Adding Peer Group Manager

to Simple Service and Simple Client

Contents

[PURPOSE 1](#_Toc340670665)

[PREREQUESITES 1](#_Toc340670666)

[SIMPLE SERVICE 1](#_Toc340670667)

[Bus Handler Members 1](#_Toc340670668)

[Connecting 2](#_Toc340670669)

[Creating a Session/Group 4](#_Toc340670670)

[Disconnect 6](#_Toc340670671)

[SIMPLE CLIENT 7](#_Toc340670672)

[Bus Handler Members 7](#_Toc340670673)

[Connecting 8](#_Toc340670674)

[Joining 10](#_Toc340670675)

[Disconnecting and Pinging 12](#_Toc340670676)

[RESULTS 12](#_Toc340670677)

PURPOSETo demonstrate the implementation of existing samples Simple Service and Simple Client using the Peer Group Manager.

PREREQUESITESThis guide was created using the samples included in AllJoyn SDK 3.0.2 – Android.

SIMPLE SERVICESimple Service is a basic AllJoyn example for Android that acts like a typical service, waiting for a connection from the client. Most of the AllJoyn related code is encapsulated in the Bus Handler, a class design to interact with the AllJoyn bus. All of the changes necessary to convert Simple Service from base AllJoyn to the Peer Group Manager are made in the Bus Handler.

Bus Handler MembersThe original Bus Handler contains a String labeled *SERVICE\_NAME*. The *SERVICE\_NAME* is the full well-known name that the Simple Client uses to connect to the Simple Service. The Peer Group Manager breaks down the well-known name into two parts – *GROUP\_PREFIX* and *GROUP\_NAME*. *GROUP\_PREFIX* acts as the base of the well-known name, and the Peer Group Manager can only find other groups with the same prefix.

The original Bus Handler also contains a *CONTACT\_PORT*. Without the Peer Group Manager, the developer must choose a port on which clients can connect and ensure that the port is well-known. In contrast, the Peer Group Manager chooses a port for the developer, removing the need for a *CONTACT\_PORT* member.

Original:

**class** BusHandler **extends** Handler {

/\*

\* Name used as the well-known name and the advertised name.

\* This name must be a unique name both to the bus and to the

\* network as a whole. The name uses reverse URL style of naming.

\*/

**private** **static** **final** String *SERVICE\_NAME* = "org.alljoyn.bus.samples.simple";

**private** **static** **final** **short** *CONTACT\_PORT*=42;

**private** BusAttachment mBus

With Peer Group Manager:

**class** BusHandler **extends** Handler {

/\*

\* Group prefix is handed to the PeerGroupManager's constructor

\* and used to advertise any created groups to peers. A reverse

\* URL naming style is used.

\*

\* Group name is an identifier for your specific grou p, used in

\* Peer Group Manager calls such as createGroup and joinGroup

\*/

**private** **static** **final** String *GROUP\_PREFIX* = "org.alljoyn.bus.samples.simple";

**private** **static** **final** String *GROUP\_NAME* = "service";

**private** PeerGroupManager mPeerGroupManager;

ConnectingMost of the changes to Simple Service appear in the *CONNECT* case within the *handleMessage* method.

Originally, a Bus Attachment must be created and then used to register a Bus Listener, register a Bus Object, and connect to the AllJoyn bus. With the Peer Group Manager, this all occurs in the constructor and is abstracted from the developer.

Original:

**case** *CONNECT*: {

org.alljoyn.bus.alljoyn.DaemonInit.*PrepareDaemon*(getApplicationContext());

/\*

\* All communication through AllJoyn begins with a BusAttachment.

\*

\* A BusAttachment needs a name. The actual name is unimportant

\* except for internal security. As a default we use the class

\* name as the name.

\*

\* By default AllJoyn does not allow communication between

\* devices (i.e. bus to bus communication). The second argument

\* must be set to Receive to allow communication between devices.

\*/

mBus = **new** BusAttachment(getPackageName(), BusAttachment.RemoteMessage.*Receive*);

/\*

\* Create a bus listener class

\*/

mBus.registerBusListener(**new** BusListener());

/\*

\* To make a service available to other AllJoyn peers, first register

\* a BusObject with the BusAttachment at a specific path.

\*

\* Our service is the SimpleService BusObject at the "/SimpleService" path.

\*/

Status status = mBus.registerBusObject(mSimpleService, "/SimpleService");

logStatus("BusAttachment.registerBusObject()", status);

**if** (status != Status.*OK*) {

finish();

**return**;

}

/\*

\* The next step in making a service available to other AllJoyn peers

\* is to connect the BusAttachment to the bus with a well-known name.

\*/

/\*

\* connect the BusAttachement to the bus

\*/

status = mBus.connect();

logStatus("BusAttachment.connect()", status);

**if** (status != Status.*OK*) {

finish();

**return**;

}

With Peer Group Manager:

**case** *CONNECT*: {

org.alljoyn.bus.alljoyn.DaemonInit.*PrepareDaemon*(getApplicationContext());

/\*

\* PeerGroupManager handles all communication with AllJoyn.

\*

\* PeerGroupManager takes a group prefix, defined previously.

\*

\* Also, a PeerGroupListener is required to receive informative callback methods.

\* Only desired callback methods must be overridden. However, no callback

\* information is needed in this example, so an instance of a PeerGroupListener

\* with no overridden methods is passed into the constructor.

\*

\* Finally, an array of BusObjects that you want the PeerGroupManager

\* to register with AllJoyn is passed in. This simple Service has only

\* one bus object.

\*/

/\*

\* Create a BusObject array

\*/

ArrayList<BusObjectData> busObjects = **new** ArrayList<BusObjectData>();

busObjects.add(**new** BusObjectData(mSimpleService, "/SimpleService"));

/\*

\* Create the PeerGroupManager

\*/

mPeerGroupManager = **new** PeerGroupManager(*GROUP\_PREFIX*, **new** PeerGroupListener(), busObjects);

## Creating a Session/Group

The next step in the *CONNECT* case is creating a session (without the Peer Group Manager) or group (with the Peer Group Manager). Originally, a contact port must be defined, as well as several session options, which are used to bind the session port. A Session Port Listener must also be defined that outlines the session’s acceptance policy in the acceptSessionJoiner callback method. Finally, the desired session name must be requested and, if granted, advertised.

With the Peer Group manager, all of this functionality is included in the createGroup method. The Peer Group Manager automatically assigns the group an available port and uses predefined options (multipoint, any proximity, any transport). The Peer Group Manager also sets a default acceptance policy allowing anyone who connects on the correct port to join. The group can still be explicitly locked by the developer, which causes any joiners to be rejected. All that is needed to create a group with Peer Group Manager is a group name.

Original:

/\*

\* Create a new session listening on the contact port of the chat service.

\*/

Mutable.ShortValue contactPort = **new** Mutable.ShortValue(*CONTACT\_PORT*);

SessionOpts sessionOpts = **new** SessionOpts();

sessionOpts.traffic = SessionOpts.*TRAFFIC\_MESSAGES*;

sessionOpts.isMultipoint = **false**;

sessionOpts.proximity = SessionOpts.*PROXIMITY\_ANY*;

sessionOpts.transports = SessionOpts.*TRANSPORT\_ANY*;

status = mBus.bindSessionPort(contactPort, sessionOpts, **new** SessionPortListener() {

@Override

**public** **boolean** acceptSessionJoiner(**short** sessionPort, String joiner, SessionOpts sessionOpts) {

**if** (sessionPort == *CONTACT\_PORT*) {

**return** **true**;

} **else** {

**return** **false**;

}

}

});

logStatus(String.*format*("BusAttachment.bindSessionPort(%d, %s)",

contactPort.value, sessionOpts.toString()), status);

**if** (status != Status.*OK*) {

finish();

**return**;

}

/\*

\* request a well-known name from the bus

\*/

**int** flag = BusAttachment.*ALLJOYN\_REQUESTNAME\_FLAG\_REPLACE\_EXISTING* | BusAttachment.*ALLJOYN\_REQUESTNAME\_FLAG\_DO\_NOT\_QUEUE*;

status = mBus.requestName(*SERVICE\_NAME*, flag);

logStatus(String.*format*("BusAttachment.requestName(%s, 0x%08x)", *SERVICE\_NAME*, flag), status);

**if** (status == Status.*OK*) {

/\*

\* If we successfully obtain a well-known name from the bus

\* advertise the same well-known name

\*/

status = mBus.advertiseName(*SERVICE\_NAME*, SessionOpts.*TRANSPORT\_ANY*);

logStatus(String.*format*("BusAttachement.advertiseName(%s)", *SERVICE\_NAME*), status);

**if** (status != Status.*OK*) {

/\*

\* If we are unable to advertise the name, release

\* the well-known name from the local bus.

\*/

status = mBus.releaseName(*SERVICE\_NAME*);

logStatus(String.*format*("BusAttachment.releaseName(%s)", *SERVICE\_NAME*), status);

finish();

**return**;

}

}

**break**;

}

With Peer Group Manager:

/\*

\* To allow peers to connect, a group must be created.

\* We use the group name previously defined.

\*/

Status status = mPeerGroupManager.createGroup(*GROUP\_NAME*);

logStatus(String.*format*("PeerGroupManager.createGroup%s)", *GROUP\_NAME*), status);

**if** (status != Status.*OK*) {

finish();

**return**;

}

**break**;

}

## Disconnect

Without the Peer Group Manager, disconnecting includes unregistering Bus Objects, and disconnecting from the AllJoyn bus. However, the Peer Group Manager includes a cleanup method that handles disconnection for you.

Original:

**case** *DISCONNECT*: {

/\*

\* It is important to unregister the BusObject before disconnecting from the bus.

\* Failing to do so could result in a resource leak.

\*/

mBus.unregisterBusObject(mSimpleService);

mBus.disconnect();

mBusHandler.getLooper().quit();

**break**;

}

With Peer Group Manager:

**case** *DISCONNECT*: {

/\*

\* PeerGroupManager has a cleanup method which unregisters bus objects and

\* disconnects from AllJoyn. The PeerGroupManager should no longer be used

\* after calling cleanup.

\*/

mPeerGroupManager.cleanup();

mBusHandler.getLooper().quit();

**break**;

}

SIMPLE CLIENTSimple Client is a basic AllJoyn example for Android that acts like a typical client, connecting to a known service. Again, most of the AllJoyn related code is encapsulated in the Bus Handler, a class design to interact with the AllJoyn bus. All of the changes necessary to convert Simple Client from base AllJoyn to the Peer Group Manager are made in the Bus Handler.

Bus Handler MembersLike in Simple Service,adding the Peer Group Manager changes *SERVICE\_NAME* to *GROUP\_PREFIX* while also removing the need to define a *CONTACT\_PORT*. Simple Client has additional members for keeping state (an integer and three Booleans) that are similarly unneeded in the Peer Group Manager implementation.

Also, the original Simple Client has a Proxy Bus Object member. With Peer Group Manager, this member is unnecessary. Peer Group Manager allows the developer to deal directly with the interface, rather than having to get a Proxy Bus Object and then call getInterface on it.

Original:

**class** BusHandler **extends** Handler {

/\*

\* Name used as the well-known name and the advertised name of the

\* service this client is interested in. This name must be a unique

\* name both to the bus and to the network as a whole.

\*

\* The name uses reverse URL style of naming, and matches the name

\* used by the service.

\*/

**private** **static** **final** String *SERVICE\_NAME* = "org.alljoyn.bus.samples.simple";

**private** **static** **final** **short** *CONTACT\_PORT*=42;

**private** BusAttachment mBus;

**private** ProxyBusObject mProxyObj;

**private** SimpleInterface mSimpleInterface;

**private** **int** mSessionId;

**private** **boolean** mIsInASession;

**private** **boolean** mIsConnected;

**private** **boolean** mIsStoppingDiscovery;

With Peer Group Manager:

**class** BusHandler **extends** Handler {

/\*

\* Group prefix is handed to the Peer Group Manager's constructor

\* and used in discovery to find matching groups. A reverse URL

\* naming style is used.

\*/

**private** **static** **final** String *GROUP\_PREFIX* = "org.alljoyn.bus.samples.simple";

**private** PeerGroupManager mPeerGroupManager;

**private** SimpleInterface mSimpleInterface;

ConnectingOriginally, a Bus Attachment must be created and then used to register a Bus Listener, register a Bus Object, connect to the AllJoyn bus, and call findAdvertiseName on the well-known name. With the Peer Group Manager, this all occurs in the constructor and is abstracted from the developer.

Unlike Simple Service, Simple Client has implemented the foundAdvertisedName and sessionLost/groupLost callback methods. Originally, Simple Service overrides the foundAdvertisedName callback of the Bus Listener in the *CONNECT* case and later overrides the sessionLost callback of the Session Listener in the *JOIN* case. However, Peer Group Manager collapses both of these listeners into a Peer Group Listener. This listener is taken as a constructor argument, so both callback methods are overridden in the *CONNECT* case.

Original:

**case** *CONNECT*: {

org.alljoyn.bus.alljoyn.DaemonInit.*PrepareDaemon*(getApplicationContext());

/\*

\* All communication through AllJoyn begins with a BusAttachment.

\*

\* A BusAttachment needs a name. The actual name is unimportant

\* except for internal security. As a default we use the class

\* name as the name.

\*

\* By default AllJoyn does not allow communication between devices

\* (i.e. bus to bus communication). The second argument must be set

\* to Receive to allow communication between devices.

\*/

mBus = **new** BusAttachment(getPackageName(), BusAttachment.RemoteMessage.*Receive*);

/\*

\* Create a bus listener class

\*/

mBus.registerBusListener(**new** BusListener() {

@Override

**public** **void** foundAdvertisedName(String name, **short** transport, String namePrefix) {

logInfo(String.*format*("MyBusListener.foundAdvertisedName(%s, 0x%04x, %s)", name, transport, namePrefix));

/\*

\* This client will only join the first service that it sees advertising

\* the indicated well-known name. If the program is already a member of

\* a session (i.e. connected to a service) we will not attempt to join

\* another session.

\* It is possible to join multiple session however joining multiple

\* sessions is not shown in this sample.

\*/

**if**(!mIsConnected) {

Message msg = obtainMessage(*JOIN\_SESSION*, name);

sendMessage(msg);

}

}

});

/\*

\* To communicate with AllJoyn objects, we must connect the BusAttachment

\* to the bus.

\*/

Status status = mBus.connect();

logStatus("BusAttachment.connect()", status);

**if** (Status.*OK* != status) {

finish();

**return**;

}

/\*

\* Now find an instance of the AllJoyn object we want to call. We start

\* by looking for a name, then connecting to the device that is advertising

\* that name.

\*

\* In this case, we are looking for the well-known SERVICE\_NAME.

\*/

status = mBus.findAdvertisedName(*SERVICE\_NAME*);

logStatus(String.*format*("BusAttachement.findAdvertisedName(%s)", *SERVICE\_NAME*), status);

**if** (Status.*OK* != status) {

finish();

**return**;

}

**break**;

}

With Peer Group Manager:

**case** *CONNECT*: {

org.alljoyn.bus.alljoyn.DaemonInit.*PrepareDaemon*(getApplicationContext());

/\*

\* PeerGroupManager handles all communication with AllJoyn.

\*

\* PeerGroupManager takes a group prefix, defined previously.

\*

\* Also, a PeerGroupListener is required to receive informative

\* callback methods. Only desired callback methods must be overridden. In this

\* client, we use the foundAdvertisedName and groupLost callback methods.

\*

\* Finally, an array of BusObjects that you want the PeerGroupManager

\* to register with AllJoyn is passed in. This simple client is a pure

\* client with no bus objects, so a null reference is passed in.

\*/

PeerGroupListener mPeerGroupListener = **new** PeerGroupListener(){

@Override

**public** **void** foundAdvertisedName(String groupName, **short** transport) {

logInfo(String.*format*("MyPeerGroupListener.foundAdvertisedName(%s, 0x%04x)", groupName, transport));

/\*

\* This client will only join the first service that

\* it sees advertising a group with the indicated

\* group prefix. If the program has already joined

\* a group (i.e. connected to a service) we will not

\* attempt to join another group.

\*

\* It is possible to join multiple groups, however

\* joining multiple groups is not shown in this sample.

\*/ \*/

Message msg = obtainMessage(*JOIN\_SESSION*, groupName);

sendMessage(msg);

};

@Override

**public** **void** groupLost(String groupName) {

/\*

\* Upon losing a group advertisement, a message is sent to start

\* the process dialog in android.

\*/

logInfo(String.*format*("MyPeerGroupListener.groupLost(%s)", groupName));

mHandler.sendEmptyMessage(*MESSAGE\_START\_PROGRESS\_DIALOG*);

};

};

mPeerGroupManager = **new** PeerGroupManager(*GROUP\_PREFIX*, mPeerGroupListener, **null**);

**break**;

}

Joining  
Similar to creating a session in the original Simple Service, joining a session in the original Simple Client requires a contact port and several session options. These values, along with a mutable integer named sessionId, are arguments in the joinSession method. The mutable sessionId acts as a return value that allows Simple Client to store the sessionId of the joined session.

When using Peer Group Manager, all of these parameters disappear. The contact port, session options, and session Id are all hidden from the developer. Instead, the developer deals simply with the group name (defined by the Simple Service). To get the remote interface, the Peer Group Manager’s getRemoteObject interface method can be used directly rather than the original getProxyBusObject method followed by a getInterface call.

Original:

**case** (*JOIN\_SESSION*): {

/\*

\* If discovery is currently being stopped don't join to any other sessions.

\*/

**if** (mIsStoppingDiscovery) {

**break**;

}

/\*

\* In order to join the session, we need to provide the well-known

\* contact port. This is pre-arranged between both sides as part

\* of the definition of the chat service. As a result of joining

\* the session, we get a session identifier which we must use to

\* identify the created session communication channel whenever we

\* talk to the remote side.

\*/

**short** contactPort = *CONTACT\_PORT*;

SessionOpts sessionOpts = **new** SessionOpts();

Mutable.IntegerValue sessionId = **new** Mutable.IntegerValue();

Status status = mBus.joinSession((String) msg.obj, contactPort, sessionId, sessionOpts, **new** SessionListener() {

@Override

**public** **void** sessionLost(**int** sessionId) {

mIsConnected = **false**;

logInfo(String.*format*("MyBusListener.sessionLost(%d)", sessionId));

mHandler.sendEmptyMessage(*MESSAGE\_START\_PROGRESS\_DIALOG*);

}

});

logStatus("BusAttachment.joinSession() - sessionId: " + sessionId.value, status);

**if** (status == Status.*OK*) {

/\*

\* To communicate with an AllJoyn object, we create a ProxyBusObject.

\* A ProxyBusObject is composed of a name, path, sessionID and interfaces.

\*

\* This ProxyBusObject is located at the well-known SERVICE\_NAME,

\* under path "/SimpleService", uses sessionID of CONTACT\_PORT,

\* and implements the SimpleInterface.

\*/

mProxyObj = mBus.getProxyBusObject(*SERVICE\_NAME*,

"/SimpleService",

sessionId.value,

**new** Class<?>[] { SimpleInterface.**class** });

/\*

\* We make calls to the methods of the AllJoyn object

\* through one of its interfaces.

\*/

mSimpleInterface = mProxyObj.getInterface(SimpleInterface.**class**);

mSessionId = sessionId.value;

mIsConnected = **true**;

mHandler.sendEmptyMessage(*MESSAGE\_STOP\_PROGRESS\_DIALOG*);

}

**break**;

}

With Peer Group Manager:

**case** (*JOIN\_SESSION*): {

/\*

\* To join a group, the only information needed is the group name,

\* obtained from the foundAdvertisedName callback method in this case.

\*/

Status status = mPeerGroupManager.joinGroup((String) msg.obj);

logStatus("PeerGroupManager.joinGroup() - groupName: " + (String) msg.obj, status);

**if** (status == Status.*OK*) {

/\*

\* To communicate with a ProxyBusObject, we need the

\* BusObject's remote interface. Getting the interface

\* requires the Peer Id of the peer who owns the BusObjects,

\* the name of the group both you and the peer are in,

\* an object path descriptor, and the interface of the class.

\*

\* The service created the BusObject as well as the group,

\* and the service's Peer Id is obtained here by using the

\* getGroupHostPeerId method.

\*/

mSimpleInterface = mPeerGroupManager.getRemoteObjectInterface(mPeerGroupManager.getGroupHostPeerId((String) msg.obj), (String) msg.obj, "/SimpleService", SimpleInterface.**class**);

mHandler.sendEmptyMessage(*MESSAGE\_STOP\_PROGRESS\_DIALOG*);

}

**break**;

}

Disconnecting and PingingThe changes to disconnecting in the Simple Client are symmetrical to that in Simple Service. Furthermore, no change to the ping code is required.

RESULTS  
Reworking Simple Service and Simple Client to use the Peer Group Manager simplifies the code significantly. With Simple Service, the lines of code in the Bus Handler class dropped from 71 to 33 (not counting comments and white space). Furthermore, Simple Client’s Bus Handler is reduced from 102 lines of code to 63. Beyond this, Peer Group Manager abstracts some AllJoyn concepts such as port numbers, session options, and advertisement while using the more readable group names rather than session id for group identification.