LASAGNE Android Service Control Investigation

Goals

- Investigate alternative ways of running and controlling a LASAGNE TAFServer on the Android Operating System

Outcomes

- Ability to control (start/stop) the TAFServer without blocking the main/UI thread
- Allow the TAFServer to be shared between application by running as a global service
- Prevent the Operating System from easily killing the service by running it as a foreground service
- Start the TAFServer when the Android Operating System is booted by handling broadcast events on startup
- Understand how further control (e.g. SDDU-like control) can be acheived from an Application by binding the service, using the Intent or Broadcast system

Assumed Knowledge

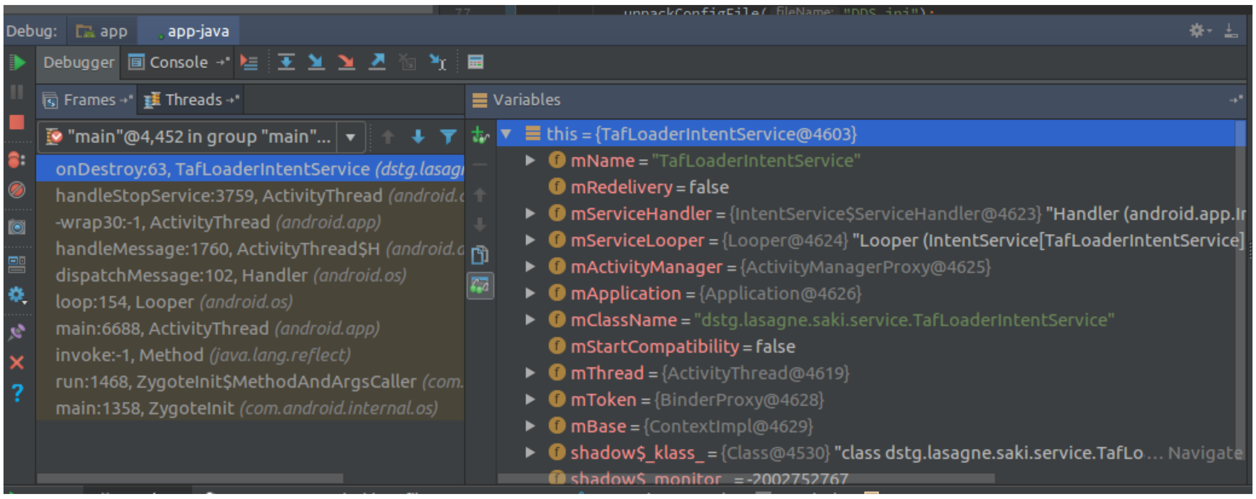
There is a good introduction to **Android Services Fundamentals on pluralsight**.

- Fundamentals
  - See the **Application Fundamentals**
  - Understand Android component types, component activation and the manifest files
  - Less relevant components include the Activities (only basic knowledge required) and Content Providers (not used at all in this work)
  - If you are reading this page, it is assumed you probably already know the fundamentals; you have probably already built and ran an application that uses the **Android NDK** in **Android Studio** with **Gradle** and **CMake**.
  - If you haven't already built and ran an application, you may want to install the Vagrant VM development environment, cross-compile LASAGNE for Android and try building a project first (see **LASAGNE Android Support**)
    - *Note the latest instructions for VM and cross-compilation may have changed from the above page, recommended getting the VM from git and following the readme*
- Services
  - See the **Services Guide**, **Service API Reference** and **Starting Background Services**
  - Understand the service types, managing the service lifecycle and strategies for dealing with the system closing services (e.g. do nothing, restart, etc.)
- Processes / Threads
  - See the **Processes and Threads Guide**, **Process Lifecycle**, **Process API** and **Java Util Concurrency**
  - Understand how the system prioritises processes, allocates components to specific processes, and when it will reclaim memory and destroy processes.
- Broadcasts / Publish-Subscribe
  - See the **Broadcasts Guide** and **Broadcasts Receiver**
  - Understand its use as a publish-subscribe mechanism and the effects on the process state (ability to override priorities and increase the likelihood of a process being killed by the system)
- Activity Lifecycle
  - See the **Activity Lifecycle Guide** and **Activity Lifecycle Operations from the course at Vanderbilt University**
  - Understand when activities can be killed abrubtly (i.e. from the pause state) and when the onDestroy method is called (providing a last chance to cleanup gracefully)
- Service Lifecycle
  - See **Implementing the Service Lifecycle Callbacks**

Preventing Blocking the Main/UI Thread

By default, an **Android Service** runs in the **main thread** of its **hosting process**. Services that perform **instensive work or blocking operations** must be started on a **new thread** otherwise the application will become unresponse.

The LASAGNE TAFServer's public interface run method suggests the concept of enabling startup both **asynchronously** or **synchronously** depending on the value of the boolean **wait_for_completion** parameter. Ideally no new thread needs to be created for the JNI call when wait_for_completion=false, but during investigation it was observed that this resulted in an immediate shutdown. It may not have been due to the TAFServer itself but inconsequently because the service subclassed **IntentService**, which uses a **"work queue processor" pattern** (Active Object) to offload tasks from an applicatons main thread. The IntentService has certain limitations: it can only process one item at a time, queue items will be lost if the service is destroyed and **the service is destroyed when its queue is emptied**. The final limitation was one that was not obvious, as it was expected that the service would remain dormant even when the work queue was empty. The documentation used terms such as the service will be "stopped" rather than destroyed, which is true but handleStopService calls onDestroy (see the image below). It was almost certainly the reason for the TAFServer being immediately shutdown since the onDestroy makes a JNI call which invokes DAF::ShutdownHandler.send_shutdown().



```
"main"@4,452 in group "main"...
onDestroy:63, TafLoaderIntentService (dstg.lasagi
handleStopService:3759, ActivityThread (android.
-wrap30:-1, ActivityThread (android.app)
handleMessage:1760, ActivityThread$H (android.
dispatchMessage:102, Handler (android.os)
loop:154, Looper (android.os)
main:6688, ActivityThread (android.app)
invoke:-1, Method (java.lang.reflect)
run:1468, ZygoteInit$MethodAndArgsCaller (com.
main:1358, ZygoteInit (com.android.internal.os)
```

```
this = {TafLoaderIntentService@4603}
  mName = "TafLoaderIntentService"
  mRedelivery = false
  mServiceHandler = {IntentService$ServiceHandler@4623} "Handler (android.app.I
  mServiceLooper = {Looper@4624} "Looper (IntentService[TafLoaderIntentService]
  mActivityManager = {ActivityManagerProxy@4625}
  mApplication = {Application@4626}
  mClassName = "dstg.lasagne.saki.service.TafLoaderIntentService"
  mStartCompatibility = false
  mThread = {ActivityThread@4619}
  mToken = {BinderProxy@4628}
  mBase = {ContextImpl@4629}
  shadow$_klass_ = {Class@4530} "class dstg.lasagne.saki.service.TafLo... Navigate
  shadow$_monitor_ = -2002752767
```

The following statement in this Stack Overflow "What is the difference between an IntentService and a Service" succinctly describes the different purposes of the IntentService and Service base classes.

> *"In short, a Service is a broader implementation for the developer to set up background operations, while an IntentService is useful for "fire and forget" operations, taking care of background Thread creation and cleanup.* ***Service*** *is an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use.* ***IntentService*** *is a base class for IntentService Services that handle asynchronous requests (expressed as Intents) on demand. Clients send requests through* `startService(Intent)`*calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work."*

There are other reasons to avoid the use of IntentService for most use cases in future. For instance, the IntentService is subject to the **Background Execution Limits** imposed in Android 8.0+ and in most cases the **JobIntentService** will provide richer functionality (note we are still using Android 7.0 though so the JobIntentService is something to investigate in future). It became clear that on Android 7.0 we need to subclass an ordinary Service but handle offloading of tasks from the main thread to worker threads in some other way. We tried using were **pthreads** in the native c++ code, **AsyncTasks** and finally Java's **java.util.concurrent** package components. Pthreads were only really used as a quick interim solution as they are not portable. AsyncTasks were promising because they publish progress back to the UI Thread, however it turns out that they are also not suited to long lived operations. The most portable and durable solution was to use the facilities provided by the **java.util.concurrent** package, specifically an **ExecutorService** interface with a **CachedThreadPool** implementation**.** Broken down to the most simple parts (other aspects such as logging removed), the code looks like this:

```
public class TafControllerService extends Service {
 // ...

 // Executor to run intent messages asynchronously
 private Executor mTaskExecutor = Executors.newCachedThreadPool();

 // JNI native binding method, internally starts TAFServer and blocks
    protected native int startTafServer();

 // JNI native binding method, internally DAF::ShutdownHandler.send_shutdown message
```

```
    protected native int shutdownTafServer();

 // Called from main activity context startService(...)
    @Override
    public int onStartCommand(final Intent intent, final int flags, final int startId) {
  mTaskExecutor.execute(() -> {
   // Start TAFServer (blocks until shutdown)
   final int returnValue = startTafServer();
   // ... handle return code here then kill the process to prevent SIGSEGV...
   Process.killProcess(Process.myPid());
  }
  // return flag to restart app if it is killed by the OS
  return START_STICKY;
 }

 // Called from main activity context stopService(...)
    @Override
    public void onDestroy() {
        mTaskExecutor.execute(() -> {
            final int returnValue = shutdownTafServer();
            // ... handle return code here...
        });
        super.onDestroy();
    }

 // ...
}
```

On a final note, after the realisation that it was the IntentService that may have been causing the immediate shutdown of the TAFServer, we may be able to try setting wait_for_completion=false in the native code to start the server without making the JNI call in a new thread. It's questionable whether the DAF::ShutdownHandler.send_shutdown() call needs to be done on a seperate thread. However, there may be situations in the future where we want to offload other SDDU-like commands onto worker threads. For that purpose the java **Executor** is still a useful tool, especially on Android 7 (which does not have a **JobIntentScheduler**).

Running as a Shared Global Process

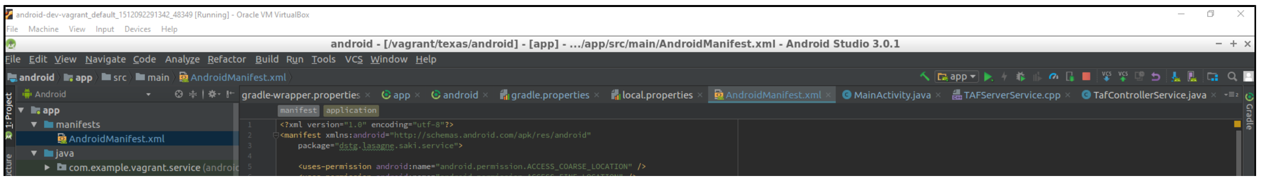
By default, Android Services are started in the same process as the application or context that starts the service. In some cases it is perfectly acceptable to have the TAFServer run in the application and shutdown when the application is closed. For various reasons such as keeping the footprint small, it may be desirable to run only a single TAFServer on android and allow any application to use a single TAFServer. The service controlling the lifecycle of the TAFServer can be declared to start in a seperate process or globally through the **android:process** attribute of the service element manifest file. As stated in the documentatio for the **Service Element**:

> "...If the name assigned to this attribute begins with a colon (':'), a new process, private to the application, is created when it's needed and the service runs in that process. If the process name begins with a lowercase character, the service will run in a global process of that name, provided that it has permission to do so. This allows components in different applications to share a process, reducing resource usage."

Simply providing a name starting with a lower case letter results in the service running in a different process to the application that starts it. This extract, directly from the manifest file demonstrates that addition of the android:process attribution with the service name starting with a lowercase letter:

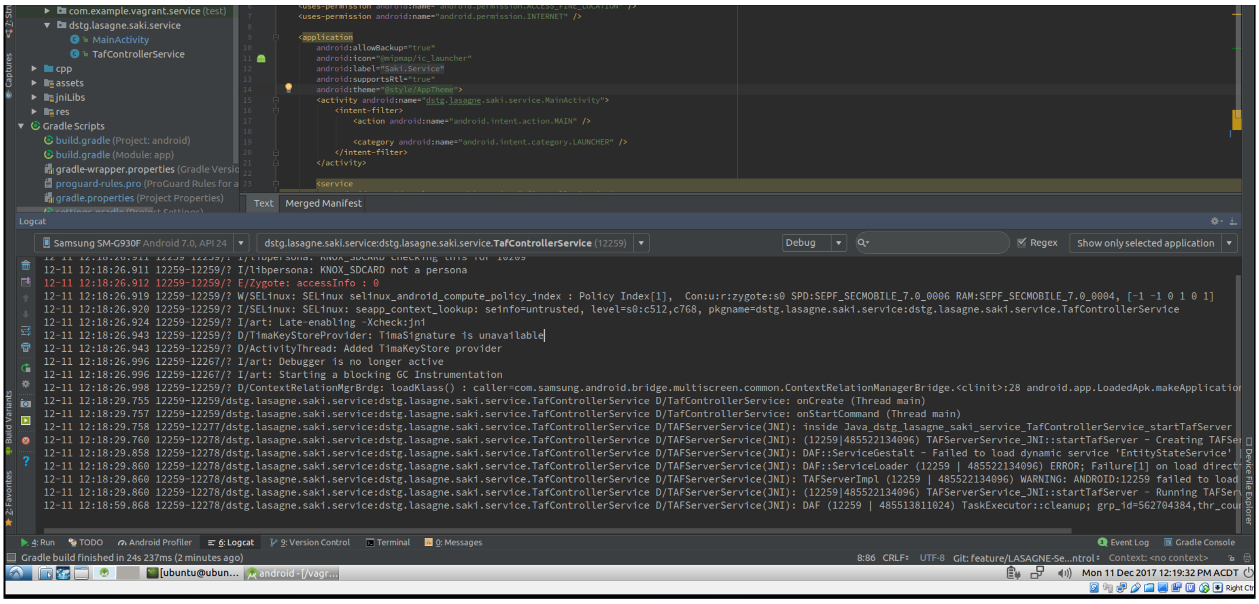
```
        <service
            android:name="dstg.lasagne.saki.service.TafControllerService"
            android:exported="true"
            android:process="dstg.lasagne.saki.service.TafControllerService"></service>
```

The following screenshots show the console logs when running the application in debug mode in Android Studio with the entry above in the manifest:


android-dev-vagrant_default_1512092291342_48349 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
android - [/vagrant/texas/android] - [app] - .../app/src/main/AndroidManifest.xml - Android Studio 3.0.1
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
android app src main AndroidManifest.xml
Android
gradle-wrapper.properties app android gradle.properties local.properties AndroidManifest.xml MainActivity.java TAFServerService.cpp TafControllerService.java
app
manifests
AndroidManifest.xml
java
com.example.vagrant.service (androi
manifest application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="dstg.lasagne.saki.service">

    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```



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

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="Saki.Service"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name="dstg.lasagne.saki.service.MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <service
```

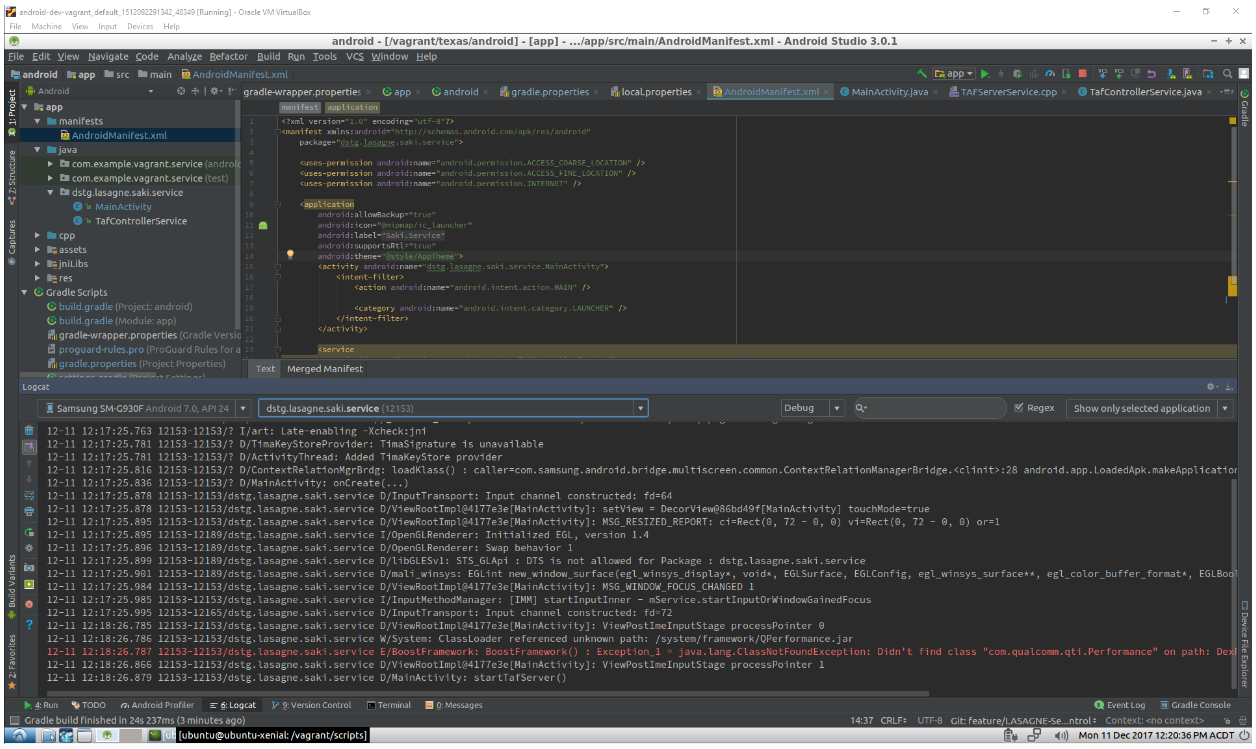
Logcat: Samsung SM-G930F Android 7.0, API 24 | dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService (12259) | Debug | Regex | Show only selected application

```
12-11 12:18:26.911 12259-12259/? I/libpersona: KNOX_SDCARD checking this for 10269
12-11 12:18:26.911 12259-12259/? I/libpersona: KNOX_SDCARD not a persona
12-11 12:18:26.912 12259-12259/? E/Zygote: accessInfo : 0
12-11 12:18:26.919 12259-12259/? W/SELinux: SELinux selinux_android_compute_policy_index : Policy Index[1],  Con:u:r:zygote:s0 SPD:SEPF_SECMOBILE_7.0_0006 RAM:SEPF_SECMOBILE_7.0_0004, [-1 -1 0 1 0 1]
12-11 12:18:26.920 12259-12259/? I/SELinux: SELinux: seapp_context_lookup: seinfo=untrusted, level=s0:c512,c768, pkgname=dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService
12-11 12:18:26.924 12259-12259/? I/art: Late-enabling -Xcheck:jni
12-11 12:18:26.943 12259-12259/? D/TimaKeyStoreProvider: TimaSignature is unavailable
12-11 12:18:26.943 12259-12259/? D/ActivityThread: Added TimaKeyStore provider
12-11 12:18:26.996 12259-12267/? I/art: Debugger is no longer active
12-11 12:18:26.996 12259-12267/? I/art: Starting a blocking GC Instrumentation
12-11 12:18:26.998 12259-12259/? D/ContextRelationMgrBrdg: loadKlass() : caller=com.samsung.android.bridge.multiscreen.common.ContextRelationManagerBridge.<clinit>:28 android.app.LoadedApk.makeApplicatio
12-11 12:18:29.755 12259-12259/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TafControllerService: onCreate (Thread main)
12-11 12:18:29.757 12259-12259/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TafControllerService: onStartCommand (Thread main)
12-11 12:18:29.758 12259-12277/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): inside Java_dstg_lasagne_saki_service_TafControllerService_startTafServer
12-11 12:18:29.760 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): (12259|485522134096) TAFServerService_JNI::startTafServer - Creating TAFSe
12-11 12:18:29.858 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): DAF::ServiceGestalt - Failed to load dynamic service 'EntityStateService'
12-11 12:18:29.860 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): DAF::ServiceLoader (12259 | 485522134096) ERROR; Failure[1] on load direct
12-11 12:18:29.860 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): TAFServerImpl (12259 | 485522134096) WARNING: ANDROID:12259 failed to load
12-11 12:18:29.860 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): (12259|485522134096) TAFServerService_JNI::startTafServer - Running TAFSer
12-11 12:18:59.868 12259-12278/dstg.lasagne.saki.service:dstg.lasagne.saki.service.TafControllerService D/TAFServerService(JNI): DAF (12259 | 485513811024) TaskExecutor::cleanup; grp_id=562704384,thr_cou
```

Gradle build finished in 24s 237ms (2 minutes ago)

8:86 CRLF UTF-8 Git: feature/LASAGNE-Se...ntrol Context: <no context>

Mon 11 Dec 2017 12:19:32 PM ACDT



android - [/vagrant/texas/android] - [app] - .../app/src/main/AndroidManifest.xml - Android Studio 3.0.1

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="dstg.lasagne.saki.service">

    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Saki.Service"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name="dstg.lasagne.saki.service.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
```

Logcat: Samsung SM-G930F Android 7.0, API 24 | dstg.lasagne.saki.service (12153) | Debug | Regex | Show only selected application

```
12-11 12:17:25.763 12153-12153/? I/art: Late-enabling -Xcheck:jni
12-11 12:17:25.781 12153-12153/? D/TimaKeyStoreProvider: TimaSignature is unavailable
12-11 12:17:25.781 12153-12153/? D/ActivityThread: Added TimaKeyStore provider
12-11 12:17:25.816 12153-12153/? D/ContextRelationMgrBrdg: loadKlass() : caller=com.samsung.android.bridge.multiscreen.common.ContextRelationManagerBridge.<clinit>:28 android.app.LoadedApk.makeApplicatio
12-11 12:17:25.836 12153-12153/? D/MainActivity: onCreate(...)
12-11 12:17:25.878 12153-12153/dstg.lasagne.saki.service D/InputTransport: Input channel constructed: fd=64
12-11 12:17:25.878 12153-12153/dstg.lasagne.saki.service D/ViewRootImpl@4177e3e[MainActivity]: setView = DecorView@86bd49f[MainActivity] touchMode=true
12-11 12:17:25.895 12153-12153/dstg.lasagne.saki.service D/ViewRootImpl@4177e3e[MainActivity]: MSG_RESIZED_REPORT: ci=Rect(0, 72 - 0, 0) vi=Rect(0, 72 - 0, 0) or=1
12-11 12:17:25.895 12153-12189/dstg.lasagne.saki.service I/OpenGLRenderer: Initialized EGL, version 1.4
12-11 12:17:25.896 12153-12189/dstg.lasagne.saki.service D/OpenGLRenderer: Swap behavior 1
12-11 12:17:25.899 12153-12189/dstg.lasagne.saki.service D/libGLESv1: STS_GLApi : DTS is not allowed for Package : dstg.lasagne.saki.service
12-11 12:17:25.901 12153-12189/dstg.lasagne.saki.service D/mali_winsys: EGLint new_window_surface(egl_winsys_display*, void*, EGLSurface, EGLConfig, egl_winsys_surface**, egl_color_buffer_format*, EGLBool
12-11 12:17:25.984 12153-12153/dstg.lasagne.saki.service D/ViewRootImpl@4177e3e[MainActivity]: MSG_WINDOW_FOCUS_CHANGED 1
12-11 12:17:25.985 12153-12153/dstg.lasagne.saki.service I/InputMethodManager: [IMM] startInputInner - mService.startInputOrWindowGainedFocus
12-11 12:17:25.995 12153-12165/dstg.lasagne.saki.service D/InputTransport: Input channel constructed: fd=72
12-11 12:18:26.785 12153-12153/dstg.lasagne.saki.service D/ViewRootImpl@4177e3e[MainActivity]: ViewPostImeInputStage processPointer 0
12-11 12:18:26.786 12153-12153/dstg.lasagne.saki.service W/System: ClassLoader referenced unknown path: /system/framework/QPerformance.jar
12-11 12:18:26.787 12153-12153/dstg.lasagne.saki.service E/BoostFramework: BoostFramework() : Exception_1 = java.lang.ClassNotFoundException: Didn't find class "com.qualcomm.qti.Performance" on path: Dex
12-11 12:18:26.866 12153-12153/dstg.lasagne.saki.service D/ViewRootImpl@4177e3e[MainActivity]: ViewPostImeInputStage processPointer 1
12-11 12:18:26.879 12153-12153/dstg.lasagne.saki.service D/MainActivity: startTafServer()
```

Gradle build finished in 24s 237ms (3 minutes ago)

14:37 CRLF UTF-8 Git: feature/LASAGNE-Se...ntrol Context: <no context>

[ubuntu@ubuntu-xenial: /vagrant/scripts]

Mon 11 Dec 2017 12:20:36 PM ACDT

Keeping the Service Process Alive

Android has several mechanisms to prioritise and kill off services based on the current state of the system. Background services (which are the default) are more suceptable than foreground services to being shutdown. Fortunately, the return value of the service's onStartCommand provides the Android Operating System with a strategy for handling this situation. The flags **START_STICKY**, **START_NOT_STICKY**, and **START_REDELIVER_INTENT** indicate if the OS shoud immediately restart the service, or not depending on the particular value (See the Android **Service Component Guide** and **Service API documentation** for more information). In a case where LASAGNE is not bound to a particular application and shared, we want to be able to go further than this and do our best to keep the service's process alive (segway to Android **Processes and Threads** and **Process Lifecycle**). It is possible to change the process type from background to foreground to raise the importance of the process and make it harder for Android to kill the service. If you do not do this, every time you close the application that started the global service will result in the service itself being restarted. Once the service type is set to foreground it will only be susceptible to be killed as an absolute last resort in low memory state. To quote the Android process lifecycle page:

"...There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive..."

The code to raise the process priority is shown below. The name of the startForeground method can be confusing as it needs to be called after startService. For this reason, it is placed in onCreate to ensure it is always called. A notification is provided as a parameter to inform the user that the service is running (assumedly to allow users to kill foreground processes that are monopolising resources).

```
public class TafControllerService extends Service {
    // ...
   @Override
    public void onCreate() {
        //...
  // Caution: The integer ID that you give to startForeground() must not be 0.
        startForeground(1, new NotificationCompat.Builder(this)
                .setContentTitle("TAFControllerService")
                .setContentText("Moving service to foreground")
                .setContentIntent(PendingIntent.getActivity(this, 0,
                        new Intent(this, MainActivity.class), 0)).build());
  //...
   }
   //...
}
```

It is worth noting that once a project is promoted to the foreground a notification is displayed to indicate this as described in Android documentation's "**Running a service in the foreground**":

"A foreground service is a service that the user is actively aware of and is not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the *Ongoing* heading. This means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground."

You can check if a service is running in the foregroundwith **dumpsys** by running the following command in a terminal:

```
adb shell dumpsys activity processes
```

You will find a record for the process:

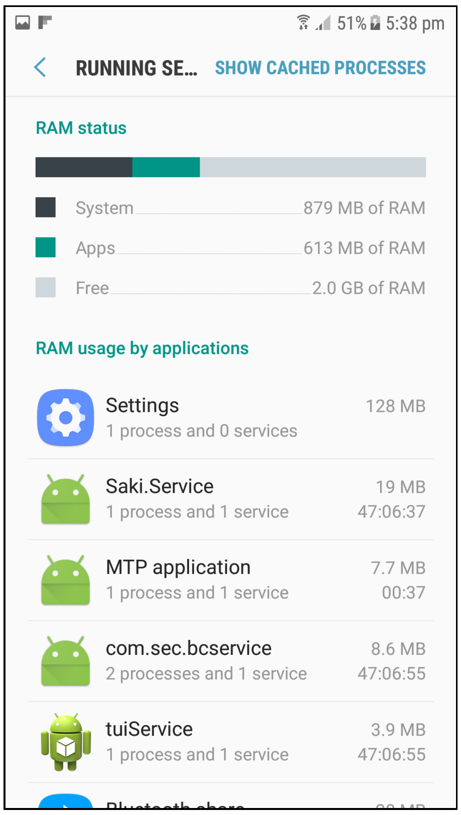
```
*APP* UID 10210 ProcessRecord{7697d71 14317:dstg.lasagne.saki.service/u0a210}
  user #0 uid=10210 gids={50210, 9997, 3003}
  requiredAbi=arm64-v8a instructionSet=arm64
  dir=/data/app/dstg.lasagne.saki.service-2/base.apk publicDir=/data/app/dstg.lasagne.saki.servic
  packageList={dstg.lasagne.saki.service}
  compat={480dpi}
  thread=android.app.ApplicationThreadProxy@2890756
  pid=14317 starting=false
  lastActivityTime=-1m32s538ms lastPssTime=-17s913ms nextPssTime=+29m42s25ms
  adjSeq=92639 lruSeq=0 lastPss=20MB lastSwapPss=215KB lastCachedPss=20MB lastCachedSwapPss=215KB
  cached=true empty=true
  oom: max=1001 curRaw=900 setRaw=900 cur=900 set=900
  curSchedGroup=0 setSchedGroup=0 systemNoUi=false trimMemoryLevel=0
  curProcState=14 repProcState=14 pssProcState=14 setProcState=14 lastStateTime=-48s497ms
  hasShownUi=true pendingUiClean=false hasAboveClient=false treatLikeActivity=false
  lastWakeTime=0 timeUsed=0
  lastCpuTime=0 timeUsed=0
  lastRequestedGc=-4m1s139ms lastLowMemory=-4m1s139ms reportLowMemory=false
  Activities:
    - ActivityRecord{f81002ad0 u0 dstg.lasagne.saki.service/.MainActivity t67}
  Connected Providers:
    - 32129e8/com.android.providers.settings/.SettingsProvider->14317:dstg.lasagne.saki.service/u
```

The LRU list will also show an item for the process:

```
Proc #48: prcp  F/S/SF trm: 0 14981:dstg.lasagne.saki.service.TafControllerService/u0a210 (fg-ser
```

There is a useful discussion on stack overflow called "Foreground service being killed by android" whereby some people have had issues with foreground services being killed and answers that discuss some of the steps required to check the priority in detail above. This discussion was particularly useful to help determine what situations a service may be downgraded from foreground, perhaps due to bugs such as Android Issue 36967794 Foreground service killed when receiving broadcast after activity swiped away in task list. We actually tried to reproduce issue 36967794 but inspection of the processes through dumpsys showed that the process was still running in the foreground. Inspection of the Android code from version 4.2 to 7.0 showed vast changes which is why some of those issues may now be obselete. It is still worth noting that the Android documentation itself mentions that receiving broadcasts can change the priority of a service and that the Operating System always has the right to kill a process (but it should only occur in very low memory situations when set to foreground).

The following screenshots show the service running for over 47 hours before stopping it. Note that the RAM status was relatively low so we have not tried doing this will many applications running yet.

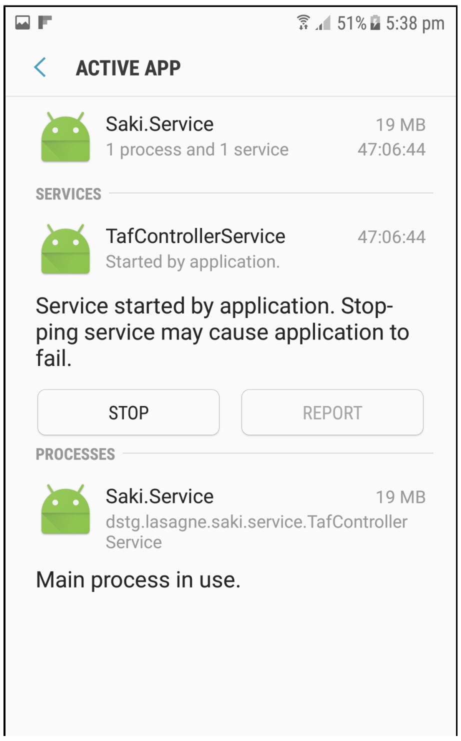


RAM status

■	System	879 MB of RAM
🟩	Apps	613 MB of RAM
□	Free	2.0 GB of RAM

RAM usage by applications

Settings	1 process and 0 services	128 MB
Saki.Service	1 process and 1 service	19 MB 47:06:37
MTP application	1 process and 1 service	7.7 MB 00:37
com.sec.bcservice	2 processes and 1 service	8.6 MB 47:06:55
tuiService	1 process and 1 service	3.9 MB 47:06:55



Saki.Service 19 MB
1 process and 1 service 47:06:44

SERVICES

TafControllerService 47:06:44
Started by application.

Service started by application. Stopping service may cause application to fail.

PROCESSES

Saki.Service 19 MB
dstg.lasagne.saki.service.TafControllerService

Main process in use.

Starting the Service on System Boot

The Android system sends broadcasts when various system events occur, for instance when the system boots up. We can register to handle the system bootup event and use this mechanism to start the TAFServer immediately on bootup. The following is required to add a handler /receiver for the boot event:

1. Add uses-permission entry to the manifest file
2. Add receiver entry to the manifest file, specifying the class which will handle the intents in the intent-filter
3. Create a class which subclasses BroadcastReceiver to handle the event and start the TAFServerService on a worker thread.

Manifest:

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />

     <receiver
            android:name="dstg.lasagne.saki.service.MyBroadcastReceiver"
            android:exported="true"
            android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
                <!-- Apparently HTC devices need QUICKBOOT_POWERON -->
                <action android:name="android.intent.action.QUICKBOOT_POWERON" />
            </intent-filter>
        </receiver>
```

BroadcastReceiver:

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    private static final String TAG = "MyBroadcastReceiver";

    @Override
    public void onReceive(final Context context, final Intent intent) {
        final PendingResult pendingResult = goAsync();
        AsyncTask<String, Integer, String> asyncTask = new AsyncTask<String, Integer, String>() {
            @Override
            protected String doInBackground(String... params) {
                // Start the TAFServer service
                Intent intent = new Intent(context, TafControllerService.class);
                context.startService(intent);

                // Start the TAFServer GUI
                intent = new Intent(context, MainActivity.class);
                intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                context.startActivity(intent);

                // Must call finish() so the BroadcastReceiver can be recycled.
                pendingResult.finish();
                return log;
            }
        };
        asyncTask.execute();
    }
}
```

Process cleanup

In this investigation we were encountering an segmentation fault error when trying to run the TAFServer after it had been previously been stopped. This problem is apparent in the application because and the log but the most information about it can be gathered from inspecting the output of *adb shell dumpsys activity processes:*

```
Bad processes:
  Bad process dstg.lasagne.saki.service uid 10055: crashed at time 264831
    Short msg: Native crash
    Long msg: Native crash: Segmentation fault
    Stack:
      *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
      Build fingerprint: 'samsung/heroltexx/herolte:7.0/NRD90M/G930FXXU1DQBH:user/release-keys'
      Revision: '8'
      ABI: 'arm64'
      pid: 11191, tid: 11232, name: pool-1-thread-2  >>> dstg.lasagne.saki.service:taf <<<
      signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0x6f72504641544d
          x0   000000710bc2297f  x1   000000710bf15d88  x2   000000710a71f9e8  x3   0000000000000
          x4   706f72504641542d  x5   fffffffffffffffd  x6   0000000000808000  x7   7affff514b4f7
          x8   7f7f7f7f7f7f7fff  x9   feff01fcd49b0a1d  x10  7f7f7fffffff7f7f  x11  0101010101010
          x12  0000000000000028  x13  0000000000000000  x14  ffffffffffffffff  x15  000000712dde9
          x16  0000007121e6b418  x17  0000007121c9b8e4  x18  000000712d1243f4  x19  000000710a71f
          x20  000000710a71feb0  x21  000000712181a480  x22  0000000000000001  x23  000000710a71f
          x24  000000710a71f9e0  x25  000000710a71f9e8  x26  000000710bbc96a4  x27  0000000000000
          x28  000000710bc3fb1a  x29  000000710a71f930  x30  0000007121e00fcc
          sp   000000710a71f930  pc   0000007121e00ff8  pstate 0000000060000000

      backtrace:
          #00 pc 0000000000108ff8  /data/app/dstg.lasagne.saki.service-1/lib/arm64/libACE.so (__g
          #01 pc 00000000000d6f1c  /data/app/dstg.lasagne.saki.service-1/lib/arm64/libgnustl_shar
          #02 pc 0000000000109324  /data/app/dstg.lasagne.saki.service-1/lib/arm64/libACE.so (__c
          #03 pc 000000000006b8f8  /data/app/dstg.lasagne.saki.service-1/lib/arm64/libTAF.so (_ZN
          #04 pc 000000000000e59c  /data/app/dstg.lasagne.saki.service-1/lib/arm64/libTAFServerSe
          #05 pc 0000000000256760  /data/app/dstg.lasagne.saki.service-1/oat/arm64/base.odex (off
```

It is widely suggested online for example on stack overflow threads (e.g. Invalid heap address and fatal signal 11) that this issue occurs from a function being called from two different threads at the same time (i.e. synchronisation issues). We didn't get time to investigate this in great detail but perhaps unloading the libraries and clearing the ACE singletons before trying to run the TAFServer in the same process may have helped (perhaps a test should be written to try running LASAGNE, shutting down and restarting from an executable on another operating system to see if the same problem occurs). Given the limited time to investigate, we worked around this by ensuring that the process the service is running in is always killed in the onDestroy method so a new process will be used when the TAFServer is restarted. This was acheived with the following statement directly after the blocking call to run the TAFServer:

```
Process.killProcess(Process.myPid());
```

Bound Services

It is stated when to create a bound service in the documentation for creating bound services:

> *"Create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications through interprocess communication (IPC)."*

At the present moment, we have only got to a point of starting and stopping the TafService using the Android Service. However, in future we may want to be able to allow control of the TafServer (similar to SDDU). We will need to be able expose further TafServer functionality to the activities that use it. We've already investigated the IntentService and determined that it is inappropriate for receiving commands since the service will be destroyed when there are no more intents in the queue. The JobIntentService may be another option if it can be kept running in the foreground as the longer it remains idle, the more likely the Operating System will kill it. In anycase, a bound service may provide advantages over these other services particularly if we have any intention of sharing the TafServer between applications. Bound services will be destroyed when there is no clients bound to the service. The service should still be setup to run in the foreground to prevent tear down of the TAFServer when it is idle (unless requirements dicate a different approach).

A bound service can be created in three ways:

1. **Extending the binder class** (in-process binding only);
2. Using a **messenger** (work accross processes, single threaded); or
3. Using **AIDL** (work accross processes, multi-threaded).

We first started trying the *binder class* approach but abandoned the code since the limitations of in-process only binding are not useful for quite a few of our use cases. We instead tried the *messenger* approach as shown in the following code fragments:

The service:

```
// ...

public class TafService extends Service {

    private static final String CLASS_NAME = TafService.class.getSimpleName();
    private static final String TAG = CLASS_NAME;

    // ------------------------------------------------------------------------------------



    /** Command to the service to start TafServer */
    static final int MSG_START = 1;

    /** Command to the service to shutdown TafServer */
    static final int MSG_SHUTDOWN = 2;

    /** Key to a bundle's server arguments */
    static final String BUNDLE_DATA_SERVER_ARGS = "DATA_SERVER_ARGS";

    /** Key to a bundle's server config files */
    static final String BUNDLE_DATA_SERVER_CONFIG_FILES = "DATA_SERVER_CONFIG_FILES";

    /**
     * Handler of incoming messages from clients.
     */
    class IncomingHandler extends Handler {
        @Override
        public void handleMessage(final Message msg) {
            switch (msg.what) {
            case MSG_START:
                final Bundle bundle = msg.getData();
                final ArrayList<String> args = bundle.getStringArrayList(BUNDLE_DATA_SERVER_ARGS)
                final ArrayList<String> files = bundle.getStringArrayList(BUNDLE_DATA_SERVER_CONF
                handle_start(args, files);
                break;
            case MSG_SHUTDOWN:
                handle_shutdown();
                break;
            default:
                super.handleMessage(msg);
            }
        }
    }

    // ------------------------------------------------------------------------------------

    /**
     * Target we publish for clients to send messages to IncomingHandler.
     */
    private final Messenger mMessenger = new Messenger(new IncomingHandler());

    @Override
    public void onCreate() {
        Log.d(TAG, "onCreate()");

        startForeground(1, new NotificationCompat.Builder(this).setContentTitle(CLASS_NAME)
                .setContentText("Moving service to foreground").build());

        super.onCreate();
    }

    @Nullable
    @Override
    public IBinder onBind(final Intent intent) {
        Log.d(TAG, "onBind(...)");
        return mMessenger.getBinder();
    }

    @Override
    public void onDestroy() {
        Log.d(TAG, "onDestroy()");

        handle_shutdown();

        super.onDestroy();
    }

    // ----------------------------------------------------------------------------------------------

    private void handle_start(final ArrayList<String> args, final ArrayList<String> files) {
        Log.d(TAG, "handle_start()");

        mTaskExecutor.execute(new Runnable() {
            @Override
            public void run() {

                /*
                 * Move the config files from the assets "www" folder to the "files" directory.
                 * This is done so that the config files can be found by the native libraries.
                 */
                new ConfigFilesHandler(getApplicationContext(), files).handle();

                Log.d(TAG, "Starting TafServer...");
                final int returnValue = TafJni.startTafServer(args.toArray(new String[args.size()
                Log.d(TAG, "startTafServer() returned: " + returnValue);

                /*
                 * once complete, crudely kill the process not a recommended approach but re-usin
                 * this process will result in SIGSEGV fault
                 */
                Process.killProcess(Process.myPid());
            }
        });
    }

    private void handle_shutdown() {
        Log.d(TAG, "handle_shutdown()");

        // Is there anyway to get a call back when shutdown??
        Log.d(TAG, "Shutting down TafServer...");
        final int returnValue = TafJni.shutdownTafServer();
        Log.d(TAG, "shutdownTafServer() returned: " + returnValue);
    }
}
```

The JNI interface (note the libs were determined by using *objdump* to see which libs are required by the TAF lib. All the libs were cross-compiled for arm-v8a)

```
// ...
public class TafJni {

    private static final String CLASS_NAME = TafJni.class.getSimpleName();
    private static final String TAG = CLASS_NAME;

    // ----------------------------------------------------------------------------------------------

    static {

        final String[] libs = {
            "ACE",
            "TAO",
            "TAO_AnyTypeCode",
            "TAO_PortableServer",
            "TAO_IORTable",
            "DAF",
            "TAO_CodecFactory",
            "TAO_PI",
            "TAO_Strategies",
            "TAO_Svc_Utils",
            "TAO_CosNaming",
            "TAF",
            "taf-jni"
        };

        for (String lib : libs) {
            Log.d(TAG, "Load Native Libraries - " + lib);
            System.loadLibrary(lib);
        }

    }

    // ------------------------------------------------------------------------------------------

    protected static native int startTafServer(String[] args);
    protected static native int shutdownTafServer();

}
```

The Client:

```
//...

public class Taf extends CordovaPlugin {

    private static final String CLASS_NAME = Taf.class.getSimpleName();
    private static final String TAG = CLASS_NAME;

    // ------------------------------------------------------------------------------------------

    /** Messenger for communicating with the service. */
    private Messenger mService = null;

    /** Flag indicating whether we have called bind on the service. */
    private boolean mBound;

    private JSONArray mData;

    // ------------------------------------------------------------------------------------------

    /**
     * Class for interacting with the main interface of the service.
     */
    private ServiceConnection mConnection = new ServiceConnection() {

        /*
         * Called when the connection with the service has been established, giving us the object
         * can use to interact with the service. e are communicating with the service using a Mes
         * so here we get a client-side representation of that from the raw IBinder object.
         */
        public void onServiceConnected(ComponentName className, IBinder service) {
            Log.d(TAG, "onServiceConnected(...)");
            mService = new Messenger(service);
            mBound = true;
            handle_start();
        }

        // Called when connection with service unexpectedly disconnected -- that is, its process
        public void onServiceDisconnected(final ComponentName className) {
            Log.d(TAG, "onServiceDisconnected(...)");
            mService = null;
            mBound = false;
        }
    };

    // ---------------------------------------------------------------------------------------------

    @Override
    public boolean execute(String action, JSONArray data, CallbackContext callbackContext) throws
        Log.d(TAG, "execute(...)");

        mData = data; // TODO Temporary - may be overridden by subsequent calls

        if ("start".equals(action))
            return handle_bind();
        if ("shutdown".equals(action))
            return handle_unbind();

        return false;
    }

    /**
     * The final call you receive before your activity is destroyed.
     */
    @Override
    public void onDestroy() {
        Log.d(TAG, "onDestroy()");
        handle_unbind();
        super.onDestroy();
    };

    // ... other lifecycle callbacks

    // ---------------------------------------------------------------------------------------------

    private boolean handle_bind() {
        Log.d(TAG, "handle_bind()");

        final Intent intent = new Intent(cordova.getActivity(), TafService.class);
        cordova.getActivity().bindService(intent, mConnection, Context.BIND_AUTO_CREATE);

        return true;
    }

    private boolean handle_unbind() {
        Log.d(TAG, "handle_unbind()");

        if (mBound) {
            handle_shutdown();
            cordova.getActivity().unbindService(mConnection);
            mBound = false;
        }
        return true;
    }


}
```

A bound service can be created in three ways: **extending the binder class** (in-process binding only), using a **messenger** (work accross processes, single threaded) or using **AIDL** (work accross processes, multi-threaded). If the TAFServer service is intended to be contained within the application process, we can use the binder class approach. If we intend to share the TAFServer, we must use messenger or AIDL approach.

We must be mindful that binding a service can change the priority of the service and the process it runs in. For instance, binding with the flag Context.BIND_ABOVE_CLIENT runs the risk of the service being downgraded at least on older versions of Android (see Stack Overflow discussion: Foreground service being killed by android).

Ensuring Graceful Shutdown of the TafServer when forcefully stopped

An service can be killed abruptly in several ways depending on its visibility settings:

1. Any service can be stopped from the "Running Services" menu;
2. Foreground services can be stopped from selecting the item in the notifications menu; and
3. Non-global services can be stopped when the application is closed (i.e. swiped away) or if it is bound, all applications that are bound to it are closed.

This behaviour was first observed and noted when working on the TEXAS RISE project whereby the service is declared in the manifest like so:

```
<service
  android:name="dst.cordova.lasagne.plugin.TafService"
  android:exported="true"
  android:process=":dst.cordova.lasagne.plugin.TafService"></service>
```

Since the name assigned to the android:process attribute beings with a colon, it indicates that the service is started in a new process that is private to the application. We experimented with starting the service using both Intents (unbound) and Messages (bound). In both approaches "swiping away" the application resulted in forcefully killing the application. Understanding the lifecycle of the Application and the order in which methods of the **CordovaPlugin** class are called were required to ensure that when the service is killed, the TafServer has a chance to shutdown gracefully. The important method was the **CordovaPlugin**'s onDestroy method which is the final call received before the application is destroyed. It is in this onDestroy method that a call must be made to stop the service as in this example for an unbound service:

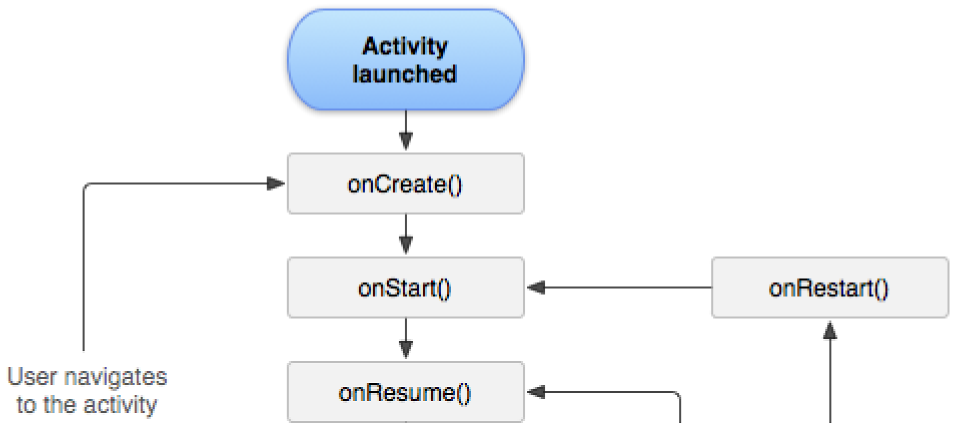
```
/**
 * The final call you receive before your activity is destroyed.
 */
@Override
public void onDestroy() {
    Log.d(TAG, "onDestroy()");
    final Intent intent = new Intent(cordova.getActivity(), TafService.class);
    cordova.getActivity().stopService(intent);
    super.onDestroy();
};
```

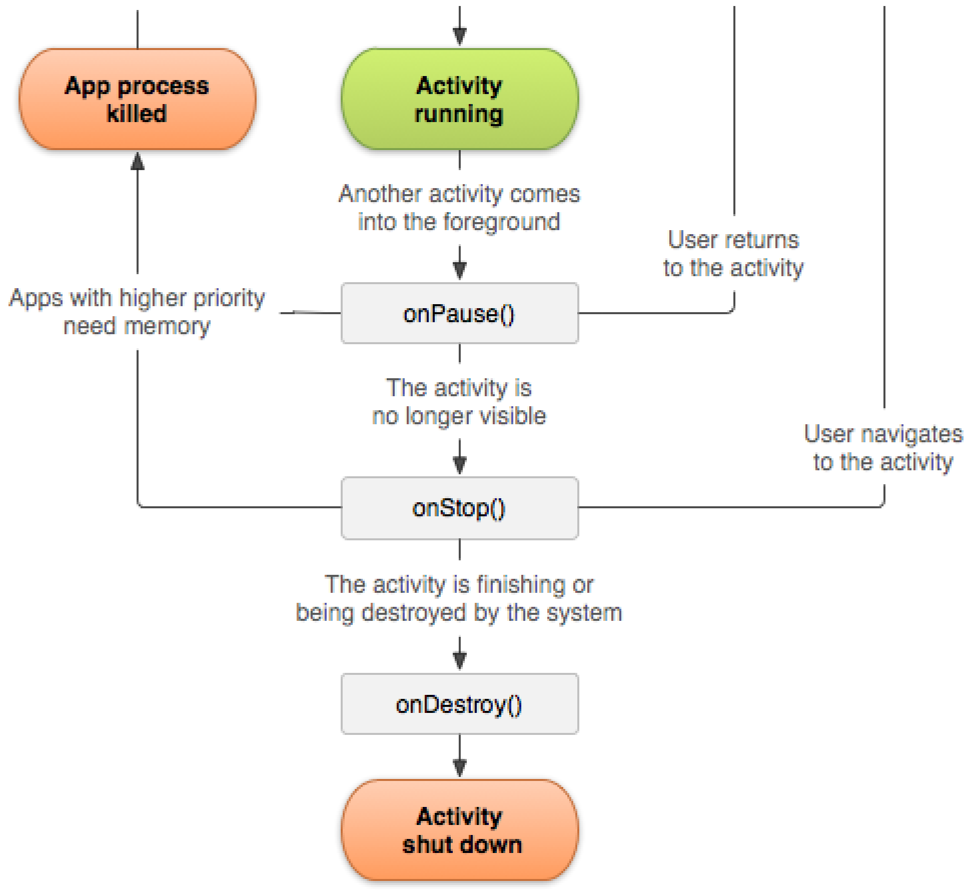
Of course, once onDestroy has finished, the service can be terminated at any time if it is a background thread. Therefore we really need to employ a strategy to ensure that the TafServer can close itself out first. Two approaches may be:

1. Ensure the service priority is set to the foreground, this may even require setting the priority to foreground in the service's onDestroy method (if it was not already running in the foreground); or
2. Use wait and notify to hold off returning from onDestroy until the TafServer is shutdown (we haven't tried this but it should work since the shutdown will result in the call *startTafServer* blocked in another thread will return and can then notify competion.

Implications of the Activity lifecycle on graceful shutdown

The following diagram displays the activity lifecycle (as presented in **Android Documentation Activity Lifecycle**).




App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

The points of interest for the TEXAS RISE project were especially the onDestroy() method and the onPause() method. We have already discussed the onDestroy() method and how it can be used as a last chance to gracefully shutdown a service the activity is using. However, it turns out that the applications process can be terminated from the paused state, without calling onDestroy(). The following quote outlines the issue:

> *"If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish, or simply killing its process." (**Stack Overflow: Android Destroying Activities Killing Processes**)*

We actually need to do further investigation into this to understand the effect on private serivces when activities are killed. At present we only know that a service running in the same process as the activity will be destroyed for certain. We don't yet know if a service running in another process that is private to the application will also be destroyed. Regardless of the outcomes it raises questions:

1. If the process of the service is destroyed, it won't be destroyed gracefully; and
2. If the process of the service is not destroyed, it will continue to run, potentially reporting track positions etc. when the application is not running.

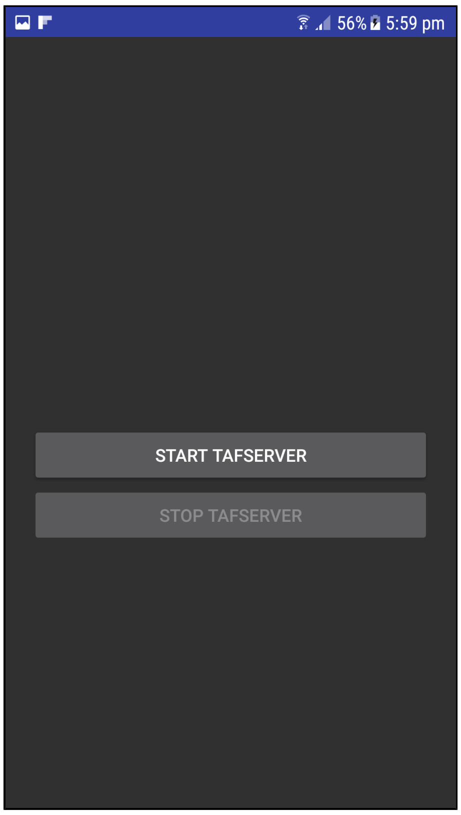
We can minimise the risk of the second question by sending a message to the TafService to inturn pause the services relevant to us in the TafServer, but of course the service and process will still exist. In anycase it is worth considering what to do in the paused state, perhaps we should continue publishing our position and continue receiving updates especially if we want to show historical information to the user. On the otherhand if we only want to see the situation at the present time, it may make sense to pause the service. It would be good to know if any of the service hooks are called prior to killing the related Activities process.

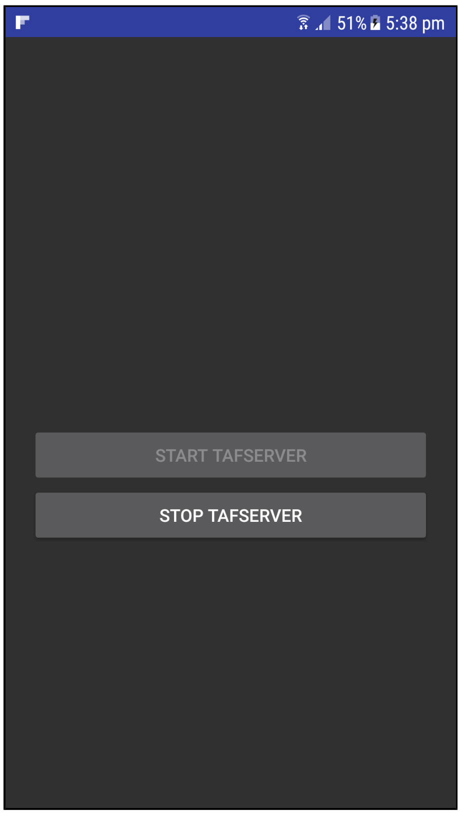
Noteworthy:

> *"Generally system kills the whole process this means all data including activities and static fields are destroyed. This is NOT done nicely - you won't get onDestroy or finialize() for any of your paused/stopped activities. This is why saveInstanceState() is called just before onPause method. onPause is basically the last method where you should save something because after this method you could never see onStop or onDestroy. System can just kill the process destroying all of your objects whatever they hold and whatever they are doing." (**Stack Overflow: Android Destroying Activities Killing Processes**)*

User Interaction

To demonstrate this work, a simple user interface was created (only for the unbound example) allowing the user to start and stop the TAFServer (then we were able to inspect the process through adb or the on device services list):





It would be good to in future extend this simple interface to at the very least show an updated status of the TAFServer and the state of the services currently loaded in it.

What Next?

- For some projects simply being able to run a pre-configured LASAGNE service in the foreground, bound to a single application may be the best usage scenario. Other projects may benefit from leveraging a shared TAFServer, in that case it may be useful to allow dynamic deployment and control of services with an enhanced SDDU-like GUI. A good starting point would be to build on the bound service example above and add some jni methods to control the TAFServer.
- It will be a good idea to run some tests with several other applications running. Perhaps run the TAFServer with OpenDDS ping /pong services (Note we have tested with RabbitMQ Example to date).
- Based on the above findings we shouldn't need a wakelock for a foreground service, but in case we find the the server is not running we may want to look into a **wakelock**.
- We should spend some more time looking at the potential issues of the OS killing a process of an activity that has bound to the TafService (will unbind get called? how can we cleanup? should we implement the saveInstanceState method)
- Update this page with zip files for each example...
- Write a checklist of gotchas and issues to look out for (e.g. especially with regards to service lifecycle such as binding services potentially changing service priorities).
- Write some information about the JNI methods and C++ side of things
- Write some information about the cordova plugin