Cast Companion Library for Android

Introduction

This document describes the Cast Companion Library for Android (CCL). Throughout the document, there may be references to the Cast Video application that is built using the CCL to show how the library can be used in a practical example.

CCL is written with the following objectives in mind:

- To provide a wrapper around the Cast SDK and related Google Play services to hide the mundane tasks that are needed for casting.
- To provide a collection of ready-to-use Cast-related components and features that are strongly recommended by the UX Guidelines.
- To provide an example of how the Cast SDK can be used to accomplish more complex tasks.

Since playing media is a common use case on a Chromecast device, CCL provides extensive support for casting and managing media content. In addition, it supports applications that are strictly using custom data channels.

Here is a collection of features that CCL provides for media centric applications:

- Wrapper for device discovery
- Customized notification
- Customized Cast Menu (Media Router controller dialog)
- Lock Screen remote control via `RemoteControlClient`
- Player Control while casting
- Global access via mini-controller

- Pain-free session recovery
- A single custom data channel for sending and receiving custom data
- Ability to check the installed Google Play services for compatibility

And for data centric applications:

- Wrapper for device discovery
- Wrappers for registering and using multiple custom data channels
- Pain-free session recovery
- Ability to check the installed Google Play services for compatibility

In the subsequent sections, we will describe how each of these features can be accessed by an application. In what follows, we use “client” to refer to the application that uses CCL.

Overall Design

Most of this document focuses on media-centric applications. Toward the end, we come back and discuss how data-centric apps can use this library.

To maintain the state of the application in regards to connectivity to a Cast device, there needs to be a central entity that transcends the lifecycles of individual activities in the application. The CCL’s main component, the `VideoCastManager` is the entity that does that. This class is a singleton that is responsible for managing the majority of the work by providing a rich set of APIs and callbacks, maintaining the state of the system (such as connectivity to the Cast device, status of the media on the receiver, etc), updating various components such as the CCL Notification Service and Mini-Controller.

In order to have a notification that lasts even if the client application is killed (directly by user or indirectly by the system) while casting, CCL starts a Cast Notification Service. The Cast UX guidelines request that the system notification for a Cast application to be visible only when the application is not visible. By providing simple hooks for the client, CCL handles this task as well.

Here is a subset of classes and their main responsibilities that will be discussed in more details in subsequent sections:

- *`VideoCastManager`*: Specifically designed for video-centric applications, this is a singleton that clients interact with directly. It internally uses other classes and components of CCL to manage and maintain the state of a video-centric application.
- *`VideoCastNotificationService`*. A local service that allows notifications to appear when needed beyond the availability of the main application.
- *`MiniController`*. A compound control that provides a mini-controller for the client.
- *`RemoteControlClientCompat`*. A wrapper around the `RemoteControlClient` to enable a graceful fallback for versions of Android that do not support `RemoteControlClient`. On those versions, it simply does nothing.
- *`VideoCastControllerActivity`*. Provides a default implementation of the Player

Control screen that clients can use with their application.
- *`DataCastManager`*. Specifically designed for data-centric applications, it is a singleton that clients interact with directly. It internally uses other classes and components of CCL to manage and maintain the state of the data-centric application and provides means to register one or more namespaces for data communication between a sender client and a receiver application.

The MediaRouter and Cast SDK, in collaboration with the Google Play services, provide a collection of asynchronous calls that start with the discovery method and continues to route selection, device connection, `RemoteMediaPlayer` creation, etc. For each of these asynchronous calls, there is one or more callbacks that inform the caller of the success and result of the API calls, and are generally a signal that moving to the next asynchronous call is now permitted. Not all the activities or components in a client app are interested in receiving all the callbacks. CCL makes it easy for client components to only register to a subset of callbacks. Although most of the Cast API calls are made by CCL, clients can still be notified of all relevant events and callbacks, if they choose to.

Dependencies and Project Setup

Here is a list of dependencies for CCL:

- android-support-v7-appcompat
- android-support-v7-mediarouter[1] (this has a dependency on android-support-v7-appcompat)
- Google Play services (which includes the Cast SDK)

Since the two support libraries contribute resources as well, you cannot simply satisfy the dependency by including their jar files; instead you need to import them as library projects[2]. Also note that these support libraries were updated along with the release of KitKat to revision 19; if you have earlier versions of these support libraries, you need to update them first. As a result, CCL assumes you are using `ActionBarCompat` in your project.

Your client project then needs to list CCL as its only dependency for Cast related functionalities. In other words, your client application does not need to directly import or declare a dependency on the support libraries.

Since Cast APIs are mainly provided by the Google Play services, the CCL library provides a convenient static method that all client applications should call at their startup to verify that the correct version of the Google Play services is available on the device. If the Google Play services is missing or needs to be updated, a dialog will inform user and direct her to go to the Play Store to download the appropriate version. If, however, the Google Play services is not enabled, it will direct user to the device settings page to correct the issue. To enable this important validation check, here is a sample code that calls this functionality in the onCreate() of your launcher

---

1 Be careful that you need to use v7 support version of MediaRouter library and not the one that is included in the Android framework

2 See http://developer.android.com/tools/support-library/setup.html for how this can be done

Activity of your application:

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        BaseCastManager.checkGooglePlaySevices(this);
        .....
    }
```

How to Use the Cast Companion Library

In what follows, we will mainly focus on the video-centric applications and use the `VideoCastManager` but in many cases, the same applies to the `DataCastManager` as well.

Initializing `VideoCastManager`

The client application needs to instantiate and initialize an instance of `VideoCastManager` as early as possible. During the initialization step, the client should provide certain data that CCL needs to perform its task. It is recommended that the clients do this step in their Application class, so they can easily access it in all activities of the client applications. The following snippet shows how the Cast Video reference application does that:

```
    public static VideoCastManager getVideoCastManager(Context ctx) {
        if (null == mCastMgr) {
            mCastMgr = VideoCastManager.initialize(ctx, APPLICATION_ID,
null, null);
            mCastMgr.enableFeatures(VideoCastManager.FEATURE_NOTIFICATION |
                                    VideoCastManager.FEATURE_LOCKSCREEN |
                                    VideoCastManager.FEATURE_DEBUGGING);
        }
        mCastMgr.setContext(ctx);
        return mCastMgr;
    }
```

The "`initialize`" static method takes four arguments: a Context object, application ID, and an activity that provides the "Player Control" functionality. If the client wants CCL to provide its default Player Control page, the third argument to the `initialize()` should be set to null. Finally, the last argument is used when there is a need to have custom data channel. Leave this at `null` if no custom data channel is required. See the section on data channel later in this document to see how this can be arranged.[3]

Immediately following the initialization, clients should inform the `VideoCastManager` what

3 `initialize` will throw a `RunTimeException` if it detects that a wrong version of Google Play services is installed, is missing or is not activated.

features they want CCL to provide by calling the static `enableFeatures` method. Currently there are three features that can be turned on: Notifications, Lock Screen remote controller and logging in the Google Play services. To specify which features should be enable, construct an OR-ed expression of the desired features and pass that to the `enableFeatures` method. All features are disabled by default.

Since each Activity in your application needs to access the `VideoCastManager` instance, get a reference to that in each activity in its "`onCreate()`" and "`onResume`()" lifecycle callbacks by calling:

```
    mVideoCastManager = CastApplication.getVideoCastManager(this);
```

Doing so will also update the context for the `VideoCastManager` instance.

Remark. *In performing certain tasks, CCL needs to have a reference to a* `Context` *object from the client application (for example, to access client resources or system services). As a client application transitions between various internal activities, it should update this* `Context` *object; for example if one transitions from one activity to another within the client, it is best to call the* `VideoCastManager.setContext()` *method and pass the new context to CCL. This is a very important step to ensure that CCL can do its job; for example if an error dialog needs to be presented in the UI, CCL needs to have the right context. That is the reason why the* `getVideoCastManager()` *method is calling the* `setContext()`. *It is best if each activity in the client calls this method in its* `onResume()`. *The* `getVideoCastManager(this)` *does this for you if you decide to set it up as shown in the above snippet, otherwise it is your responsibility to call* `CastManger.setContext()` *at the appropriate time.*

Adding Cast Button to Action Bar

After having `VideoCastManager` set up, the next step is to put the Cast button in the Action Bar. The assumption here is that your activity is subclass of ActionBarActivity (from appcompat-v7 library):

A. Add the following snippet to the xml file that defines your menu:

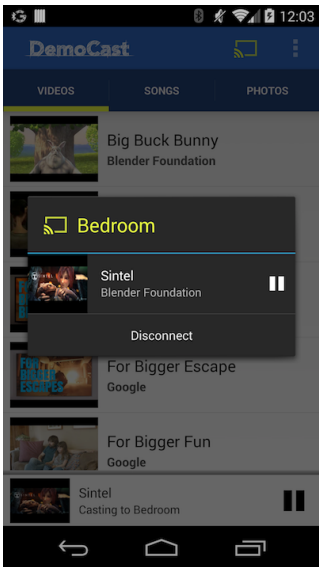
```
   <item
        android:id="@+id/media_route_menu_item"
        android:title="@string/media_route_menu_title"

app:actionProviderClass="android.support.v7.app.MediaRouteActionProvider"
    app:showAsAction="always"/>
```

B. Add the following line to the "`onCreateOptionMenu`()" of all your activities:

```
    @Override
```

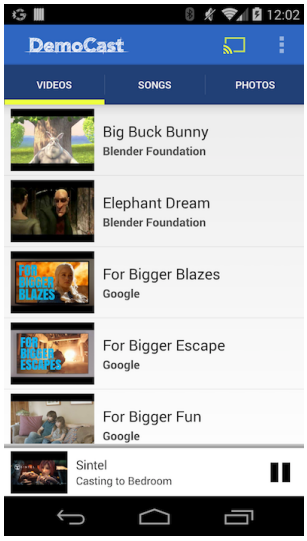
```
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        getMenuInflater().inflate(R.menu.main, menu);
       mVideoCastManager.addMediaRouterButton(menu,
R.id.media_route_menu_item);
        return true;
    }
```

Adding the above line will put a Cast button in your Action Bar and puts the `VideoCastManager` in charge of all the required plumbing to provide the appropriate dialogs, etc. Note that this additional line also returns a pointer to the MenuItem that represents the Cast button if you need to have access to it. CCL also provides a custom



Cast MediaRouter dialog when the client is connected:

Adding Mini-Controller



The mini-Controller is a small persistent component that enables users to quickly see the content that is playing on the Cast device, to perform basic operations on the remote media (such as play/pause) and to provide a way to get back to the Cast Player page (by clicking on album art). The mini-Controller should be visible if and only if the remote video is playing[4].

The CCL provides a compound component for the mini-controller that can be added to the layout XML of your pages. Here is an example:

```
    <com.google.sample.castcompanionlibrary.widgets.MiniController
        android:id="@+id/miniController1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:background="@drawable/shadow4"
        android:visibility="gone">
    </com.google.sample.castcompanionlibrary.widgets.MiniController>
```

Then you need to register this component with the instance of `VideoCastManager` by adding the following lines to your Activities' "`onCreate()`":

```
    mMini = (MiniController) findViewById(R.id.miniController1);
    mVideoCastManager.addMiniController(mMini);
```

Remember to unregister this component when you are done by adding the following line of code

4 For Audio-only content such as music, it may be required to have this component available even for local playback.

to "`onDestroy()`" for example:

```
    mVideoCastManager.removeMiniController(mMini);
```

Upon registration, the CastManger handles updating the metadata shown in this component, updating the status of the play/pause button, and also the visibility of the whole control (e.g. if you disconnect your client application from the Cast device, it will hide this component.)

Notifications

Notifications will be provided for you if you enabled that feature after initializing the `VideoCastManager`. What remains to be done is to help CCL discover when the client application is visible (to hide the notification) and when it is not visible (to show the notification). This can be accomplished by adding the following line to your "`onResume()`":

```
    mVideoCastManager.incrementUiCounter();
```

and the following line to "`onPause()`" of all your activities:

```
    mVideoCastManager.decrementUiCounter();
```

Lock Screen Controllers and Volume

CCL can provide default lock screen controllers based on the RemoteControlClient. If desired, this feature needs to be enabled at the initialization time of the VideoCastManager. When this feature is enabled, a Play/Pause button will be shown on Android devices running Jelly Bean or above. On KitKat devices, the layout of the lock screen controllers is different and uses a full-screen album art while on Jelly Bean it is a small version of the album art.

In addition, when this feature is enabled, CCL provides the ability for users to control the Cast device's system's volume even if the application is in the background. On KitKat devices, it provides that capability even if the screen is off.

Cast Player Screen

If during the initialization of `VideoCastManager` you opted for the default Cast Player screen, you will get that without any additional settings.

The Cast Player page shows a full screen artwork with some additional metadata and some controls to manage the playback of the media on a remote device. It is important to provide a larger size image for your media for the best experience.

If user clicks on the artworks in the mini-controller or notification, a transition to the Cast Player page will happen automatically. If, however, you want to start casting from within the client application, the Cast Manager provides a simple mechanism for you to launch that screen:

```
mVideoCastManager.startCastControllerActivity(context, mSelectedMedia,
position, autoPlay);
```

Here, `mSelectedMedia` is the `MediaInfo` object that represents your desired media that needs to be casted, `position` (integer) is the starting point of the video and the `autoPlay` (boolean) signals whether you want to automatically start the playback of the media remotely or not.

Remark 1. *To have all the metadata for a movie available when needed, you need to build and populate an instance of `MediaInfo (defined in the Cast SDK)` to represent a movie. The field that will be used in various places are:*

- metadata: use an instance of `MovieMetadata` that is populated with:
  - title
  - sub-title
  - studio
  - images
    - index 0: small size image used for notification, mini-controller and Lock Screen on JB
    - index 1: large image, used on the Cast Player page and Lock Screen on KitKat
- contentId (populate this with the URL pointing to the media)
- contentType: e.g. “video/mp4”
- streamType: typically of type `MediaInfo.STREAM_TYPE_BUFFERED`

Remark 2. *It is very tempting to try to pass around `MediaInfo` objects from one activity to another. For example, if user is browsing a catalog and selects a video to be taken to the next activity that has detailed info on the media, it is ideal to use the `MediaInfo` that represents the selected media and pass that to the next activity. Unfortunately, `MediaInfo` is neither `Serializable` nor `Parceable` so you cannot pass that in an intent; however, CCL provides a wrapper, called `MediaInformationProxy`, that is serializable and can be passed around. Conversion between `MediaInfo` and `MediaInformationProxy` is a simple operation that can be done by methods provided in the proxy class itself:*

```
    public static MediaInformationProxy fromMediaInformation(MediaInfo
info);
    public MediaInfo toMediaInformation();
```

Session Recovery

To provide a seamless experience for users, we need to support the following scenario: assume that a client application is connected to a Cast device. If the user closes the application without calling an explicit disconnect, and later on opens the same application, the client application needs to try to re-establish the connection to the same device as long as the Cast device is still running the same session that the user had started earlier. If system has not yet killed your application, this does not require any effort but if your application has been killed by the system (or your connectivity was lost because you stepped outside of the wifi range, etc), some work needs to be done. To accomplish that, CCL provides an API that client application can call in the "`onCreate()`" of the launcher activity:

```
  mVideoCastManager.reconnectSessionIfPossible(context, showDialog,
timeoutInSeconds);
```

This method makes a best-effort attempt to reconnect to the same device if the same session is still running. The (boolean) showDialog parameter causes a dialog to be shown[5] during this period and the optional (int) timeoutInSeconds limits how long (in seconds) the whole reconnection effort should be attempted. If the last parameter is not present, a 5 seconds timeout will be enforced. If during this timeout period the client was not able to establish a connection, the effort will be stopped.

Single Custom Data Channel

For media-centric applications, CCL also provides a single data channel (in addition to the standard media channel) to enable out-of-bound communication between the sender and receiver applications. For example, a sender application may need to provide a mechanism for its users to give a thumbs-up or thumbs-down when they are watching a content, or toggle the Closed Caption. These types of communications require a custom data channel.

To facilitate this, the last argument to the `initialize()` call to initialize the `VideoCastManager` can be used. You can choose a namespace of your choice and pass that as the last argument to that call. Then CCL will set up the sender part of the data channel and provides a single API for you to send a message to the receiver (`sendDataMessage()`), and two callbacks for you to remain informed when a message is received from receiver and when the send command encounters an error. CCL also provides an API for you to remove your custom data channel (`removeDataChannel()`). Refer to the JavaDoc for the library for the documentation on these APIs and callbacks.

Support for data-centric applications

If you are working with an application that is mainly data-centric and need one or more custom data channels but no media data channel, you can initialize and use a different class in the library, the "`DataCastManager`". This class is the equivalent of the `VideoCastManager` and behaves

---

5 The text shown in the dialog can be customized via a string resource with id `session_reconnection_attempt`

similarly when you are setting it up. The initialization of this singleton is done by calling the following method:

```
    initialize(Context context, String applicationId, String...namespaces)
```

As is clear from the above, you can register one or more namespaces by appending them as last arguments of this method. Note that you can add/remove namespaces later on by calling:

```
   addNamespace(String namespace); // to add a new namespace
   removeNamespace(String namespace); // to remove
```

The library takes care of setting up the data channel(s) for you. You can then use the following API to send a message on any of these channel(s):

```
long sendDataMessage(String message, String namespace)
```

This will send a message and returns a message ID that can be used to track the response to the message. Messages that are sent to the sender from the receiver can be captured by extending `DataCastConsumerImpl` class and overriding the following callbacks:

```
    public void onMessageReceived(CastDevice castDevice, String namespace,
String message) {}

    public void onMessageSendFailed(CastDevice castDevice, String namespace,
long messageId, int errorCode) {}
```

Hooking into Lifecycle Callbacks

The CCL library tries to handle most common scenarios but it also provides clients with access to events that can be useful in more advanced cases. For media-centric applications, clients can implement `IVideoCastConsumer` interface and register the implementation with the `VideoCastManager` instance. In case of data-centric apps, implement `IDataCastConsumer` and register the implementation with the instance of `DataCastManager`.

If you refer to the documentation on these interfaces, you will notice a long list of callbacks that range from connectivity success to application launch failure, etc. To make your life easier, the library introduces two no-op implementations of these interfaces: `VideoCastConsumerImpl` and `DataCastConsumerImpl`. Consequently, you can extend these classes and ignore all the interface methods except the ones that you are interested in and only override those methods. For example, here is how you can extend and register one:

```
        mVideoCastManager.addVideoCastConsumer(new VideoCastConsumerImpl() {
            @Override
```

```
        public boolean onApplicationConnectionFailed(int errorCode) {
            // do what your application wants
            return true;
        }
    }
```

Remark 1. *In order to use the custom data channels properly, you need to implement the custom data related methods of these interfaces to at least receive messages from the receiver.*

Remark 2. *Most methods in these two interfaces have no return values. However, there are few of them that should return a* `boolean`*. These methods are related to failures in various stages (for example "launching application failed"). CCL can provide a standard error dialog for your application if your implementation of these methods returns* `true` *(that is the behavior of the default implementation classes, as well). If you want to provide your own error dialogs, make sure you override these methods and return* `false`*.*

Remark 3. *All the lifecycle callbacks are called on the Main (i.e. UI) Thread.*

Configuration and Setup

Manifest file

Your `AndroidManifest.xml` file needs to include certain elements and metadata for the library and Cast SDK proper operation. Here is a list of requirements:

- Minimum SDK version: the minimum SDK version that the library and Cast SDK work with is 10 (GingerBread).
- Permissions: The following is the minimum required permissions:
    - `android.permission.INTERNET`
    - `android.permission.ACCESS_NETWORK_STATE`
    - `android.permission.ACCESS_WIFI_STATE`
- Application Metadata: add the following metadata to your application declaration:

```
    <meta-data
                android:name="com.google.android.gms.version"

    android:value="@integer/google_play_services_version" />
```

- Makes sure your application's "theme" is correctly set based on the min SDK version. For example, you may need to use a variant of `Theme.AppCompat` or one that inherits from these themes. Note that this is not a requirement for Cast, per se.
- Declare the VideoCastControllerActivity:

```
   <activity
android:name="com.google.sample.castcompanionlibrary.cast.player.Video
CastControllerActivity"
```

```
            android:screenOrientation="portrait"
            android:label="@string/app_name"
            android:launchMode="singleTask"
            android:parentActivityName="*PARENT_ACTIVITY*"
            android:theme="@style/Theme.DemocastOverlay
">
            <meta-data
                android:name="*PARENT_ACTIVITY*" />

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
```

There are a few things that you need to set in the above snippet. Assume that user clicks on your application's notification from the status bar while casting. The desired behavior is to open your application and direct user to the `VideoCastControllerActivity`. However, this is not the main launcher activity of your application, so if the user were to press the back button, she should not be sent out of the application. The correct behavior would be to send the user to the parent activity; the one that in a normal execution of your application would show up when user goes back fro the `VideoCastControllerActivity`. This special behavior can be achieved by creating a "back stack" for the `PendingIntent` in the notification service. CCL can handle this for you if you declare in the manifest what activity is the correct parent activity; you need to enter the name of that activity (possibly a fully qualified name) in the two places in the above snippet where the `PARENT_ACTIVITY` is used[6]. In addition, the `VideoCastControllerActivity` is best presented if you use an overlay Action Bar theme; CLL provides one named `DemocastOverlay` but you can update that in the above snippet if needed.

- Declare the following receiver:

```
<receiver
android:name="com.google.sample.castcompanionlibrary.remotecontrol.MediaInten
tReceiver" >
   <intent-filter>
      <action android:name="android.media.AUDIO_BECOMING_NOISY" />
      <action android:name="android.intent.action.MEDIA_BUTTON" />
      <action android:name="android.media.VOLUME_CHANGED_ACTION" />
      <action
android:name="com.google.sample.cast.refplayer.notification.toggleplayback" /
>
      <action
android:name="com.google.sample.cast.refplayer.notification.stop" />
   </intent-filter>
</receiver>
```

6 See http://developer.android.com/guide/topics/ui/notifiers/notifications.html#NotificationResponse for more information

This receiver is called when user interacts with the VideoCast`NotificationService` or `RemoteControlClient`; each of these components broadcast a message that this receiver catches and handles appropriately.

- Declare the following service (for `VideoCastNotificationService`):

```
<service
android:name="com.google.sample.castcompanionlibrary.notification.VideoCastNo
tificationService"
       android:exported="false">
   <intent-filter>
      <action
android:name="com.google.sample.cast.refplayer.notification.toggleplayback" /
>
      <action
android:name="com.google.sample.cast.refplayer.notification.stop" />
      <action
android:name="com.google.sample.cast.refplayer.notification.visibility" />
   </intent-filter>
</service>
```

Configurable Messages

Here is a list of strings defined in the strings.xml file that the application client can override in its own strings.xml:

```
    <!--  Used in Error Dialog -->
    <string name="ok">OK</string>

    <!-- Title of the Error Dialog -->
    <string name="error">Error</string>

    <!-- Shown an Video Cast Controller Activity -->
    <string name="casting_to_device">Casting to %1$s</string>
    <string name="loading">Loading&amp;#8230;</string>

    <!-- Used in Router Contrller Dialog -->
    <string name="no_media_info">No media information available</string>

    <!-- Session Recovery Dialog message -->
    <string name="session_reconnection_attempt">Attempting to recover
previous session&amp;#8230;</string>

    <!-- onApplicationConnectionFailed() errors -->
    <string name="failed_to_launch_app">Failed to lauch application</string>
    <string name="failed_app_launch_timeout">The request to launch the
application has timed out!</string>
```

```
    <string name="failed_to_find_app">The application you are trying to
launch is not availabled on your Chromecast device</string>

    <!--  Error messages for the play.pause in mini player -->
    <string name="failed_to_play">Failed to start the playback of
media</string>
    <string name="failed_to_pause">Failed to pause the playback of
media</string>
    <string name="failed_unknown">An unknown error was encountered</string>

    <!--  Failure to connect -->
    <string name="failed_to_connect">Could not connect to the
device</string>
```