Mobile Accelerator SDK

Version 20.1.2

Android Integration Guide

1 Summary

This document details the process of integrating MAP SDK with your Android application to accelerate web traffic.

2 Introduction

The SDK internally takes care of pre-positioning the content based on user preferences and policies set up between client and server. SDK provides APIs (networking libraries) to be used by developers that takes care of acceleration and stats collection.

The SDK provides API for developers to access real time network conditions such as congestion state. This information can be used to augment user experience by taking necessary action based on network state.

In addition, SDK also provides APIs for logging user events which could be used to associate traffic originated from the app with events such as the click of a button.

3 Getting Started

There are primarily two ways of accessing content over the network. One of the most common ways is to use standard HTTP libraries such as *HttpURLConnection*.By default, after successful initialization SDK intercepts all the HttpUrlConnection and HttpsUrlConnection requests.

Another way to access content over the network in an app is through WebViews. SDK provides an API for developers which delegates all network-related requests originating from WebViews to the *MAP SDK.*

The *SDK* also collects network-related statistics (such as HTTP time to first byte, request size, response size, duration etc.) alongside serving content. These stats are sent periodically to a server and can be later accessed via portal.

3.1 Requirements and Dependencies

The SDK supports API 15 and above

3.2 Installing the Android SDK

Download the SDK and unzip it.

In a file explorer (*not* Android Studio), drag the unzipped map-**sdk-***version***.aar** file into the /app/libs directory in your project's root directory.

In Android Studio, edit the build.gradle file in the **app** directory (*not the one in the root folder*) and edit the dependencies sub-section to include .AAR file and following libraries:

```
useLibrary 'org.apache.http.legacy'

dependencies {
       implementation fileTree(dir: 'libs', include: ['*.jar'])
       implementation 'com.google.firebase:firebase-messaging:17.3.4'
       implementation 'com.android.support:support-v4:26.1.0'
       implementation (name:'map-sdk-version', ext:'aar')
}
```

```
Or use the direct maven dependency from jcenter()

implementation 'com.akamai.android:map-sdk:<version>@aar'
```

Note : You will need to add google maven repo(for **`support-v4:26.1.0)`** in the project's root build.gradle also

Example :

```
allprojects {
  repositories {
     mavenCentral()
     jcenter()
     google()
  }
}
```

For Apps that target Android P
<uses-library android:name="org.apache.http.legacy" android:required="false"/> is needed in AndroidManifest.xml

MAP SDK version 20.1.1 and later has been updated to use Firebase Messaging as Google has deprecated GCM. If you are updating from a previous version of SDK to version 20.1.1 or later, please make sure to update the app's build.gradle file with firebase-messaging dependency as above . Remove the GCM dependency and add the FCM as suggested above. For additional information on getting background notifications working for prepositioning follow section 4.3

Updating your Android Manifest
Client AndroidManifest.xml will need to be updated in order to complete the SDK integration.

```
<application
….


<provider
android:name="com.akamai.android.sdk.db.AnaContentProvider"
<!-- <your_package_name> Refers to unique package id of the app -->
android:authorities="<your_package_name>.AnaContentProvider" >
</provider>

<!-- Refers to sdk init file in res/xml/ -->
<meta-data
android:name="com.akamai.android.sdk"
android:resource="@xml/akamai_sdk_init" />


….
</application>
```

Creating the SDK initialization file referred in the manifest

The SDK uses an xml file to look up the license keys and other information to authorize the client app. This information is stored in a file android_sdk_init.xml in the client app's resources folder. The sample file is shown below.

In …./main/res/xml (create if it doesn't exist!) folder, add a new file android_sdk_init.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<com_akamai_sdk_init>
<!-- SDK license key created on portal-->
```

```
<com_akamai_sdk_license_key></com_akamai_sdk_license_key>
<!-- In case of MAP license, this is comma separated list of segments to register with
(optional)-->
<com_akamai_sdk_segments></com_akamai_sdk_segments>
<!--Regional info (optional)-->
<com_akamai_sdk_region></com_akamai_sdk_region>
<!--SDK user id specified by the app (optional)-->
<com_akamai_sdk_sdk_user_id></com_akamai_sdk_sdk_user_id>

 </com_akamai_sdk_init>
```

The SDK requires these permissions for full functionality:

```
<manifest . . . >
     <uses-permission android:name="android.permission.INTERNET" />
     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
     <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
     <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

4 Integration with your Android Application

4.1 Initialization

The first step on app start would be to just create an instance of VocService using VocService.createInstance (Context applicationContext) which initializes and registers the sdk with parameters provided in android_sdk_init.xml. This should be done on main application create or onCreate of the main activity.

```
// Create a VocService instance
VocService vocService = VocService.createVocService(getApplicationContext());
```

Once initialized, any content listed in the subscribed segments automatically starts getting prepositioned based on network policy(wifi/cellular) defined.

4.2 Updating segment subscription

The list of subscribed segments may be changed any time after registration. Pass an array of segment names to the VOC service call *updateSegmentSubscription*.

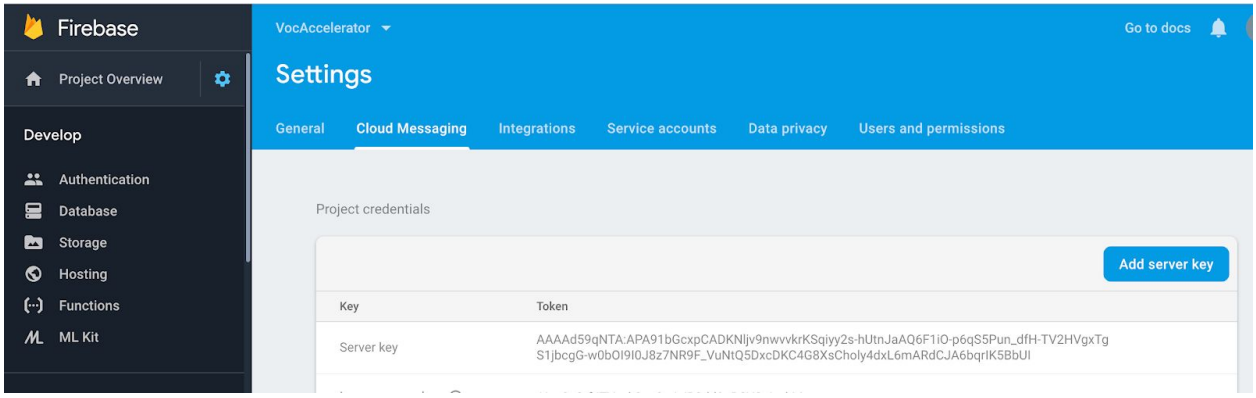
```
/**
* API method to update segment subscription once registered. Multiple calls to this API
would update previous subscription.
* Note - Changes to the subscription are not reflected immediately and are stored in a
persistent storage.
* SDK notifies server about any changes (if any) during next sync with server.
* This method is not synchronized and caller must ensure the sequence if called multiple
times.
* @param segments - Array containing segment names. Passing an empty array would clear all
segment subscriptions including current ones.
*/
public VocServiceResult updateSegmentSubscription(@NonNull final String []segments)
```

4.3 Integrating with Firebase Cloud Messaging

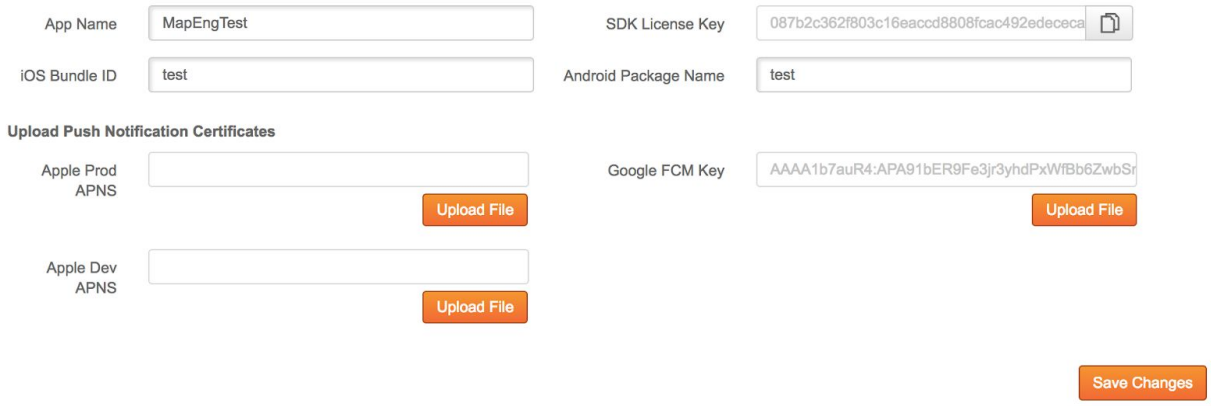
MAP SDK uses Firebase Cloud notifications to mainly sync prepositioning content and sdk config in background, **If the app does not have a need for prepositioning content in background then this step is optional**.

In order for FCM background notifications to work - follow the firebase messaging integration guide as documented at Google Firebase website(https://console.firebase.google.com) . After firebase is integrated in the app, complete the following two steps:

1. Add/Update the server api key to the portal for the required app. The server api key could be found on the firebase console project settings as shown below.



Then add/update the key to MAP SDK license portal in the “Google FCM Key” field on MAP Portal as shown below. The API Key is used by MAP Control Server to trigger map related notifications - for prepositioning any content configured for the app .



2. The app needs to provide hooks in its implementation of its FirebaseMessagingService to pass the messages to SDK. A returned ‘true’ value means the message was meant for the MAP SDK and the app doesn’t need to do any further processing on it. A ‘false’ is returned if it is not a MAP SDK message. An example is included below:

```
public class AppFirebaseMessagingService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage message) {
        // If handleFirebaseMessage returns true, the message is for map sdk.
        if (VocService.createVocService(getApplicationContext())
                                .handleFirebaseMessage(message)) {
            return;
        }
```

```
        // Handling for app messages.
        ...
    }

    @Override
    public void onNewToken(String newToken) {
        VocService.createVocService(getApplicationContext()).updateFirebaseToken(newToken);

        // App handling for tokens.
        ...
    }
}
```

5 API Reference

In this section, we see examples of how to use APIs provided by the SDK.

5.1 Using SDK with Android HttpsUrlConnection/HttpUrlConnection

By default, after successful initialization SDK intercepts all the HttpUrlConnection and HttpsUrlConnection requests. There is no additional code change required by the application. SDK intercepts these requests by setting a global URLStreamHandlerFactory. All the relevant network statistics related to each request will be captured by the SDK.

5.2 Using SDK with WebViews

In order to accelerate traffic originated from WebViews, the SDK provides following custom WebViewClient.
AkaWebViewL21Client - For Android API level 21 and above.
AkaWebViewL15Client - For Android API level 15 and above.

Both WebViewClients delegate all network calls via the MAP SDK library internally. The only difference between the two is that the L21 WebViewClient uses newer APIs added in level 21 and above.
The caller can use the appropriate client depending on API level as follows:

```
WebView webView;
...
if (Build.VERSION.SDK_INT >= 21) {
        webView.setWebViewClient(new AkaWebViewL21Client());
```

```
} else {
        webView.setWebViewClient(new AkaWebViewL15Client());
}
```

Note - Currently, both WebViewClients do not handle POST requests as there is no API provided by Android platform to access POST data.

5.3 Using SDK with Third party HTTP client wrappers

In order to accelerate traffic originating from Third party http clients like Okhttp, Retrofit, Picasso, an interceptor needs to be added to the request. Sample interceptor classes for all these libraries have been added in the wrappers folder of the voc accelerator example.
Note : SDK has to be registered successfully for the wrappers to work as expected.

OkHttp

An interceptor for OkHttp needs to be added to the OkHttpClient.Builder as below

```
OkHttpClient client = new OkHttpClient.Builder()
        .addInterceptor(new AkaOkHttpInterceptor())
        .build();
Request request = new Request.Builder()
        .url(uri)
        .build();
Response response = client.newCall(request).execute();
```

Retrofit - Version 1

An interceptor for Retrofit needs to be added to the RestAdapter class as below

```
RestAdapter.Builder builder =
        new RestAdapter.Builder()
                .setEndpoint(MapUrlTestMockedServer.HTTPSHOST)
                .setClient(new AkaRetrofitClient(getContext()));
RestAdapter adapter = builder.build();
RetrofitApi api = adapter.create(RetrofitApi.class);
```

Retrofit - Version 2

An interceptor for Retrofit 2 needs to be added as the client to retrofit as shown below

```
Retrofit retrofit = new Retrofit.Builder()
       .baseUrl(uri)
       .addConverterFactory(GsonConverterFactory.create())
       .client(AkaRetrofit2Client.getClient())
       .build();
Retrofit2Api client =  retrofit.create(Retrofit2Api.class);
Call<ResponseBody> call = client.get(path);
```

Picasso

An interceptor for Picasso needs to be added to the Picasso.Builder as below

```
Picasso picasso = new Picasso.Builder(getContext())
       .downloader(new AkaPicassoDownloader(getContext()))
       .build();
Picasso.setSingletonInstance(picasso);
```

5.4 Custom Event Tracking

Custom events are actions triggered due to some activity performed by the end user, such as a button click. The SDK provides APIs that are helpful for developers to clock one or more custom events.

We classify events as timed and instantaneous events. Timed events are ones that have a start and an end point associated with them. Apart from clocking the duration between start and stop, such events can be used to determine network calls originated between start and stop. Please ensure not to add any data to the event name that has privacy implications.

```
/**
* API method to track timed events. Caller must call {@link #stopEvent(String)} to mark an
event as complete.
* @param eventName Name of the event to track. Same name should be used for {@link
#stopEvent(String)}
*/
public void startEvent(String eventName);
```

```
/**
* API method to track timed events. Caller must call {@link #startEvent(String)} before
calling this method.
* @param eventName Name of the event used during {@link #startEvent(String)}
*/
public void stopEvent(String eventName);
```

Instantaneous events, unlike timed events, do not have a start and stop associated with them. Such events can be used to log a set of sequence or form a timeline of operations. Note that instantaneous events are in Tech Preview and are not yet displayed in the portal.

```
/**
* API method to track instantaneous events.
* @param eventName Name of the event to track.
*/
public void logEvent(String eventName);
```

Examples

Sample usage of above webview and user events API.

```
VocService vocService;
...
vocService.logEvent("Initialization");
...
vocService.logEvent("Clicked XYZ.com");
String url = "http://www.xyz.com/abc.html";
WebView webView;
...
webView.setWebViewClient(new AkaWebViewL21Client() {
        @Override
        public void onPageStarted(WebView view, String url, Bitmap favicon) {
                super.onPageStarted(view, url, favicon);
                // url loading started.
                vocService.startEvent(url);
        }

        @Override
        public void onPageFinished(WebView view, String url) {
                super.onPageFinished(view, url);
```

```
                // url loading finished.
                vocService.stopEvent(url);
        }
});
```

5.5 Network Aware Experience

The SDK provides APIs that helps a developer to access client side network quality state to augment client request.

For example - Load the absolute minimum needed for a particular user event. The example below shows how loading of a web page can be tweaked based on network quality state.

```
VocService vocService;
...
int networkQualityState = vocService.getNetworkQuality();

switch(networkQualityState) {
        case VocNetworkQualityStatus.POOR:
                // load a webpage with no images.
                break;
        case VocNetworkQualityStatus.GOOD:
                // Load a webpage with medium quality images.
                break;
        case VocNetworkQualityStatus.EXCELLENT:
                // Download content.
                break;
}
```

5.6 Customer Pinned Certificates

The sdk can be configured to use pinned certificates (SSL Socket Factory) and/or custom hostname verifiers to download prepositioned content. The certificates and hostname verifier

can be configured on a per host basis and should be done before registering since the content starts downloading automatically after registration is successful. If configured, the sdk will use the authentication parameters while connecting to host part in the prepositioned url.

```
// Create a VocService instance
VocService vocService = VocService.createVocService(getApplicationContext());

MapConnectionParameters parameters = new MapConnectionParameters();

paramaters.setSSLSocketFactory(yourCustomSocketFactory);
parameters.setHostnameVerifier(yourCustomHostnameVerifier);

vocService.setCustomConnectionParameters("www.akamai.com", parameters);
```

5.7 Debugging APIs

SDK provides APIs for developers to debug requests made through SDK. There are two logging levels supported viz. *DEBUG* and *INFO*. *INFO* is the default logging mode and contains logging with minimal output. *DEBUG* on the other hand is the enhanced logging mode. Logging levels can be changed at the runtime and is not persisted through multiple app sessions. All the APIs are supported through *Logger.java* class.

```
public enum LEVEL {
  /**
   * The default logging mode. This is the production level with minimal output.
   */
  INFO,
  /**
   * The enhanced logging mode for DEBUGGING purposes only.
   */
  DEBUG
}

/**
*
* @param level defines the SDK logging level.
*                The default level is LEVEL.INFO. This is the production level with minimal
output.
*                LEVEL.DEBUG is enhanced logging mode for DEBUGGING purposes only.
* Also see, {@link Logger.LEVEL}
```

```
*/
public static void setLevel(LEVEL level)


/**
*
* @param ctx
* @throws Exception if current log level is not set to LEVEL.DEBUG.
*/
public static void logCurrentConfiguration(Context ctx) throws Exception

/**
* Logs content corresponding to all the segments.
* @param ctx
 throws Exception if current log level is not set to LEVEL.DEBUG.
*/
public static void logExistingContent(Context ctx) throws Exception
```

Sample Debug Logs when Logger.setLevel(Logger.LEVEL.***DEBUG***);

Content Served from network

D/AkaSDKLogger: AkaURLConnection: Stats: URL: https://www.akamai.com/, Type: CACHE_MISS, Connection: cellular/LTE, RespCode: 200, ContentLength: 251167, StartTime: 1523471453095, Duration: 670, Ttfb: 618

Content served from Cache

D/AkaSDKLogger: D/AkaSDKLogger: AkaURLConnection: Stats: URL: https://www.akamai.com/, Type: CACHE_FETCH_ADHOC, Connection: cellular/LTE, RespCode: 200, ContentLength: 251167, StartTime: 1523471307223, Duration: 55, Ttfb: 6
On receiving Push notification

04-11 14:59:36.651 31167-31192/example.com.vocaccelerator D/AkaSDKLogger: AnaCacheService: com.akamai.anaina.PREPARE_SYNC
04-11 14:59:36.651 31167-31192/example.com.vocaccelerator D/AkaSDKLogger: AnaCacheService: Prepare sync cache request
0

5.8 Using SDK's AkaURLStreamHandler

If the application or a library within the app is setting its own global URLStreamHandlerFactory, then sdk provides a solution where developers can use `AkaURLStreamHandler` for specific requests that need to be handled by MAP

```
final String uri = "http://www.bestbuy.com/";

//Before
URL url = new URL(uri);

//After
URL url = new URL(null, uri, new AkaURLStreamHandler());
```

```
// HttpURLConnection usage.

final String uri = "http://www.bestbuy.com/";

HtttpURLConnection urlConnection = null;
try {
        // AkaURLStreamHandler that instantiates an object of AkaURLConnection.
        URL url = new URL(null, uri, new AkaURLStreamHandler());
        urlConnection = (HttpURLConnection) url.openConnection();
        ...
        // Download content using the InputStream
        InputStream inputStream = new BufferedInputStream(urlConnection.getInputStream());
        ...
        // Close the stream once done with the download.
        inputStream.close();
        } catch (IOException e) {
                e.printStackTrace();
        } finally {
 // Make sure to call HttpURLConnection#disconnect() to release resources and collect stats.
                if (urlConnection != null) {
                                urlConnection.disconnect();
                }
        }
```

Here is a side-by-side comparison of API usage for HttpURLConnection

Before	After

`final String uri = "http://www.bestbuy.com/";`
 	
`HttpURLConnection urlConnection = null;`
`try {`
`URL url = new URL(uri);`
 	
`urlConnection = (HttpURLConnection)`
`url.openConnection();`
`...`
`InputStream inputStream = new`
`BufferedInputStream(urlConnection.getInputStr`
`eam());`
`...`
`inputStream.close();`
`} catch (IOException e) {`
`e.printStackTrace();`
`} finally {`
`if (urlConnection != null) {`
`urlConnection.disconnect();`
`}`
`}`
	`final String uri = "http://www.bestbuy.com/";`
	
	`HttpURLConnection urlConnection = null;`
	`try {`
	`URL url = new URL(null, uri, new`
	`AkaURLStreamHandler());`
	
	`urlConnection = (HttpURLConnection)`
	`url.openConnection();`
	`...`
	`InputStream inputStream = new`
	`BufferedInputStream(urlConnection.getInputStr`
	`eam());`
	`...`
	`inputStream.close();`
	`} catch (IOException e) {`
	`e.printStackTrace();`
	`} finally {`
	`if (urlConnection != null) {`
	`urlConnection.disconnect();`
	`}`
	`}`

For all https requests use `AkaURLStreamHandler(true)`

```
final String uri = "https://www.akamai.com/";

HttpsURLConnection urlConnection = null;
try {
        // AkaURLStreamHandler(true) that instantiates an object of AkaSURLConnection.
        URL url = new URL(null, uri, new AkaURLStreamHandler(true));
        urlConnection = (HttpsURLConnection) url.openConnection();
```

```
        // set a custom ssl socket factory if needed.
        urlconnection.setSSLSocketFactory(customFactory);

        ...
        // Download content using the InputStream
        InputStream inputStream = new BufferedInputStream(urlConnection.getInputStream());
        ...
        // Close the stream once done with the download.
        inputStream.close();
        } catch (IOException e) {
                e.printStackTrace();
        } finally {
 // Make sure to call HttpsURLConnection#disconnect() to release resources and collect
stats.
                if (urlConnection != null) {
                                urlConnection.disconnect();
                }
        }
```

5.9 Cache-Control request parameters

AkaURLConnection ensures delivery of fresh content. Content is either served from the cache or from the network transparently. In certain cases, it may be desirable to override this behavior. For ex - In case of poor connectivity, a caller may be okay to use stale responses for a particular request(s). Or in certain cases, a caller may want the content that's being served (in case of cached content) to be revalidated by origin server by controlling its expiry or forcing a revalidation in a particular scenario such as a certain time of day.
AkaURLConnection provides following API for this:

```
/**
*
* @param field non-null key
* @param value non-null value<br>
* SDK cache behavior can be controlled with use of following key-value pairs. <br>
*
*          Pragma:no-cache: Forces SDK to revalidate cached response.<br>
*          Cache-Control:no-cache: Same as Pragma:no-cache.<br>
*          Cache-Control:max-age='x': Forces SDK to select expiry for the content as Min('x', expiry calculated from
response headers)<br>
```

```
*          Cache-Control:max-stale='x': If assigned a value, then the client is willing to accept a response that has exceeded its expiration time by no more than the specified number of seconds. If present and no value is assigned to max-stale, then the client is willing to accept a stale response of any age. Developers can use
*          this under poor network conditions to serve stale responses.<br><br>
*
*          Note: max-age/max-stale is ignored if no-cache is present.
*
*
*/
@Override
public void setRequestProperty(String field, String value);
```

6 QUIC Library Integration

MAP sdk has a capability to accelerate requests using chromium's QUIC protocol. If the configuration to enable quic from the portal is turned on, all the requests by default will try to use QUIC and if the server supports QUIC the response is served over QUIC else it will fall back to HTTP. To enable this capability on the app, an additional aar file needs to be included as below

- Download the SDK and unzip it.
- In a file explorer (*not* Android Studio), drag the unzipped akamai-cronet-lib-release-version.aar file into the /app/libs directory in your project's root directory.
- In Android Studio, edit the build.gradle file in the **app** directory (*not the one in the root folder*) and edit the dependencies sub-section to include .AAR file and following library

```
dependencies {
       implementation(name:'akamai-cronet-lib-release-<version>', ext:'aar')
       ...
}
```

If you build infrastructure of your application is not using Java 8, the code below needs to be added to the build.gradle file

```
android {
     compileOptions {
   sourceCompatibility 1.8
   targetCompatibility 1.8
```

```
}
}
```

ABI Management for different architectures

Akamai cronet aar file is around 4.8MB and includes supports two different architectures

- armeabi-v7a
- arm64-v8a

As the size of SDK is always a priority, these architectures can be selectively filtered based on the target sdk version and audience of the host application using abiFilters in the build.gradle file. For instance, just including armeabi-v7a is enough to support majority of the devices,except v64 devices. So based on audience of the host application these architecture can be selectively chosen.

Example : just including “armeabi-v7a” architecture (Recommended for most cases)

```
android{
defaultConfig{ ndk{ abiFilters "armeabi-v7a" } }
}
```

7 BROTLI Library Integration

MAP sdk has a capability to accelerate requests using Brotli compression. Brotli provides a much denser compression of the data when compared to gzip. If the configuration to enable brotli from the portal is turned on, all the requests by default will try to use brotli encoding by including it in the ‘accept-encoding’ header. If the server supports brotli and responds with a brotli encoded stream, it will be decoded by the sdk and provided to the application. In case the requirements are not met, default ‘gzip’ encoding is used. To enable this capability on the app, an additional aar file needs to be included as below

- Download the SDK and unzip it.
- In a file explorer (*not* Android Studio), drag the unzipped akamai-brotli-lib-release-<version>.aar file into the /app/libs directory in your project’s root directory.
- In Android Studio, edit the build.gradle file in the **app** directory (*not the one in the root folder*) and edit the dependencies sub-section to include .AAR file and following library

```
dependencies {
        implementation(name:'akamai-brotli-lib-release-version', ext:'aar')
        ...
}
```

8 mPulse Library Integration

MAP SDK has the capability to send mPulse beacons for real-time performance monitoring. This feature can be enabled through the MAP portal - a mPulse license key(pre-configured on mPulse Dashboard) will be needed. If enabled MAP SDK will initialize mPulse and send beacons to soasta dashboard for real time monitoring. To integrate this feature

- Download the SDK and unzip it.
- In a file explorer (not Android Studio), drag the unzipped mpulse-android-release-<version>.aar file into the /app/libs directory in your project's root directory.
- In Android Studio, edit the build.gradle file in the app directory (not the one in the root folder) and edit the dependencies sub-section to include .aar file and following library

```
dependencies {
        implementation (name:'mpulse-android-release-<version>', ext:'aar')
        ...
}
```

For more information on mPulse integration, refer https://docs.soasta.com/beacon-api/#android. The initialization of mPulse is already taken care by MAP SDK.

MAP SDK can identify the mPulse Custom Metric based on the URL patterns defined by the client and report this to mPulse portal. The client needs to configure the Custom Metric in the mPulse portal and then define the URL patterns associated with the metric in android_sdk_init.xml file (refer section 3.2 for more details about this file). An example URL pattern definition is as below.

```
        <com_akamai_sdk_custom_metric name="Shopping Cart">
            <com_akamai_sdk_url_pattern pattern="http://www.akamai.com/cart/"/>
            <com_akamai_sdk_url_pattern pattern="https://www.akamai.com/cart/"/>
        </com_akamai_sdk_custom_metric>
        <com_akamai_sdk_custom_metric name="Check Out">
            <com_akamai_sdk_url_pattern pattern="http://www.akamai.com/checkout"/>
            <com_akamai_sdk_url_pattern pattern="https://www.akamai.com/checkout"/>
        </com_akamai_sdk_custom_metric>
```

Additionally, MAP SDK also implicitly set mPulse Page View Groups and Custom Timer(predefined in the mPulse portal) when `vocService.startEvent("<Name>")` is called. The client can reset the mPulse View Group and custom Timer on calling `vocService.stopEvent("<Name>")`. All requests within the startEvent and stopEvent will be identified with "<Name>" page view group and a custom timer with name "<Name>") will be triggered.

10 Managing Cookies

With a multitude of third party libraries available for network request and image downloading (such as OkHttp, Picasso, Glide) as well as the combined use of android WebView and HttpUrlConnection, managing cookies becomes an essential part of any app. Please note that if the java.net.CookieManager is already setup in the app, there is nothing else to do and this section may be skipped. This part serves as a general guidance for managing the cookies for MAP-SDK use. The code snippets here are for example only.

- The MAP SDK uses android HttpUrlConnection to send the request. The HttpUrlConnection uses java.net.CookieManager to get and save the cookies for a session. Once the cookie manager is created, the cookies are automatically saved from the response and appended to the subsequent requests (applying all the rules for appending the cookie). Here is what the app can do to create a new cookie manager.

```
CookieManager cookieManager = new CookieManager();
cookieManager.setCookiePolicy(CookiePolicy.ACCEPT_ALL);
CookieHandler.setDefault(cookieManager);
```

This cookieManager will by default use an in memory cookie store. Optionally, the cookie manager can be created with a custom implementation of the Cookie store that would persist the cookies across app restarts, for example.

```
CookieManager cookieManager = new CookieManager(new CustomCookieStore(),
CookiePolicy.ACCEPT_ALL);
```

- The OkHttp client uses a CookieJar. In case the app wants to continue to use the CookieJar, it can be created in the following way to make the cookies also available to the HttpUrlConnection.

```
CookieManager cookieManager = new CookieManager();
cookieManager.setCookiePolicy(CookiePolicy.ACCEPT_ALL);
CookieHandler.setDefault(cookieManager);
CookieJar cookieJar = new JavaNetCookieJar(cookieManager);

dependencies {
       implementation "com.squareup.okhttp3:okhttp-urlconnection:3.10.0"
       ...
}
```

Any cookies now added to cookie jar will also be accessible to the MAP-SDK.

- Syncing cookies between the web views and the httpUrlConnection requests: The cookies for HttpUrlConnection are maintained by java.net.CookieManager while the cookies for Webview are maintained by android.webkit.CookieManager. If the app is using a combination of Webview and Net (HttpUrlConnection) requests, the app needs to manually synchronize the cookies between the two managers.

  Here is a code example to synchronize the cookies from java.net.CookieManager to android.webkit.CookieManager for a request to ‘uri’ after getting a response.

```
CookieHandler handler = CookieHandler.getDefault();
Map<String, List<String>> cookie = handler.get(uri, responseHeaders);
android.webkit.CookieManager.getInstance().setCookie(uri, cookie.get("Cookie").get(i));
```

- Picasso, Volley and Glide: These packages don’t have any specific cookie handling. Once the cookie Manager is created and set as in section 1, the intercepted requests by MAP-SDK will save and append the cookies.

9 Appendix - Requirements and Dependencies

9.1 Background Execution

The MAP SDK downloads any prepositioned content while the application is running . Various factors determine when to start downloading, how much to download, and when to pause

downloads. Influencing factors include the state of the mobile network and the quality state of the provider network.

When your app is in the foreground, downloads are happening without any need for changes to your code. MAP SDK also downloads content(marked for prepositioned) in background triggered by GCM push notifications. There is no additional certificate configuration needed for push notifications to work

9.2 SDK Events

The client app can listen to sdk background events (not normally needed) status by registering a BroadCastReceiver with filter set to "com.akamai.android.sdk.ACTION_VOC_STATUS" .

Example in Client Manifest

```
<application
....
  <receiver android:name={ApplicationId}.MyMapBroadcastReceiver">
        <intent-filter>
            <action android:name="com.akamai.android.sdk.ACTION_VOC_STATUS" />
            <category android:name="{ApplicationId}" />
        </intent-filter>
</receiver>
....</application>
```

The client app can then extend the com.akamai.android.sdk.VocStatusReceiver

API and override methods to listen to different status updates like sdk sync start, cache sync done

```
public class MyMapBroadcastReceiver extends VocStatusReceiver {
   @Override
   protected void onCacheSynchStart(Context context) {
     // Called on any background or foreground sync
    // Here you can set any prepositioning specific parameters
    // if needed VocService.setCustomConnectionParameters ( section 5.6)
   }
}
```

9.3 SDK Debugging

Android: On Android Logs are tagged with AkaSDKLogger for map-sdk

Common Log Lines for tracking activity:

Initialization Logs

06-01 16:59:07.424 22964-22997/example.com.vocaccelerator E/AkaSDKLogger: AnaCacheService: postStatusToServer: not registered

Check akamai_sdk_init.xml existing in res/xml folder and licenseKey is added , and segments subscribed are added in provided within the tags
<**com_akamai_sdk_license_key**>
</**com_akamai_sdk_license_key**>

E/AkaSDKLogger: registerIfNeeded: Couldn't find resource file for meta-data key com.akamai.android.sdk!

Missing akamai_sdk_init in res/xml and reference in the AndroidManifest.xml to the init file

<meta-data

android:name="com.akamai.android.sdk"

android:resource="@xml/akamai_sdk_init" />

Content Logs

Content served from Cache

- D/AkaSDKLogger: D/AkaSDKLogger: AkaURLConnection: Stats: URL: https://www.akamai.com/, Type: CACHE_FETCH_ADHOC, Connection: cellular/LTE, RespCode: 200, ContentLength: 251167, StartTime: 1523471307223, Duration: 55, Ttfb: 6

Content served from Network

- D/AkaSDKLogger: D/AkaSDKLogger: AkaURLConnection: Stats: URL: https://www.akamai.com/, Type: CACHE_MISS, Connection: cellular/LTE, RespCode: 200, ContentLength: 251167, StartTime: 1523471307223, Duration: 55, Ttfb: 6

On receiving Push notification

- 04-11 14:59:36.651 31167-31192/example.com.vocaccelerator D/AkaSDKLogger: AnaCacheService: com.akamai.anaina.PREPARE_SYNC

Upload Analytics

- 06-01 16:35:07.821 19777-19803/example.com.vocaccelerator D/AkaSDKLogger: RestWrapper: sendWebAccAnalytics: 200

Prepositioning triggered

- 06-01 16:46:23.170 21803-21842/example.com.vocaccelerator D/AkaSDKLogger: AnaWebContentDownloader: WebContent: Queued for download 11, policy 0

9.4 Upgrading from a previous SDK version to 20.1.1 or later

The MAP SDK version 20.1.1 and later has been updated to use Firebase Messaging as Google has deprecated GCM. If you are updating from a previous version of SDK to version 20.1.1 or later, please make the following change to the app's build.gradle file. Remove the GCM dependency and add the FCM as suggested below. For additional information on getting background notifications working for prepositioning follow section 4.3

```
useLibrary 'org.apache.http.legacy'

dependencies {
        implementation fileTree(dir: 'libs', include: ['*.jar'])
        implementation 'com.google.android.gms:play-services-gcm:10.2.1'
        implementation 'com.google.firebase:firebase-messaging:17.3.4'
        implementation 'com.android.support:support-v4:26.1.0'
        implementation (name:'map-sdk-version', ext:'aar')
}
```