An API for Developing Mobile Ad-hoc Networking Applications Using a Public and Private Screen-based Programming Model
User Manual

A User Manual
Presented to
the Faculty of the College of Computer Studies
De La Salle University

In Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science in Computer Science

by
LARON, Andrew V.
LIM, Sharmaine Amanda S.
SYSON, Oliver Brian C.
XU, Peigen

SEE, Solomon
Faculty Adviser

March 31, 2015

Table of Contents

1.0. Introduction....1-1
1.1. System Requirements....1-2
1.2. Installation....1-2
2.0. Getting Started....2-4
3.0. Device Identity....3-5
4.0. Sessions....4-8
4.1. Session Creation....4-11
4.2. Session Joining....4-12
4.3. Session Locking and Unlocking....4-14
4.4. Handling Events in the SessionActivity....4-14
5.0. Events and Game Logic....5-16
5.1. PlayPlayerActivity (Private Screen)....5-16
5.2. PlayBoardActivity (Public Screen)....5-21
5.3. CardGameEvent....5-23
5.4. CardGameEventHandler....5-23
6.0. System Messages....6-25
6.1. Information and Error Messages....6-25
6.2. Status Messages....6-26

1.0. Introduction

The Public/Private Screen API (PPS API) is for developing networked applications on Android devices, with functions that enable developers to create applications that allow for public and private screen assignment, session handling, and the use of a distributed event model which will allow developers to produce applications without strictly thinking of inter-device messaging as messages being sent, but rather as events being triggered. This API allows developers to code distributed applications as if they were standalone, single-device applications.

You can use the API to

- discover devices;
- create, join, leave, lock and unlock a session;
- trigger and handle events among devices that belong to a common session; and
- set the device as either a public or a private screen.

Assigning a public screen to a device will cause it to display only data that is public knowledge to all devices; in a game of Scrabble, this may be the game board; in a card game, this may be the table or “play area”; in a presentation, this may be the “slides”. Assigning a private screen to a device will cause it to display only data that is known to that private screen device; in a game of Scrabble, this may be one player’s set of tiles; in a card game, this may be one player’s set of cards; in a presentation, this maybe the speaker’s personal notes. A session contains people that wish to work with the same data; in a game of Scrabble or a card game, one session would contain the group of people that want to play together while another session would contain a different group of people who want to play together; in a presentation, this may be for the actual file the users want to edit.

The PPS API uses Samsung’s Chord as the underlying API which abstracts data connection and data transfer. It uses UDP-broadcast (UDP: User Datagram Protocol) for device discovery, and TCP/IP-based protocol (TCP: Transmission Control Protocol; IP: Internet Protocol) to create a local peer-to-peer communication network.

The PPS API provides functionality for creating applications that use public and private screens, and provides functions for session handling. It includes an event-based model which programmers can use to update the application state across the connected devices. It also abstracts data transfer, connection, disconnection and reconnection (to the network only, not including application state) from the developer.

In order for the devices to be connected, they must first connect to the same ad-hoc network, whether by WiFi access point or hotspot. The PPS API allows multiple sessions on a single network. This is accomplished by creating a session and generating a session ID. Whenever a new device joins the network, it has the ability to update the internal list of available sessions. A device will be able to join a session by setting that session’s ID as the session to join. When this is done, the device adds all other devices with the same session ID to an internal list of devices that it can communicate with. Similarly, a device can leave a session by choosing a different session ID or choosing the default session ID, which will remove its device name from other devices and clear its local list of devices to include devices from the new session. A session can be opened or locked to new devices. A locked session simply means other devices are no longer allowed to join.

Whenever the application or API state is changed on a device, whether by a player or by the application, an event is triggered. Whenever an event is triggered, the PPS API converts the event to a message and sends the message to the devices specified by the event’s recipient field. It then applies the event on the local device (the device that triggered the event) if necessary. An event’s recipients may be all screens (a

global event), all public screens (a public event), all private screens, a specific private screen, or only the local screen/device (in which case the event is directly applied). On the recipient device, the message is converted back into an event and applied. The developers only need to create custom events and how the recipient device going to handle it.

1.1. System Requirements

Hardware Requirements:

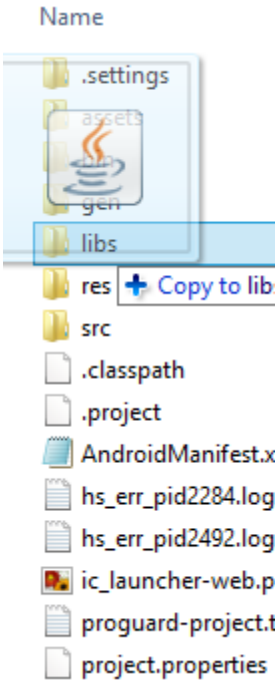
- Samsung smartphone or tablet
- WiFi

Software Requirements:

- Android Ice Cream Sandwich (4.0, API Level 14) or higher
- Samsung Chord API 2.0

1.2. Installation

1. Add the contents of libs.rar to the libs folder of your project.



2. Right click on your project and select Properties, go to Java Build Path and click the Libraries tab. Click add JARs and choose the pps.jar in <your project>/libs. Press OK.

3. Add the following permissions in your Android manifest file (AndroidManifest.xml) under the <manifest> tag. (It can be placed anywhere.)

```
<manifest ...>
    ...
    <uses-permission
        android:name="android.permission.INTERNET"
    />
    <uses-permission
        android:name="android.permission.ACCESS_WIFI_STATE"
    />
    <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE"
    />
    ...
</manifest>
```

4. Inside your Android manifest file (AndroidManifest.xml), find the <uses-sdk> tag and set the minimum SDK to 14 and the target SDK any higher value the developer wants. In this case, the target SDK version is written as 19.

```
<manifest ...>
    ...
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="19"
    />
    ...
</manifest>
```

2.0. Getting Started

The use of the API modules will be shown by demonstrating a sequence of programming flow to develop a “Monkey Monkey Card Game App”. This app allows individual players to place a pair of same-numbered cards (9♥ and 9♣, or Q♦ and Q♥, or A♣ and A♠, etc.) in the play area or board screen. It also allows players to draw cards from other players.

There are three main parts that need to be considered in building the application using the PPS API.

1. Device identity - setting of device screen type (public or private) and device name
2. Session - creation and joining of sessions
3. Events - implementation of application events with their corresponding event handlers

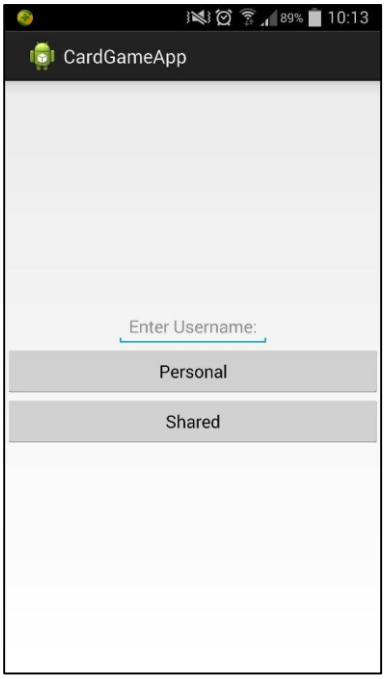
3.0. Device Identity

We first need to build the lobby screen where the device's screen type and name can be set.

1. First, set up the LobbyActivity to connect to its respective XML. In our case, we named our XML "activity_lobby.xml" and it is found under "res/layout/".

```
public class LobbyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lobby);
    }
}
```

2. We make 2 Buttons for the XML layout of this activity, one for setting the device as a private screen device and the other as a public screen device. The buttons call the functions **selectAsPrivate()** and **selectAsPublic()** respectively.




```
public class LobbyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    public void selectAsPrivate(View v) {
        openNextPage(PpsManager.PRIVATE);
    }
```

```
    public void selectAsPublic(View v) {
        openNextPage(PpsManager.PUBLIC);
    }
}
```

3. We declare a variable **ppsManager** and then instantiate it once the user has chosen the screen type. This is done by passing **3 parameters** to the constructor of PpsManager:

    a. first is the context of the application

    b. second is either the boolean PpsManager.PRIVATE or PpsManager.PUBLIC depending on the choice of the user

    c. third is the mode of the next activity. We pass PpsManager.SESSION_MODE because we would be making our session management screen on the next activity before we actually go into building the logic for the card game app.

```
public class LobbyActivity extends Activity {
    public PpsManager ppsManager;

    // Previous functions here

    public void openNextPage(boolean screenType) {
        ppsManager = new PpsManager(this, screenType,
                                    PpsManager.SESSION_MODE);
    }
}
```

4. We set the device's name through the method **setDeviceName()** of the instance of the **SessionManager**, by passing the chosen device name of the user from the TextView in this case.

```
public class LobbyActivity extends Activity {
    public PpsManager ppsManager;

    // Previous functions here

    public void openNextPage(boolean screenType) {
        ppsManager = new PpsManager(this, screenType,
                                    PpsManager.SESSION_MODE);

        TextView mUserNameView;
        mUserNameView = (TextView) findViewById(R.id.txtUserName);
        SessionManager.getInstance().
            setDeviceName(mUserNameView.getText().toString());
    }
}
```

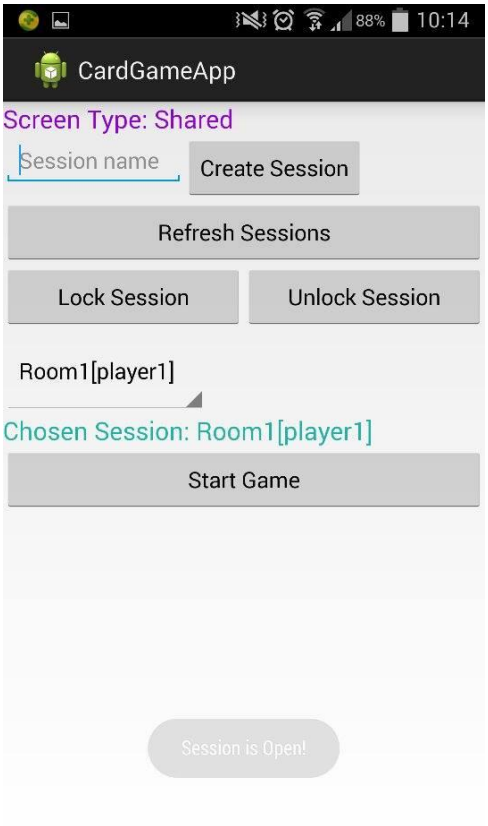
5. We then open the next activity: the SessionActivity!

```
public class LobbyActivity extends Activity {
    public PpsManager ppsManager;

    // Previous functions here

    public void openNextPage(boolean screenType) {
        ppsManager = new PpsManager(this, screenType,
                                    PpsManager.SESSION_MODE);

        TextView mUserNameView;
        mUserNameView = (TextView) findViewById(R.id.txtUserName);
        SessionManager.getInstance().
                       setDeviceName(mUserNameView.getText().toString());

        Intent intent = new Intent(this, SessionActivity.class);
        startActivity(intent);
    }
}
```

4.0. Sessions

The session management will be demonstrated through SessionActivity and SessionEventHandler.

We will now make a SessionActivity which handle the creation, joining, locking and unlocking of sessions. Before that, we have to setup first the EventHandler and the ScreenMode for the session management.



1. In the **onCreate()** method, we can check the screen type chosen by the user, and update our app's interface by updating the TextView (UI).

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_session);

        txtScreenType = (TextView) findViewById(R.id.txt_screen_type);

        if(PpsManager.getInstance().isPrivate())
            txtScreenType.setText("Screen Type: Player");
        else
```

```
            txtScreenType.setText("Screen Type: Game Area");
    }
}
```

2. We also need to set our custom **event handler** to be able to handle updating the view when someone in the network creates a session (SessionEventHandler will be explained after this part).

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_session);

        txtScreenType = (TextView) findViewById(R.id.txt_screen_type);

        if(PpsManager.getInstance().isPrivate())
            txtScreenType.setText("Screen Type: Player");
        else
            txtScreenType.setText("Screen Type: Game Area");

        EventManager.getInstance().
            setEventHandler(new SessionEventHandler());
    }
}
```

3. In the **onPause()** method, place the **stop()** method from the instance of the **PpsManager**; this is important for properly handling of connections and disconnections.

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    @Override
    protected void onPause() {
        super.onPause();
        PpsManager.getInstance().stop();
    }
}
```

4. In the **onResume()** method, pass the **SESSION_MODE** and start the instance of the **PpsManager**.

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;

    // Previously written functions here

    @Override
    protected void onResume() {
        super.onResume();
        PpsManager.getInstance().
            setScreenMode(PpsManager.SESSION_MODE);
        PpsManager.getInstance().start();
    }
}
```

5. Lastly it is important as well to set the **event handler** back to the **SessionEventHandler**. This for later when we create a separate event handler for the actual game. This is done because the API can only handle 1 event handler at a time and you may return to the SessionActivity from the later game activities.

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;

    // Previously written functions here

    @Override
    protected void onResume() {
        super.onResume();
        PpsManager.getInstance().
            setScreenMode(PpsManager.SESSION_MODE);
        PpsManager.getInstance().start();

        EventManager.getInstance().
            setEventHandler(new SessionEventHandler());
    }
}
```

The creation, locking, unlocking and joining of sessions for the game is handled by functions connected to 4 buttons.

4.1. Session Creation

In order to create a session, we just pass the session name we want to the **createSession()** method of the instance of the **SessionManager**. We get the session name through a TextView in this case. Once the session is successfully created, it will return the updated session name where your device's name is appended. (We call this the session ID.) We then update our spinner view of available sessions to show the newly created session by calling the adapter's notify function. Let's initialize the relevant UI while we do this.

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;
    private EditText txtChannel;

    private Spinner spinChannels;
    public static List<String> listChannels;
    public static ArrayAdapter<String> channelsAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...

        txtScreenType = (TextView) findViewById(R.id.txt_screen_type);
        txtChannel = (EditText) findViewById(R.id.txt_channel_name);
        spinChannels = (Spinner) findViewById(R.id.spinner_channel_list);

        if(PpsManager.getInstance().isPrivate())
            txtScreenType.setText("Screen Type: Player");
        else
            txtScreenType.setText("Screen Type: Game Area");

        listChannels = new ArrayList<String>();

        channelsAdapter = new ArrayAdapter<String>(this,
                              android.R.layout.simple_list_item_1,
                              listChannels);

        spinChannels.setAdapter(channelsAdapter);
    }

    // Other previously written functions here

    public void selectCreateSession(View v) {
        String newSession = SessionManager.getInstance().
                            createSession(txtChannel.getText().toString());
        listChannels.add(newSession);
        channelsAdapter.notifyDataSetChanged();
```

```
        txtChannel.setText("");
    }
}
```

4.2. Session Joining

1. To choose the session we want to participate in during the game, we can opt to call SessionManager's **joinSession()** to directly join the session or **setSessionToJoin()** to pre-set the session and the session will be joined after PpsManager.start() is called in the next activity. In the following guideline we will use **setSessionToJoin()** as basis.

```
public class SessionActivity extends Activity {
    private TextView txtScreenType;
    private EditText txtChannel;
    private TextView txtSelected;

    // Other previously written class variables

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_session);

        txtSelected = (TextView)
                      findViewById(R.id.txt_selected_session);

        ...
    }

    // Other previously written functions here

    public void selectJoinSession(View v) {
        String session = spinChannels.getSelectedItem().toString();
        // either
            SessionManager.getInstance().setSessionToJoin(session);
        // or
            SessionManager.getInstance().joinSession(session);
    }
}
```

2. To verify if this was successful, we get the session to join from the SessionManager afterwards. If it returns the **name of the session and the appended device name**, it was successful; however, if it returns the string '**DEFAULT'**, it means it failed to set the session to join for the game. The main reason would be that the session we want to join was locked by the session's owner.

```
public class SessionActivity extends Activity {
    // Previously written class variables

    // Previously written functions here

    public void selectJoinSession(View v) {
        String session = spinChannels.getSelectedItem().toString();
        SessionManager.getInstance().setSessionToJoin(session);

        txtSelectedSession.setText("Session to join: " +
            SessionManager.getInstance().getSessionToJoin());
    }
}
```

4.3. Session Locking and Unlocking

1. To lock a session, we just call the **lockSession()** method from the **SessionManager** and pass the session name with the device name appended. The result will show a log in logcat; a failure message appearing would most probably be because the user is trying to lock a session he did not create.

```
public class SessionActivity extends Activity {
    // Previously written class variables

    // Previously written functions here

    public void selectLock(View v) {
        String session = spinChannels.getSelectedItem().toString();
        SessionManager.getInstance().lockSession(session);
    }
}
```

2. To unlock a session, we just call the **unlockSession()** method from the **SessionManager** and pass the session name with the device name appended. Thie result will show show a log in logcat; a failure message appearing would most probably be because the user is trying to unlock a session he did not create.

```
public class SessionActivity extends Activity {
    // Previously written class variables

    // Previously written functions here

    public void selectUnlock(View v) {
        String session = spinChannels.getSelectedItem().toString();
        SessionManager.getInstance().unlockSession(session);
    }
}
```

4.4. Handling Events in the SessionActivity

1. In the **SessionEventHandler**, we **implement EventHandler** of the PPS API and **override the handleEvent()** method. And then we can catch a particular Event type which is an '**ADD_NEW_SESSION'** event type. This is for the UI to update on the SessionActivity's spinner widget whenever a new session is created.

```
public class SessionEventHandler implements EventHandler {
    @Override
    public void handleEvent(Event event) {
        switch(event.getType()) {
            case Event.T_ADD_NEW_SESSION:
```

```
                SessionActivity.listChannels.add(event.getSession());
                SessionActivity.channelsAdapter.
                    notifyDataSetChanged();
                break;
        }
    }
}
```

2. Once the session to join is setted we want to proceed to the game itself. This is done through **selectProceed()** method, we first check if the user has a session to join and then check the device type of the user before going to the game activity. Before starting the activity, it is important to set the screen mode to APP_MODE as shown below.

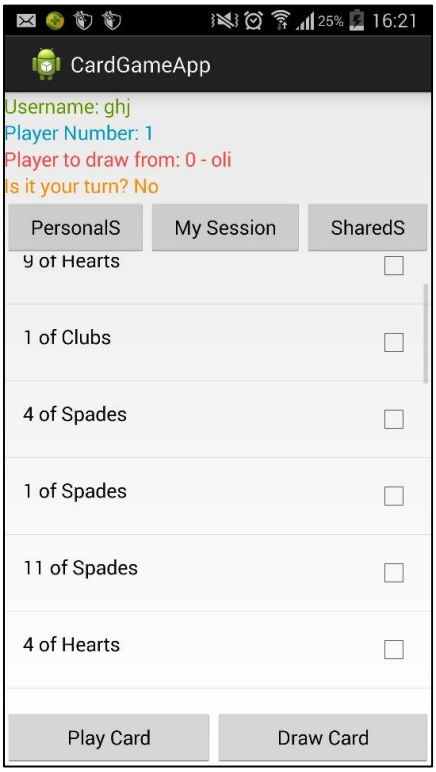
```
public class SessionActivity extends Activity {
    // Previously written class variables

    // Previously written functions here

    public void selectProceed(View v) {
        if(!SessionManager.getInstance().getSessionToJoin().
                isEmpty()) {
            if(PpsManager.getInstance().isPrivate()) {
                Intent intent = new Intent(this,
                    PlayPlayerActivity.class);
                PpsManager.getInstance().
                    setScreenMode(PpsManager.APP_MODE);
                startActivity(intent);
            }
            else if(!PpsManager.getInstance().isPrivate()) {
                Intent intent = new Intent(this,
                    PlayBoardActivity.class);
                PpsManager.getInstance().
                    setScreenMode(PpsManager.APP_MODE);
                startActivity(intent);
            }
        }
    }
}
```

5.0. Events and Game Logic

In this section, we implement the UI and logic for the player/private screen. For this tutorial, we will just implement passing a pair of cards from the player screen to the table/board/public screen.

5.1. PlayPlayerActivity (Private Screen)

We first have to implement a card object containing the attributes number and suit.



1. First, we need to set up for the **PpsManager** to handle connection and disconnection from the network. In the **onPause()** method, call the **stop()** method from **PpsManager**; and in the **onResume()** method, call the **start()** method from the **PpsManager**.

```
public class PlayPlayerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_personal);
    }

    @Override
    protected void onPause() {
        super.onPause();
        PpsManager.getInstance().stop();
```

```
    }

    @Override
    protected void onResume() {
        super.onResume();
        PpsManager.getInstance().start();
    }
}
```

2. We then need to set the event handler on the **onCreate()** method so that the events are handled correctly. It's called **CardGameEventHandler** in this case.

```
public class PlayPlayerActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_personal);

        EventManager.getInstance().
            setEventHandler(new CardGameEventHandler());
    }

    // Previously written functions here
}
```

3. To send a pair of cards to the table/board device, make a button which would call the function for sending the paired card. Let's call it **selectPlay()** and we'll write it in the next step.

4. The function corresponds to selectPlay button, it needs to do the following:

    a. Get the selected card items from the UI. Let's initialize the necessary UI and variables while we're at it.

```
public class PlayPlayerActivity extends Activity {
    private ListView listCards;
    private static HandAdapter handAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_personal);

        EventManager.getInstance().
            setEventHandler(new CardGameEventHandler());
```

```
        listCards = (ListView)
                    findViewById(R.id.listPersonalCards);

        handAdapter = new HandAdapter(this);
        listCards.setAdapter(handAdapter);
    }

    // Previously written functions here

    public void selectPlay(View v) {
        List<Card> cardsToPlay = new ArrayList<Card>();

        // Get the selected cards
        for (int i = 0; i < handAdapter.getCount(); i++) {
            Card item = (Card) handAdapter.getItem(i);
            if(item.isSelected())
                cardsToPlay.add(item);
        }
    }
}
```

b. Check if the user has selected 2 cards and check if the chosen cards have the same number.

```
public class PlayPlayerActivity extends Activity {
    // Previously written class variables here
    // Previously written functions here

    public void selectPlay(View v) {
        List<Card> cardsToPlay = new ArrayList<Card>();

        // Get the selected cards
        for( int i = 0; i < handAdapter.getCount(); i++ ) {
            Card item = (Card) handAdapter.getItem(i);
            if(item.isSelected())
                cardsToPlay.add(item);
        }

        if(cardsToPlay.size() == 2) {
            if (cardsToPlay.get(0).getNumber() ==
                cardsToPlay.get(1).getNumber()) {
                // TODO See next step
            }
        }
    }
}
```

c. If these are all true, we can now send them as an **event** to the shared device, for each card chosen, we make an instance of the Event class and pass the 3 parameters in the constructor:

   i. The first parameter contains the **recipient**, all public screens in this case.

   ii. The second parameter is the **event type** (integer value) so we can handle it on our custom event handler.

   iii. The third paremeter contains the **object** we want to send, the card object in this case.

```
public class PlayPlayerActivity extends Activity {
    // Previously written class variables here
    // Previously written functions here

    public void selectPlay(View v) {
        List<Card> cardsToPlay = new ArrayList<Card>();

        // Get the selected cards
        for( int i = 0; i < handAdapter.getCount(); i++ ) {
            Card item = (Card) handAdapter.getItem(i);
            if(item.isSelected())
                cardsToPlay.add(item);
        }

        if(cardsToPlay.size() == 2) {
            if (cardsToPlay.get(0).getNumber() ==
                cardsToPlay.get(1). getNumber()) {
                for(Card card: cardsToPlay) {
                    Event e = new Event(Event.R_PUBLIC_SCREENS,
                              CardGameEvent.CARD_PLAYED, card);
                    // TODO See next step
                }
            }
        }
    }
}
```

d. We can now send the event by calling **triggerEvent()**, which will trigger the event handlers in the recipient devices to apply the event to those devices and our own device..

```
public class PlayPlayerActivity extends Activity {
    // Previously written class variables here

    // Previously written functions here

    public void selectPlay(View v) {
        List<Card> cardsToPlay = new ArrayList<Card>();

        // Get the selected cards
        for( int i = 0; i < handAdapter.getCount(); i++ ) {
            Card item = (Card) handAdapter.getItem(i);
            if(item.isSelected())
                cardsToPlay.add(item);
        }

        if(cardsToPlay.size() == 2) {
            if (cardsToPlay.get(0).getNumber() ==
                cardsToPlay.get(1). getNumber()) {
                for(Card card: cardsToPlay) {
                    Event e = new Event(Event.R_PUBLIC_SCREENS,
                              CardGameEvent.CARD_PLAYED, card);
                    EventManager.getInstance().triggerEvent(e);
                }
            }
        }
    }
}
```

5. We added a **removeCard()** function which will remove a card on the UI, this can be triggered by the event handler.

```
public class PlayPlayerActivity extends Activity {
    // Previously written class variables here
    // Previously written functions here

    public static void removeCard(Card card) {
        handAdapter.removeCard(card);
    }
}
```

6. We also added a custom check session function wherein we can check on what session we are in. We can check it by calling the **getCurrentSessionName()** method from the instance of the **SessionManager**.

```
public class PlayPlayerActivity extends Activity {
    // Previously written class variables here

    // Previously written functions here

    public void selectCheckSession(View v) {
        Toast.makeText(this,
            SessionManager.getInstance().getCurrentSessionName(),
            Toast.LENGTH_LONG).show();
    }
}
```

5.2. PlayBoardActivity (Public Screen)

Under this section, we would implement the UI and logic for the table/board/public screen.

1. First, we need to set up for the **PpsManager** to handle connection and disconnection from the network. In the **onPause()** method, call the **stop()** method from **PpsManager**; and in the **onResume()** method, call the **start()** method from the **PpsManager**.

```
public class PlayBoardActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_shared);
    }

    @Override
    protected void onPause() {
        super.onPause();
        PpsManager.getInstance().stop();
    }

    @Override
    protected void onResume() {
        super.onResume();
        PpsManager.getInstance().start();
    }
}
```

2. Similar to the player screen, we first need to set the **event handler** in the **onCreate()** method, **CardGameEventHandler** in this case.

```
public class PlayBoardActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_shared);

        EventManager.getInstance().
            setEventHandler(new CardGameEventHandler());
    }

    // Previously written functions here
}
```

3. We then implement the **addCard()** function which will be used by the event handler when a card is sent from other private devices. It will add the received cards to the UI. Let's initialize the UI as well.

```
public class PlayBoardActivity extends Activity {
    private ListView listCards;
    private static HandAdapter handAdapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_play_shared);

        EventManager.getInstance().
            setEventHandler(new CardGameEventHandler());

        listCards = (ListView)
                    findViewById(R.id.listSharedCardsPlayed);

        handAdapter = new HandAdapter(this);
        listCards.setAdapter(handAdapter);
    }

    // Previously written functions here

    public static void addCard(Card c) {
        handAdapter.addCard(c);
    }
}
```

5.3. CardGameEvent

We make a class which **extends Event** and then place all our constant integer types in here. We only have 1 for this particular tutorial, which is for the card played. (Note: Payload means content, so in this case, event payload means event content/message.)

```
public class CardGameEvent extends Event {
      public CardGameEvent(String recipient, int type, String payload) {
            super(recipient, type, payload);
      }

      public static final int CARD_PLAYED = 1;
}
```

5.4. CardGameEventHandler

1. Similar to the session event handler, we implement the **EventHandler** and override the **handleEvent()** method.

```
public class CardGameEventHandler implements EventHandler {
    @Override
    public void handleEvent(final Event event) {
        // TODO See next step
    }
}
```

2. We handle the “card played” event in here. Since we made the event earlier call **triggerEvent()** instead of **sendEvent()**, we need to handle the UI changes for both the player and table/board screens since both of them would be receiving the same event type.

```
public class CardGameEventHandler implements EventHandler {
    @Override
    public void handleEvent(final Event event) {
        switch(event.getType()) {
            case CardGameEvent.CARD_PLAYED:
                // TODO See next step
                break;
        }
    }
}
```

3. We now check if the device where the event was triggered is a table/board device, so we know that what we want to do is add the received cards to the table/board device's list of cards. We simply add it to the UI by calling the **addCard()** method on the **PlayBoardActivity**. The card object can be retrieved from the payload of the event which is the 3rd parameter earlier in the event constructor.

```
public class CardGameEventHandler implements EventHandler {
    @Override
    public void handleEvent(final Event event) {
        switch(event.getType()) {
            case CardGameEvent.CARD_PLAYED:
                if(!PpsManager.getInstance().isPrivate())
                    PlayBoardActivity.addCard(((Card)
                        event.getPayload()));
                // TODO See next step
                break;
        }
    }
}
```

4. If the event is triggered on a player device, we know that we want to remove the cards mentioned in the event by calling **removeCard()** function from the **PlayPlayerActivity**.

```
public class CardGameEventHandler implements EventHandler {
    @Override
    public void handleEvent(final Event event) {
        switch(event.getType()) {
            case CardGameEvent.CARD_PLAYED:
                if(!PpsManager.getInstance().isPrivate())
                    PlayBoardActivity.addCard(((Card)
                        event.getPayload()));
                else
                    PlayPlayerActivity.removeCard(((Card)
                        event.getPayload()));
                break;
        }
    }
}
```

6.0. System Messages

This section lists all system messages – error message, status message, information, and instruction messages.

6.1. Information and Error Messages

1. Message: PPSManager is null, getInstance PPS
   Description: PPSManager is null
   Action: Re-instantiate PPSManager

2. Message: Starting Chord, Successfully connected to the network
   Description: Chord started successfully
   Action: Nothing

3. Message: Starting Chord, Unable to connect to network
   Description: Chord being started again
   Action: Call start() on PpsManager

4. Message: Chord listener started successfully
   Description: Successfully joined the default/custom session
   Action: Nothing

5. Message: Chord listener stopped
   Description: Device left the default/custom session
   Action: Call start() on PpsManager

6. Message: Join default session, error
   Description: Failed to join the default session
   Action: Check if properly connected to mobile AP or wifi

7. Message: SESSION ERROR, Default session is null
   Description: Default channel might not be instantiate properly
   Action: Re-instantiate PpsManager and check network connection

8. Message: Join custom session, error
   Description: Failed to join the custom session
   Action: Check if properly connected to mobile AP or wifi

9. Message: SESSION ERROR, Custom channel is null
   Description: Custom session might not be instantiate properly
   Action: Re-instantiate PpsManager and check network connection

10. Message: <device node name that joined session>, New device connected
    Description: node name of device that joined the session is logged
    Action: Nothing

11. Message: <device node name that left session>, Device disconnected
    Description: node name of device that left the session is logged
    Action: Nothing

12. Message: ChordTransportInterface, sendToAll failed
    Description: Failed to send message to everyone in the session
    Action: Check network connection

13. Message: ChordTransportInterface, send failed
    Description: Failed to send message to a single or group of nodes in the session
    Action: Check network connection

14. Message: Create Session, Invalid session name
    Description: Failed to create a session because of blank parameter
    Action: Check if session name being created is not blank

15. Message: Create Session, The session name already exist
    Description: Failed to create a session because the session name already exist
    Action: Change session name

16. Message: Set Session, The session is open
    Description: Set session to join successful
    Action: Nothing

17. Message: Set Session, The session is locked, you cannot join
    Description: Failed to join a locked session
    Action: Unlock the locked session

18. Message: Set Device Name, Invalid device name
    Description: Device name is blank
    Action: Device name must be alphanumeric and not empty

6.2. Status Messages

1. Message: Screen Type, Device is set as private screen
   Description: The device's screen type is set as private

2. Message: Screen Type, Device is set as public screen
   Description: The device's screen type is set as public

3. Message: PPS event type, <actual event type>
   Description: The received event type
   Action: Handles the event type through the PpsEventHandler

   <PpsEventHandler Logs>
4. Message: PRIVATE USER JOIN
   Description: A new device set as private joins the session. The nodename and alias of the user is added to the SessionManager device map.

5. Message: PUBLIC USER JOIN
   Description: A new device set as public joins the session. The nodename and alias of the user is added to the SessionManager device map.

6. Message: PRIVATE USER LEFT.
   Description: An existing device set as private leaves the session. The nodename and alias of the user is removed from the SessionManager device map.

7. Message: PUBLIC USER LEFT
   Description: An existing device set as public leaves the session. The nodename and alias of the user is removed from the SessionManager device map.

8. Message: NEW SESSION ADDED
   Description: A new session is created and added to the available sessions list in the SessionManager.

9. Message: LOCK SESSION
   Description: An existing session is locked. Other devices are notified of the newly locked session.

10. Message: UNLOCK SESSION
    Description: An existing session is unlocked. Other devices are notified of the newly unlocked session.

11. Message: REQUEST SESSIONS
    Description: A device requests for the sessions other devices has.

12. Message: RESPOND REQUEST SESSIONS
    Description: Devices who receive a request sessions responds by sending the list of sessions to the requester.