Adding Real-time Multiplayer Support to Your Android Game

This guide shows you how to implement a real-time multiplayer game using the Google Play games services in an Android application.

Before you begin

If you haven't already done so, you might find it helpful to review the real-time multiplayer game concepts.

Before you start to code your real-time multiplayer game:

- Make sure to enable real-time multiplayer support for your game in the Google Play Developer Console.
- Download and review the real-time multiplayer game code samples in the Android samples page.
- Familiarize yourself with the recommendations described in Quality Checklist.

Once the player is signed in and the `GoogleApiClient` is connected, you can start using the real-time multiplayer API.

Starting a real-time multiplayer game

Your main screen is the player's primary entry point to start a real-time multiplayer game, invite other players, or accept a pending invitation. We recommend that at minimum you implement these UI components on the main screen of your game:

- **Quick Game button** - Lets the player play against randomly selected opponents (via auto-matching).
- **Invite Players button** - Lets the player invite friends in their Google+ circles to join a game session or specify some number of random opponents for auto-matching.
- **Show Invitations button** - Lets the player see any pending invitations sent by another player. Selecting this option should launch the invitation inbox, as described in Handling invitations.

Note: You should adjust the button labels to best suit your game's context. For example, "Quick Game"

could be renamed "Quick Race".

Quick Game option

When the player selects the **Quick Game** option, your game should create a virtual room object to join players, auto-match the player to randomly selected opponents without displaying the player picker UI, and immediately start the game.

```
private void startQuickGame() {
    // auto-match criteria to invite one random automatch opponent.
    // You can also specify more opponents (up to 3).
    Bundle am = RoomConfig.createAutoMatchCriteria(1, 1, 0);

    // build the room config:
    RoomConfig.Builder roomConfigBuilder = makeBasicRoomConfigBuilder();
    roomConfigBuilder.setAutoMatchCriteria(am);
    RoomConfig roomConfig = roomConfigBuilder.build();

    // create room:
    Games.RealTimeMultiplayer.create(mGoogleApiClient, roomConfig);

    // prevent screen from sleeping during handshake
    getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

    // go to game screen
}
```

If your game has multiple player roles (such as farmer, archer, and wizard) and you want to restrict auto-matched games to one player of each role, add an exclusive bitmask to your room configuration. When auto-matching with this option, players will only be considered for a match when the logical **AND** of their exclusive bit masks is equal to 0. The following example shows how to use the bit mask to perform auto matching with three exclusive roles:

```
private static final long ROLE_FARMER = 0x1; // 001 in binary
private static final long ROLE_ARCHER = 0x2; // 010 in binary
private static final long ROLE_WIZARD = 0x4; // 100 in binary

private void startQuickGame(long role) {
    // auto-match with two random auto-match opponents of different roles
    Bundle am = RoomConfig.createAutoMatchCriteria(2, 2, role);

    // build the room config
    RoomConfig.Builder roomConfigBuilder = makeBasicRoomConfigBuilder();
    roomConfigBuilder.setAutoMatchCriteria(am);
```

```
    // create room, etc.
    // ...
}
```

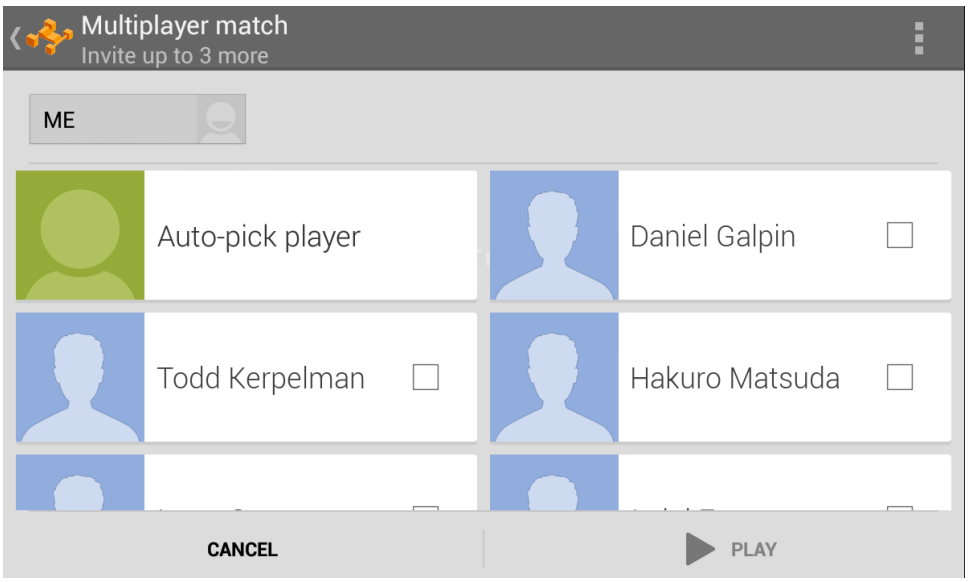
Invite Players option

When the **Invite Players** option is selected, your game should launch a player picker UI that prompts the initiating player to select friends to invite to a real-time game session or select a number of random players for auto-matching. Your game should create a virtual room object using the player's criteria, then start the game session, once players are connected to the room.

To obtain the user's selection, your game can display the built-in player picker UI provided by Google Play games services or a custom player picker UI. To launch the default player picker UI, call the `getSelectOpponentsIntent()` method and use the `Intent` it returns to start a `Activity`.

```
// request code for the "select players" UI
// can be any number as long as it's unique
final static int RC_SELECT_PLAYERS = 10000;

// launch the player selection screen
// minimum: 1 other player; maximum: 3 other players
Intent intent = Games.RealTimeMultiplayer.getSelectOpponentsIntent(mGoogleApiCli
startActivityForResult(intent, RC_SELECT_PLAYERS);
```

An example of the default player picker UI is shown below.



Your game receives the initiating player's criteria on the `onActivityResult()` callback. It

should then create the room and set up listeners to receive notifications of room status changes or incoming messages.

```
@Override
public void onActivityResult(int request, int response, Intent data) {
    if (request == RC_SELECT_PLAYERS) {
        if (response != Activity.RESULT_OK) {
            // user canceled
            return;
        }

        // get the invitee list
        Bundle extras = data.getExtras();
        final ArrayList<String> invitees =
            data.getStringArrayListExtra(Games.EXTRA_PLAYER_IDS);

        // get auto-match criteria
        Bundle autoMatchCriteria = null;
        int minAutoMatchPlayers =
            data.getIntExtra(Multiplayer.EXTRA_MIN_AUTOMATCH_PLAYERS, 0);
        int maxAutoMatchPlayers =
            data.getIntExtra(Multiplayer.EXTRA_MAX_AUTOMATCH_PLAYERS, 0);

        if (minAutoMatchPlayers > 0) {
            autoMatchCriteria = RoomConfig.createAutoMatchCriteria(
                    minAutoMatchPlayers, maxAutoMatchPlayers, 0);
        } else {
            autoMatchCriteria = null;
        }

        // create the room and specify a variant if appropriate
        RoomConfig.Builder roomConfigBuilder = makeBasicRoomConfigBuilder();
        roomConfigBuilder.addPlayersToInvite(invitees);
        if (autoMatchCriteria != null) {
            roomConfigBuilder.setAutoMatchCriteria(autoMatchCriteria);
        }
        RoomConfig roomConfig = roomConfigBuilder.build();
        Games.RealTimeMultiplayer.create(mGoogleApiClient, roomConfig);

        // prevent screen from sleeping during handshake
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    }
}

// create a RoomConfigBuilder that's appropriate for your implementation
private RoomConfig.Builder makeBasicRoomConfigBuilder() {
    return RoomConfig.builder(this)
        .setMessageReceivedListener(this)
```

```
        .setRoomStatusUpdateListener(this);
}
```

Handling room creation errors

To be notified of errors during room creation, your game can use the **`RoomUpdateListener`** callbacks. If a room creation error occurred, your game should display a message to notify players and return to the main screen.

```
@Override
public void onRoomCreated(int statusCode, Room room) {
    if (statusCode != GamesStatusCodes.STATUS_OK) {
        // let screen go to sleep
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // show error message, return to main screen.
    }
}

@Override
public void onJoinedRoom(int statusCode, Room room) {
    if (statusCode != GamesStatusCodes.STATUS_OK) {
        // let screen go to sleep
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // show error message, return to main screen.
    }
}

@Override
public void onRoomConnected(int statusCode, Room room) {
    if (statusCode != GamesStatusCodes.STATUS_OK) {
        // let screen go to sleep
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // show error message, return to main screen.
    }
}
```

Connecting players

To be notified when all players are connected, your game can use the `RoomUpdateListener.onRoomConnected()` callback. Your game can also use the

RoomStatusUpdateListener callbacks to monitor the connection status of the participants. Based on the participant connection status, your game can decide whether to start or cancel the gaming session. For example:

```
// are we already playing?
boolean mPlaying = false;

// at least 2 players required for our game
final static int MIN_PLAYERS = 2;

// returns whether there are enough players to start the game
boolean shouldStartGame(Room room) {
    int connectedPlayers = 0;
    for (Participant p : room.getParticipants()) {
        if (p.isConnectedToRoom()) ++connectedPlayers;
    }
    return connectedPlayers >= MIN_PLAYERS;
}

// Returns whether the room is in a state where the game should be canceled.
boolean shouldCancelGame(Room room) {
    // TODO: Your game-specific cancellation logic here. For example, you might
    // cancel the game if enough people have declined the invitation or left the
    // You can check a participant's status with Participant.getStatus().
    // (Also, your UI should have a Cancel button that cancels the game too)
}

@Override
public void onPeersConnected(Room room, List<String> peers) {
    if (mPlaying) {
        // add new player to an ongoing game
    } else if (shouldStartGame(room)) {
        // start game!
    }
}

@Override
public void onPeersDisconnected(Room room, List<String> peers) {
    if (mPlaying) {
        // do game-specific handling of this -- remove player's avatar
        // from the screen, etc. If not enough players are left for
        // the game to go on, end the game and leave the room.
    } else if (shouldCancelGame(room)) {
        // cancel the game
        Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    }
}
```

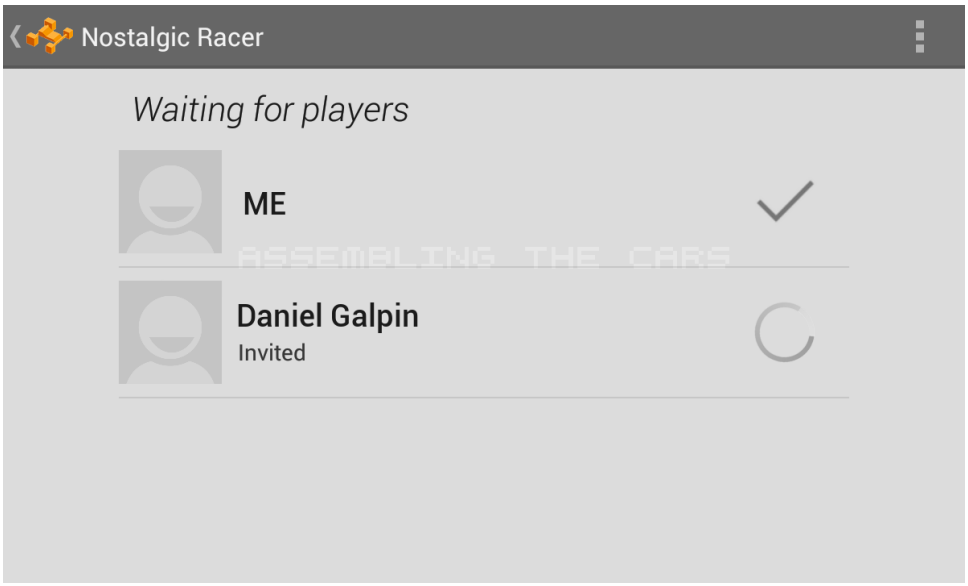
```
@Override
public void onPeerLeft(Room room, List<String> peers) {
    // peer left -- see if game should be canceled
    if (!mPlaying && shouldCancelGame(room)) {
        Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    }
}

@Override
public void onPeerDeclined(Room room, List<String> peers) {
    // peer declined invitation -- see if game should be canceled
    if (!mPlaying && shouldCancelGame(room)) {
        Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);
        getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
    }
}
```

Note: The *player ID* is not the same as the *participant ID*. A player ID is a permanent identifier for a particular user and is consistent from game to game. A participant ID is a temporary identifier that is valid only for a particular room. Invitees from a player's circles have both a player ID and a participant ID. However, to preserve anonymity, players who joined the room via auto-match will only have a participant ID (and not a player ID).

Warning: The `Room` object passed to a callback may not be the same as the object passed to previous callbacks. If your game keeps a reference to the `Room` object, it should update that reference whenever it receives a callback that includes a `Room` argument, such as `onPeersConnected` and `onPeersDisconnected`. Calling methods on an out-of-date `Room` object may return stale information or cause other problems.

Optional: Adding a waiting room UI

We recommend that your game use a "waiting room" UI so that players can see the current status of the room as participants join and get connected. Your game can display the default waiting room UI (shown in the figure below) or a custom UI.



To launch the default waiting room UI, call the **`getWaitingRoomIntent()`** method and use the **`Intent`** it returns to start a **`Activity`**.

Your game can launch the waiting room UI from the **`RoomUpdateListener.onRoomConnected`** and the **`RoomUpdateListener.onJoinedRoom`** callbacks.

```
// arbitrary request code for the waiting room UI.
// This can be any integer that's unique in your Activity.
final static int RC_WAITING_ROOM = 10002;

@Override
public void onJoinedRoom(int statusCode, Room room) {
    if (statusCode != GamesStatusCodes.STATUS_OK) {
       // display error
       return;
    }

    // get waiting room intent
    Intent i = Games.RealTimeMultiplayer.getWaitingRoomIntent(mGoogleApiClient,
    startActivityForResult(i, RC_WAITING_ROOM);
}

@Override
public void onRoomCreated(int statusCode, Room room) {
    if (statusCode != GamesStatusCodes.STATUS_OK) {
        // display error
        return;
    }

    // get waiting room intent
    Intent i = Games.RealTimeMultiplayer.getWaitingRoomIntent(mGoogleApiClient,
    startActivityForResult(i, RC_WAITING_ROOM);
}
```

The second parameter in `getWaitingRoomIntent()` indicates the number of players that must be connected in the room before the **Start playing** option is displayed. In the example, we specify `MAX_VALUE` which indicates that the **Start playing** option is never displayed. Instead, the waiting room UI automatically exits when all players are connected.

Warning: If your game uses the room's total number of participants (returned by `Room.getParticipants().size()`), be aware that the number of participants may grow as new auto-match participants join the room. For example, if you invite two specific players and two auto-matched participants, the number of participants returned by `getParticipants().size()` starts initially at two but may increase to three or four as participants are auto-matched.

When the waiting room UI is dismissed, your game receives the result through your `Activity`'s `onActivityResult()` callback. The reason for the dismissal is indicated in the `responseCode` callback parameter and can be one of the following:

- `Activity.RESULT_OK` - All invited players were successfully connected to the room.
- `Activity.RESULT_CANCELED` - The player pressed the **Back** button or the **Up** button on the Action Bar.
- `GamesActivityResultCodes.RESULT_LEFT_ROOM` - The player selected the **Leave Room** option.

Next, your game should handle the waiting room results in its `onActivityResult()` callback:

```
@Override
public void onActivityResult(int request, int response, Intent intent) {
    if (request == RC_WAITING_ROOM) {
        if (response == Activity.RESULT_OK) {
            // (start game)
        }
        else if (response == Activity.RESULT_CANCELED) {
            // Waiting room was dismissed with the back button. The meaning of t
            // action is up to the game. You may choose to leave the room and ca
            // match, or do something else like minimize the waiting room and
            // continue to connect in the background.

            // in this example, we take the simple approach and just leave the r
            Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);
            getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_O
        }
        else if (response == GamesActivityResultCodes.RESULT_LEFT_ROOM) {
            // player wants to leave the room.
```

```
            Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);
            getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_O
        }
    }
}
```

You can implement a different response depending on whether the user explicitly canceled the game (`GamesActivityResultCodes.RESULT_LEFT_ROOM`) or whether the user quit the waiting room UI (`Activity.RESULT_CANCELED`). For example, if the player dismisses the UI with the **Back** button, you can minimize the app and still continue the handshake process in the background. However, if the player chose the **Leave Room** button from the waiting room UI, you can cancel the handshake.

If you use the waiting room UI, you do not need to implement additional logic to decide when the game should be started or canceled, as seen earlier in `shouldStartGame()` and `shouldCancelGame()`. When you obtain an `Activity.RESULT_OK` result, you can start right away since the required number of participants have been connected. Likewise, when you get an error result from the waiting room UI, you can simply leave the room.

Starting the game before all players are connected

When creating the waiting room, your game can specify the minimum number of players required to start the game session. If the number of connected participants is more than or equal to the specified minimum to start the game, the system enables the **Start Playing** option in the waiting room UI. When the user clicks this option, the system dismisses the waiting room UI and delivers the `Activity.RESULT_OK` result code.

```
final static int MIN_PLAYERS = 2;

// get waiting room intent
Intent i = Games.RealTimeMultiplayer
        .getWaitingRoomIntent(mGoogleApiClient, room, Integer.MIN_PLAYERS);
startActivityForResult(i, RC_WAITING_ROOM);
```

When a player clicks the **Start Playing** option, the waiting room UI is not automatically dismissed for the other players in the game. To dismiss the waiting room for the other players, your game should send a reliable real-time message to the other players to indicate that the game is starting early. When your game receives the message, it should dismiss the waiting room UI:

```
boolean mWaitingRoomFinishedFromCode = false;
```

```
// if "start game" message is received:
mWaitingRoomFinishedFromCode = true;
finishActivity(RC_WAITING_ROOM);
```

The **`mWaitingRoomFinishedFromCode`** flag is necessary because dismissing the waiting room as shown above causes a result code of **`Activity.RESULT_CANCELED`** to be returned. You must differentiate this case from the case where the player has dismissed the waiting room using the back button:

```
@Override
public void onActivityResult(int request, int response, Intent intent) {
    if (request == RC_WAITING_ROOM) {
        // ignore response code if the waiting room was dismissed from code:
        if (mWaitingRoomFinishedFromCode) return;

        // ...(normal implementation, as above)...
    }
}
```

Querying a participant's status

The **`Participant.getStatus()`** method returns the current status of the participant.

- **`STATUS_INVITED`**: The participant has been invited but has not yet acted on the invitation.
- **`STATUS_DECLINED`**: The participant has declined the invitation.
- **`STATUS_JOINED`**: The participant has joined the room.
- **`STATUS_LEFT`**: The participants has left the room.

Your game can also detect if a participant is connected by calling **`Participant.isConnectedToRoom()`**.

Make sure to construct your game logic carefully to take each participant's status and connectedness into account. For example, in a racing game, to determine if all racers have crossed the finish line, your game should only consider the participants who are connected. Your game should not wait for all players in the room cross the finish line, because not all participants might be connected (for example, one or more players might have left the room or declined the invitation).

Here's a code example:

```
Set<String> mFinishedRacers;

boolean haveAllRacersFinished(Room room) {
```

```
    for (Participant p : room.getParticipants()) {
        String pid = p.getParticipantId();
        if (p.isConnectedToRoom() && !mFinishedRacers.contains(pid)) {
            // at least one racer is connected but hasn't finished
            return false;
        }
    }
    // all racers who are connected have finished the race
    return true;
}
```

Detecting when a player is disconnected

Your player might be disconnected from the room due to network connectivity or server issues. To be notified when the player is disconnected from the room, implement the `RoomStatusUpdateListener.onDisconnectedFromRoom` callback.

```
@Override
public void onDisconnectedFromRoom(Room room) {
    // leave the room
    Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);

    // clear the flag that keeps the screen on
    getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

    // show error message and return to main screen
}
```

Handling invitations

Once the player has signed in, your game may be notified of invitations to join a room created by another player. Your game should handle invitations for these scenarios.

At player sign-in

If the signed-in player accepts an invitation from the notification area on the Android status bar, your game should accept the invitation and go directly to the game screen (skipping the main menu).

First, check if an invitation is available after the player signs in successfully. Override the `onConnected()` callback. The `connectionHint` parameter of the `onConnected()` callback indicates if there is an invitation to accept:

```
private RoomConfig.Builder makeBasicRoomConfigBuilder() {
    return RoomConfig.builder(this)
            .setMessageReceivedListener(this)
            .setRoomStatusUpdateListener(this)
}

@Override
public void onConnected(Bundle connectionHint) {
    // ...

    if (connectionHint != null) {
        Invitation inv =
            connectionHint.getParcelable(Multiplayer.EXTRA_INVITATION);

        if (inv != null) {
            // accept invitation
            RoomConfig.Builder roomConfigBuilder = makeBasicRoomConfigBuilder();
            roomConfigBuilder.setInvitationIdToAccept(inv.getInvitationId());
            Games.RealTimeMultiplayer.join(mGoogleApiClient, roomConfigBuilder.b

            // prevent screen from sleeping during handshake
            getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON)

            // go to game screen
        }
    }
}
```

If the device goes to sleep during handshake or gameplay, the player will be disconnected from the room. To prevent the device from sleeping during multiplayer handshake or gameplay, we recommend that you enable the **`FLAG_KEEP_SCREEN_ON`** flag in your **`Activity`**'s **`onCreate()`** method:

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

Don't forget to clear this flag at the end of gameplay or when the game is canceled.

During gameplay

To be notified of incoming invitations, your game can register an **`OnInvitationReceivedListener`**. Incoming invitations will not generate a status bar notification. Instead, the listener is notified, and your game can then display an in-game popup dialog or notification to inform the user. If the user accepts, your game should process the invitation and launch the game screen.

First, register an `OnInvitationReceivedListener` after the player signs in successfully:

```
@Override
protected void onConnected(Bundle connectionHint) {
    // ...
    Games.Invitations.registerInvitationListener(mGoogleApiClient, mListener);
    // ...
}
```

Then, handle the `onInvitationReceived()` callback:

```
@Override
public void onInvitationReceived(Invitation invitation) {
    // show in-game popup to let user know of pending invitation

    // store invitation for use when player accepts this invitation
    mIncomingInvitationId = invitation.getInvitationId();
}
```

When the user clicks that popup to accept the incoming invitation, join the room:

```
RoomConfig.Builder roomConfigBuilder = makeBasicRoomConfigBuilder();
roomConfigBuilder.setInvitationIdToAccept(mIncomingInvitationId);
Games.RealTimeMultiplayer.join(mGoogleApiClient, roomConfigBuilder.build());

// prevent screen from sleeping during handshake
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

// now, go to game screen
```

From the Invitation Inbox

The **Invitation Inbox** is an optional UI component that your game can display by using the `Intent` from `getInvitationInboxIntent()`. The Inbox displays all the available invitations that a player received. If the player selects a pending invitation from the Inbox, your game should accept the invitation and launch the game screen.

You can add a button to launch the Invitation Inbox from the main screen of your game. To launch the inbox:

```
// request code (can be any number, as long as it's unique)
final static int RC_INVITATION_INBOX = 10001;

// launch the intent to show the invitation inbox screen
```

```
Intent intent = Games.Invitations.getInvitationInboxIntent();
mActivity.startActivityForResult(intent, RC_INVITATION_INBOX);
```

When the player selects an invitation from the Inbox, your game is notified via **`onActivityResult()`**. Your game can then process the invitation:

```
@Override
public void onActivityResult(int request, int response, Intent data) {
    if (request == RC_INVITATION_INBOX) {
        if (response != Activity.RESULT_OK) {
            // canceled
            return;
        }

        // get the selected invitation
        Bundle extras = data.getExtras();
        Invitation invitation =
            extras.getParcelable(Multiplayer.EXTRA_INVITATION);

        // accept it!
        RoomConfig roomConfig = makeBasicRoomConfigBuilder()
                .setInvitationIdToAccept(invitation.getInvitationId())
                .build();
        Games.RealTimeMultiplayer.join(mGoogleApiClient, roomConfig);

        // prevent screen from sleeping during handshake
        getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

        // go to game screen
    }
}
```

Exchanging game data between clients

Please review Sending game data to familiarize yourself with the concepts behind data messaging using the real-time multiplayer API.

Preparing the room for data messaging

Before using Google Play games services to send or receive messages in your game, you must first configure the room.

- To use reliable or unreliable messaging, configure the room by calling **`setMessageReceivedListener()`** and pass in the listener as the argument.

Sending messages

Note: Your game can only send messages to connected participants. To know when peer clients are connected, your game can monitor the `RoomStatusUpdateListener.onPeerConnected` callback.

To send a message, your game can use either the **`sendReliableMessage()`** or **`sendUnreliableMessage()`** method, depending on the kind of message to send.

For example, to send a reliable message to all other participants:

```
byte[] message = .......;
for (Participant p : mParticipants) {
    if (!p.getParticipantId().equals(mMyId)) {
        Games.RealTimeMultiplayer.sendReliableMessage(mGoogleApiClient, null, me
                mRoomId, p.getParticipantId());
    }
}
```

Alternatively, your game can use the **`sendUnreliableMessageToOthers()`** method to send a broadcast message.

Receiving messages

When your game receives a message, it is notified by the **`RealTimeMessageReceivedListener.onRealTimeMessageReceived`** callback:

```
@Override
public void onRealTimeMessageReceived(RealTimeMessage rtm) {
    // get real-time message
    byte[] b = rtm.getMessageData();

    // process message
}
```

This method will be called whether it's a reliable or unreliable message.

Note: Be sure to register this listener when setting up your room configuration, in the `makeBasicRoomConfig()` method or its equivalent.

Leaving the room

Your game should leave the active room when one of these scenarios occurs:

- Gameplay is over - If you don't leave the room, Google Play games services will continue to send notifications for that room to your `GoogleApiClient` and room listeners.
- `onStop()` is called - If the `onStop()` method is called, this might indicate that your `Activity` is being destroyed. You should leave the room and call `disconnect()`.
- The user cancels the game in the waiting room UI - Your game should also leave the room if the response code returned in the `onActivityResult()` callback is `GamesActivityResultCodes.RESULT_LEFT_ROOM`.

To leave the room, call `leave()`:

```
// leave room
Games.RealTimeMultiplayer.leave(mGoogleApiClient, null, mRoomId);

// remove the flag that keeps the screen on
getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

After you leave the room, wait until you receive a call to the `RoomUpdateListener.onLeftRoom()` callback before attempting to start or join another room:

```
@Override
public void onLeftRoom(int code, String roomId) {
    // left room. Ready to start or join another room.
}
```


Last updated January 21, 2016.