing, the database rejects the other modifying activities from other connections. As a result, the database management does not update changes of the other connections. To use multiple threads for maximizing database performance is unbenefited, since there is only one modifying connection at a specific time.

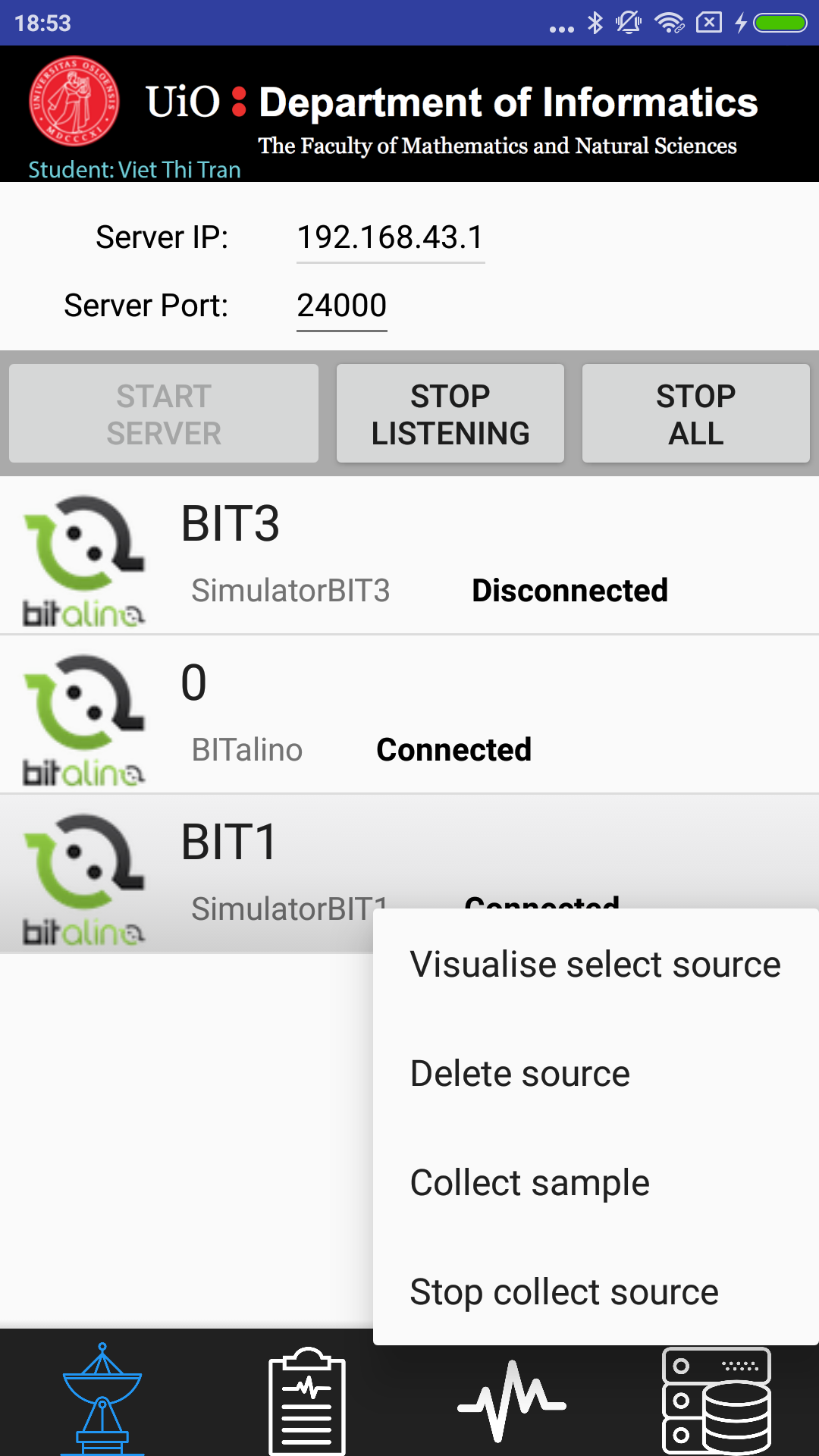
Threads can be used to maintain different data sources, or different data using purposes. These threads take turn using a database connection. By sharing the connection, the application makes sure that all threads can correctly update their data into the database. In the thesis, a helper object OSADataBaseManager is implemented in a way that is transparent for threads who using it. That is, a thread can initialize an instance of the object, then it can get the SQLiteDatabase via the openDatabase() method of the instance. After doing database operations, the thread can ask the instance for closing the database connection. At the view of the thread, it is logic when the thread open database connection for doing some tasks, then close the connection. However, the OSADataBaseManager instance creates and maintains only one SQLiteDatabase connection for all threads. The connection is created whenever there is at least one thread want to connect to the database, and closed when there are no database requests. List code L1 presents how to manage the shared database connection when working with multi-threaded database access.

private int mOpenCounter;  
private static OSADataBaseManager *instance*;  
private static OSADBHelper *mOSADBHelper*;  
private SQLiteDatabase mDatabase;  
  
public static synchronized void initializeInstance(OSADBHelper helper) {  
 if (*instance* == null) {  
 *instance* = new OSADataBaseManager();  
 *mOSADBHelper* = helper;  
 }  
}  
  
public static synchronized OSADataBaseManager getInstance() throws Exception{  
 if (*instance* == null) {  
 throw new Exception(OSADataBaseManager.class.getSimpleName()   
 + " is not initialized, call initializeInstance(..) to initialize instance.");  
 }  
 return *instance*;  
}  
  
public synchronized SQLiteDatabase openDatabase() {  
 mOpenCounter++;  
 //If it is the first time  
 if(mOpenCounter == 1) {  
 mDatabase = *mOSADBHelper*.getWritableDatabase();  
 }  
 //else just return the opened instance  
 return mDatabase;  
}  
  
public synchronized void closeDatabase() {  
 //We do not want to close the DB while the other use it  
 mOpenCounter--;  
 if(mOpenCounter == 0) {  
 //REAL CLOSE  
 mDatabase.close();  
 }  
}

L1

5.3.2 CESAR wrapper

As presented in the high level design of real-time wrapper, the wrapper must implement two kinds of threads. The first one is used for maintaining a list of connected sources which not only from CESAR acquisition tool, but also from other acquisition tools, as long as they follow the package interfaces that are discussed in Chapter 3. The second one is used for maintaining the connection between the application and a certain sensor source. Figure F1 presents the GUI of the wrapper. In the figure, the application provides the its IP number and the port it listens to. There are two active sources (with status “connected”) and one inactive source (with status “disconnected”) in the figure. By long clicking on a source, a list of functions, which the user can interact with, is showed. Each component that is presented in the figure is equivalent to a function the wrapper need to provide. The functions are divided into two groups; the first group consists of starting server, stopping listening, stopping all client threads and disconnect the server, managing the current connected list, and providing functions for that users can interact with the current connected list. These functions belong to server thread which is implemented in ServerFragment.java in the implementation code. The second group consists of collecting data from a specific source, forwarding data to graphical view (if requested), managing a batch processing for storing data into the database; these functions belong to client management thread which is implemented in ClientThread.java.



F1

**ServerFragment.java**

As Figure 5.3 in abstraction level design presented, the application must get the IP address of the device and show it on GUI. It can be done by checking all the network interface devices, and finding the IPv4 from the interfaces. Users are freely to choose a port number which must be bigger than 1024, it is because port smaller than 1024 are system ports[https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt]. To make it easy for the user, a default port is provided before the application is started. Once everything is initialized, the user can click on “START SERVER” button to begin the listening process which waiting for sensor sources at the presented address and port number. When the user clicks on the “START SERVER” button, the button is disenabled, and a server thread is created. The procedure when the button is clicked is illustrated as following:

server = new ServerSocket(portNr);

loop:

1. wait for connections from clients; if get connection, go to Step 2

2. create a ClientTread object with necessary parameters

3. add the ClientThread into the managing list

4. start the thread and go to Step 1

if server socket is closed (by clicking either “STOP LISTENING” or “STOP ALL” buttons), the server is shutdown, and the “START SERVER” button is enabled.

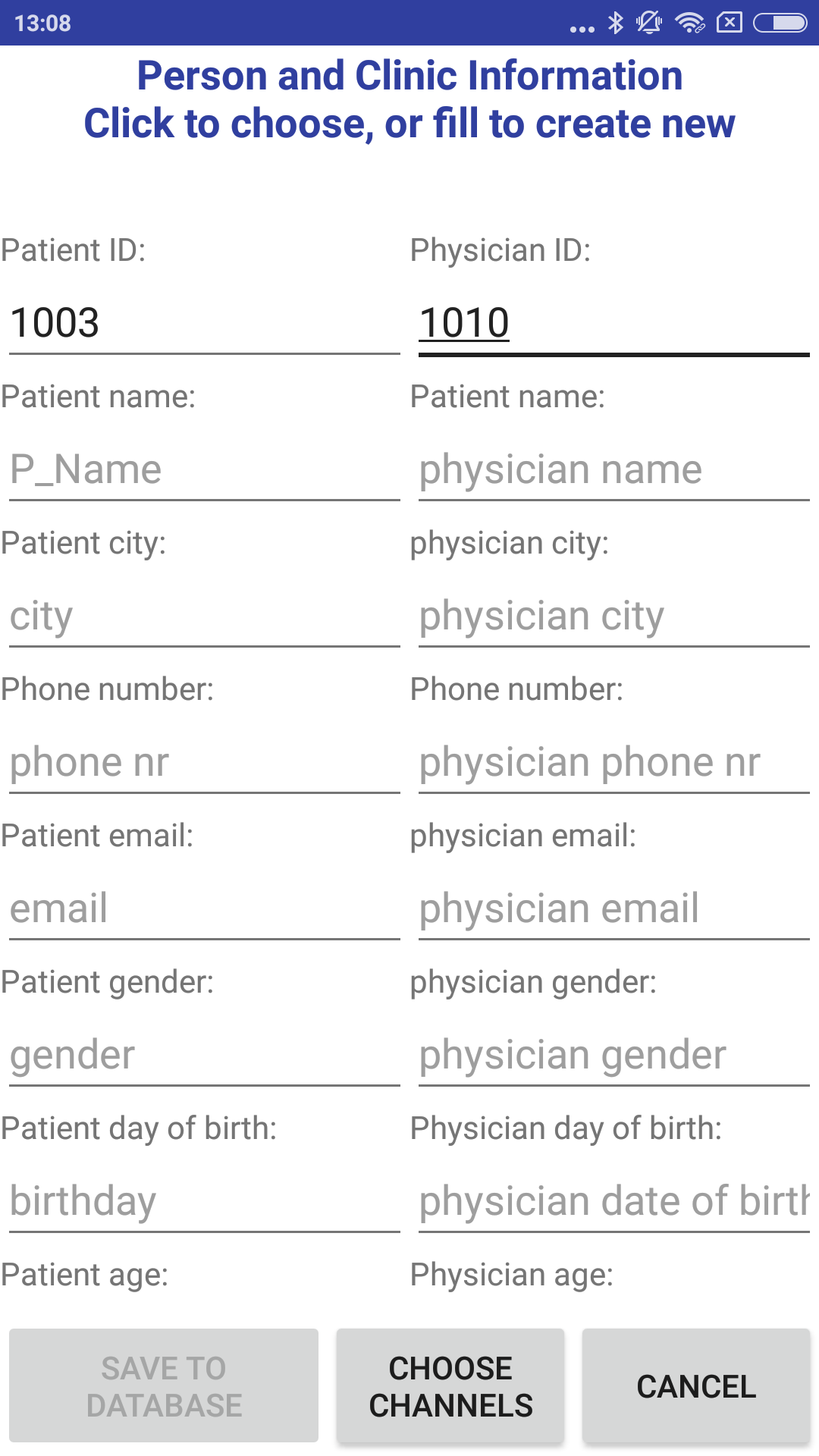
Listing L2 presents how the abstract design and the procedure are translated into coding in Android. With respect to GUI, line xx and xx in L1 post a notification such that the user knows some events have happened. Other logics and codes for managing GUI are not the main focuses of the thesis, and therefore not to be deeply discussed in the writing. These codes can be found in the included project folder.

serverPort = Integer.*parseInt*(txtServerPort.getText().toString());  
final Context CONTEXT = getContext();  
Thread server = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 //Create a server socket object, bind it to server\_port  
 sockServer = new ServerSocket(serverPort);  
 //Multi clients management  
 while (true) {  
 //Accept the client connection, then give it to ServerThread with client socket  
 Socket socClient = sockServer.accept();  
 final String clientIP = socClient.getRemoteSocketAddress().toString();  
 serverUpdateUI.post(new Runnable() {  
 @Override  
 public void run() {  
 Toast.*makeText*(CONTEXT,"Got connect from: "+clientIP,Toast.*LENGTH\_SHORT*).show();  
 }  
 });  
 ClientThread clientConnected = new ClientThread(socClient,CONTEXT,serverUpdateUI, selv);  
 addNewSource(clientConnected);  
 clientConnected.start();  
 }  
 } catch (IOException e) {  
 serverUpdateUI.post(new Runnable() {  
 @Override  
 public void run() {  
 Toast.*makeText*(getContext(),"SERVER IS SHUT DOWN",Toast.*LENGTH\_SHORT*).show();  
 }  
 });  
 //WHEN sockServer close, it will be here  
 sockServer = null;  
 }  
 }  
});  
startServer.setEnabled(false);  
stopListening.setEnabled(true);  
stopAll.setEnabled(true);  
server.start();

L2

For each of sensor source in the connected list, the user can interact with the connected source via four provided functions, that are collecting sample, stopping collecting sample, visualizing, and delete the source from the list. With visualizing, if the source is not fully initialized, or disconnected, the user is not allowed to view the source. Otherwise, the application disconnects all client threads from the graphical graph, then connects the selected source to the graphical graph. It is because the graphical graph allows only one source to be visualized. Any optimizations for the graphical graph are considered future works, since the thesis mainly focus on database and wrapper implementations. If the user clicks on “Delete source”, the application double checks if the user really want to delete the source from the connected list. If it is a case, the selected source is forced to be disconnected, and is removed from the managing list. If it is collecting data for the database, all the data in the buffer must be flushed into the database to ensure that no data losing under collecting process.

The user can perform data collection on the sources with “connected” status. By clicking on one of them, the user is asked for the information for filling patient, physician and clinic information as presented in Figure F2. Since the patients are freely to choose the information they want to save, the fields in the form can be empty. However, the database need some information to manage the patient, physician and clinic, the user must at least provide information for the ID-fields. After that, the user must choose which channels to be stored, then by clicking on “SAVE TO DATABASE” button, the connected source is flagged as storing, and storing process is started. Stopping collecting is quite easy to implement, it can be done by flagging the selected thread as not storing, and ask the thread to flush the buffer into the database.



F2

**ClientThread.java**

After a client thread is created, it immediately listeners for the incoming data at the input stream from its socket. As presented in Chapter 3, CESAR separates the sending data packages by using a newline character. The thread can parse the arrival packages by calling readLine() on the input stream. For each line from the stream, the thread tries to parse the line into a JSON object, then collecting process is initialized if the “type” of the object is “metadata”. Otherwise, the object is sent to graphical graph and the database if “isPlotting” and “isStoring” are flagged. The UML activity diagram from Figure 5.4 in the high level design can be translated into specific implementation as following:

BufferedReader bf = get input stream from client socket;

Loop: as long as the client thread is not disconnected

1. read a line from bf

2. parse the line to JSON object

3. if object type is “metadata”, register new sensor source, go to Loop

4. if object type is “data”, update sample to graphical view and database

then go to Loop

If the user clicks on “STOP ALL” button, or delete a source, the thread is flagged as disconnected, and the collecting process is ended.

The metadata and data package of CESAR acquisition tool are modified in this implementation. The tool has a collecting frequency for each channel, but it does not include the frequencies in the metadata package. A modification is made by including these frequencies into metadata package. In the data package, the timestamp in each package is converted from Unix timestamp into a string before sending. It is obviously not a good solution. There is overhead to convert timestamp for each sample, especially when the sending rate is high. Moreover, a string text ("HH:mm:ss.SSS" is 12 bytes) is more expensive to send compared to Unix timestamp (8 bytes). If the collection is performed at midnight, the text timestamp is confusing the receiver, i.e. from 23:59:59.000 to 00:00:01.000, because the timestamp does not include the date. By sending a Unix timestamp, the problems are solved.

To register a new sensor source is to parse the metadata package into SensorSource and Channel objects. These objects are not pushed into database unless the user performs collecting process. On the other hand, update a sample is performed at least one of the “isPlotting” and “isStoring” flags is flagged. Listing L3 presents how a sample is processed and updated in the system.

void updateSample(JSONObject jsonObj) throws JSONException{  
 if(!isPlotting && !isStoring) return;  
 long timeStamp = jsonObj.getLong("time");  
 // CHANNELS DATA Getting JSON Array node  
 JSONArray channelsData = jsonObj.getJSONArray("data");  
 BitalinoDataSample[] samples = new BitalinoDataSample[channelsData.length()];  
 for(int i = 0; i < channelsData.length(); i++){  
 JSONObject channelData = channelsData.getJSONObject(i);  
 String channel\_nr = channelData.getString("id");  
 float channel\_data = Float.*parseFloat*(channelData.getString("value"));  
 samples[i] = new BitalinoDataSample(timeStamp,channel\_nr,channel\_data);  
 }  
 //SEND TO DATABASE BUFFER OR PLOTTING  
 if(isStoring) manageIsStoring(samples);  
 if(isPlotting) manageIsPlotting(samples);  
}

L3

As presented is Listing L3, if neither plotting nor storing flags are flagged, the sample is thrown. Otherwise, samples are forwarded to graphical view and storing process if they are flagged. The graphical view maintains a sliding buffer to hold samples, and implements the interface BeNotifiedComingSample. The interface has a function addNewSample(BitalinoDataSample[] samples). By calling this function, the graphical view is notified such that it can slide the buffer (if the maximum thread hold is reached), and refresh the GUI. In contrast to plotting that needs to update samples immediately, storing process add new samples into a fixed buffer. The samples are flushed into the database when the buffer is full by submitting to a database update thread. The client thread and the database update thread are synchronized by using producer-consumer algorithm[https://en.wikipedia.org/wiki/Producer–consumer\_problem]. However, a modification is made on the shared buffer for that the application can meet the real-time requirements. That is, the shared buffer is unbounded. The client thread does not need to wait for an empty slot in the buffer, such that it can submit the samples. It just adds the samples into the buffer, notify the database update thread, then continue to get new arrival samples. The database update thread waits for samples if the shared buffer is empty, otherwise it gets samples and initial a SQLite transaction to insert the samples into the database. By using transaction, INSERT statements that are surround with BEGIN and COMMIT are grouped into a single transaction instead of one transaction per INSERT statement. As a result, the performance of the system is increased. Listing L4 illustrates how to use transaction to store samples.

mDatabase.beginTransaction();  
try{  
 for(Sample s : listSample){  
 ContentValues values = new ContentValues();  
 values.put(OSADBHelper.*SAMPLE\_RECORD\_ID*,s.getR\_id());  
 values.put(OSADBHelper.*SAMPLE\_TIMESTAMP*,s.getTimestamp());  
 values.put(OSADBHelper.*SAMPLE\_VALUE*,s.getSample\_data());  
  
 mDatabase.insert(OSADBHelper.*TABLE\_SAMPLE*, null, values);  
 }  
 mDatabase.setTransactionSuccessful();  
}catch(Exception e){  
 e.printStackTrace();  
}finally{  
 mDatabase.endTransaction();  
}

L4

5.3.3 EDF wrapper

The wrapper allows Bio-signals from Physionet databases can be imported into the database system. However, the wrapper accepts only EDF format, therefore the data from Physionet databases need to be exported to EDF format by using the mit2edf function before it can be used by the system. Users can load multiple EDF files simultaneously. Figure F3a presents the GUI in which two EDF files are parallel loading with their status bar which show how far the files have loaded. Users can partially load a EDF file, and can stop the loading process at any moment in time they want. Once a file is chosen, a thread is created to manage the file. At first, the header of the EDF file is parsed to a EDFHeader object. If the EDF file has wrong format, the EDFHeader object is set to null, and therefore the programming stops reading the EDF file. It is to say, nothing is stored to the database if the file does not have the correct format. After parsing the EDF header, objects for Source, Patient, Physician, Clinic, Channel, and Record are created and pushed into the database. To avoid memory overflow, and not to hold the shared SQLite connect for long time, a fixed buffer is used for holding samples. That is, the samples are partially load into memory (buffer). When the buffer is full, the thread starts a SQLite transaction for the collected samples in the buffer. When the SQLite transaction is finished, the thread repeats the reading procedure until the EDF is totally read.

List L5 presents an overview of the EDF file read procedure.

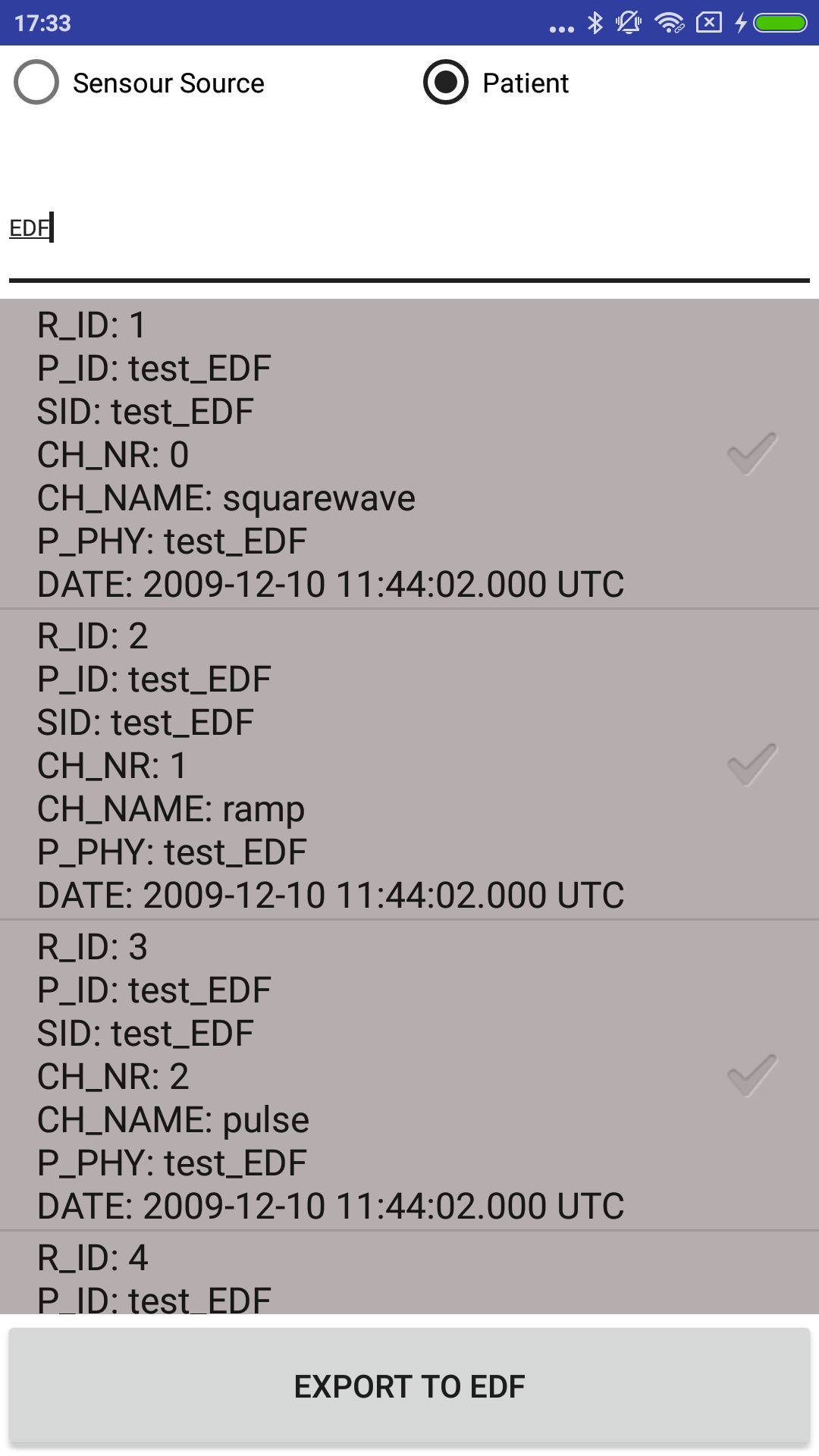
public void run(){  
 final String filePath = file\_source.getFilePath();  
 try {  
 EDFHeader header = null;  
 InputStream is = new BufferedInputStream(new FileInputStream(new File(filePath)));  
 //Parse Header to create new sensor source  
 header = EDFHeaderParser.*parseHeader*(is);  
 is.close();  
 if (header == null) {  
 sendMessageToHandler(*FILE\_IS\_LOADED*, file\_source.getIndex());  
 return;  
 }  
 createAndStoreSensorSource(header);  
 createAndSavePatientPhysicianClinic(header);  
 createAndSaveRecord(header);  
 createAndStoreChannels(header);  
  
 saveRecordFragmentAndSample(header);  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 sendMessageToHandler(*FILE\_IS\_LOADED*, file\_source.getIndex());  
}

L5

The wrappers also allow users to export information in the database to EDF or EDF+ file formats. As Figure3B presents, the users can search all records based on either their ids, or the source ids they used for collecting data. After that, the users can choose which records they want to export to the EDF file. Annotations are depended on records, therefore, all annotations are queried into a buffer before storing procedure for data records begins. As Figure 5.7 in the high level design presents, samples are partially queried into a buffer before flushing to EDF file. While writing the data records, the number of sample in the record must be set to minus one. It is because if something wrong happened under writing, the EDF file is registered as invalid file. When all data records are written to the file, the header of the file need to be updated to valid the file. List L6 presents an overview of the EDF exporting procedure.

public void run(){  
 try{  
 raf = new RandomAccessFile(this.fileName, "rw");  
 buildEDFheader();  
 storeDataRecord();  
 //UPDATE TOTAL RECORD  
 raf.seek(0);  
 EDFWriter.*writeEDFHeaderToFile*(raf,edfHeader);  
 raf.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
}

L6

F3a 3b

5.3.4 Real-time and non-real-time visualization

For representing samples, a graphical technique is used in the implementation, that is a line-based plot. Each sample is presented in the graph with respect of time and its values. Arrival samples are updated to the graph dynamically. To avoid memory problem, the implementation does not keep all samples in the buffer for plotting. A sliding window buffer is used to keep the presented samples. That is, when the buffer is full, the oldest sample is replaced by the new arrival sample. The implementation uses an open source module, which is Android Graph View [http://www.android-graphview.org] for presenting the samples. To solving dynamic plotting, the source also uses a sliding window to avoid memory leaks. That is, before adding a data point to the graph, the buffer is checked if it is full, and the oldest data point is removed in case the max count is reached. List L7[https://github.com/appsthatmatter/GraphView/blob/master/src/main/java/com/jjoe64/graphview/series/LineGraphSeries.java] presents the interface for appending new data points, and L8 presents how it is used in the implementation.

*/\*\*  
 \*  
 \** ***@param*** *dataPoint values the values must be in the correct order!  
 \* x-value has to be ASC. First the lowest x value and at least the highest x value.  
 \** ***@param*** *scrollToEnd true => graphview will scroll to the end (maxX)  
 \** ***@param*** *maxDataPoints if max data count is reached, the oldest data  
 \* value will be lost to avoid memory leaks  
 \** ***@param*** *silent set true to avoid rerender the graph  
 \*/*public void appendData(E dataPoint, boolean scrollToEnd, int maxDataPoints, boolean silent);

L7

v.post(new Runnable() {  
 @Override  
 public void run() {  
 for(BitalinoDataSample sample: samples){  
 LineGraphSeries<DataPoint> tmp = channelLines.get(sample.getChannel\_nr());  
 if(tmp != null && isReady)  
 tmp.appendData(new DataPoint(channels.get(sample.getChannel\_nr()).getLastXRealtime(),sample.getSample\_data()),true,*NR\_ENTRIES\_WINDOW*);  
 }  
 }  
});

L8

In real-time visualization, the graph is passively waiting for other threads update its’ buffer, and notify it when the buffer is updated such that the graph can refresh the GUI. Non-real-time visualization, in contrast, has to query data from the database, and presents the queried data on the graph. A user, therefore, can pause and play queried samples at any moment in time. However, the user cannot do it in real-time visualization. It is because if the feature is supported, some samples do not have a chance to show in the graph. The feature is easily optimized in case the user wants to pause the plotting process in real-time, and the implementation for the visualization is a proof-of-concept, therefore it is not further discussed in detail in this subsection. Figure F4 presents a GUI for non-real-time visualization, in which sources of data can be retrieved by searching the database based on either sensor source id, or patient id. By clicking on a source from the result list and applying it, the user can perform visualization process by interacting with play, pause, write annotation, select channels, save annotation components in the GUI.

