Università degli Studi di Catania
Dipartimento di Matematica ed Informatica
Corso di Laurea Magistrale in Informatica

STUDIO DELLA GESTIONE DEI PERMESSI NELLE APPLICAZIONI ANDROID

Prof. Emiliano Tramontana

Francesco Anastasio

Sommario

Introduzione

1.1 Scopo del progetto

Il progetto si pone l'obiettivo di studiare la gestione dei permessi da parte del sistema operativo all'interno di un'applicazione Android sfruttando la programmazione ad aspetti e il codice sorgente del sistema operativo.
Inoltre, si ricerca l'utilizzo di design pattern noti come l'Authenticator o l'Authorizator per la gestione degli stessi.

1.2 Programmazione orientata agli Aspetti

La programmazione orientata agli aspetti è un paradigma basato sulla creazione di software, gli aspetti per l'appunto, che intercettano le iterazioni tra gli oggetti. Essa è utilizzata principalmente per ridurre la scrittura di codice comune tra varie classi e "iniettarlo" quando necessario, riuscendo ad avere, in questo modo, del codice comune ma concentrato in un solo punto.
Punti chiave della programmazione orientata agli aspetti sono:

1. **Aspetto**: ci permette di aggiungere dinamicamente comportamenti agli oggetti senza che questi ne siano a conoscenza.

2. **Join Point**: rappresenta un punto all'interno dell'esecuzione del programma come, ad esempio, la chiamata ad un metodo o la creazione di un oggetto.

```
//Join Point
private static final String MAINACTIVITY = "call(* com.example.francy.progettoingegneriasistemidistribuiti.MainActivity..*(..))";
```

Figura 1: Esempio di Join Point.

3. **Pointcut**: descrive le regole con cui associare l'esecuzione di un Advice ad un determinato Join Point.

```
//Pointcut
@Pointcut(MAINACTIVITY)
public void checkMainActivity(){}
```

Figura 2: Esempio di Pointcut.

4. **Advice**: descrive l'operazione da compiere da parte di un aspetto in un determinato Join Point. A differenza di un metodo che deve essere invocato, viene eseguito automaticamente quando si verifica un determinato evento (Join Point) ad una determinata condizione (Pointcut). Vi sono diversi tipi di Advice, ricordiamo: "before", "after", "around".

```
//Advice
@Around("checkMainActivity()")
public Object implementationcheckMainActivity(@NonNull ProceedingJoinPoint jp) throws Throwable {
    Object result = jp.proceed();
    Log.d( tag: DebugName.cont++ + ") " + DebugName.MAINACTIVITY, jp.toString());
    return result;
}
```

Figura 3: Esempio di Advice.

1.2.1 Libreria Aspectj

Gli aspetti non sono nativamente supportati da Android, per questo motivo è stata utilizzata una libreria di terze parti, Aspectj, che dopo l'installazione e la corretta configurazione ne permette l'uso.

1.2.2 Configurazione di Aspectj

Per configurare correttamente Aspectj in Android bisogna modificare i file build.grandle del progetto e del modulo che stiamo sviluppando affinché vengano aggiunte le corrette dipendenze:

Come si può vedere nella figura 4 seguente, nel build.grandle relativo al progetto è necessario aggiungere la repository "mavenCentral()" nel buildscript e il classpath "org.aspectj:aspectjtools:-VERSION-" nelle dipendenze:

```
buildscript {
    repositories {
        google()
        jcenter()
        mavenCentral() //For Aspectj
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.0'
        classpath 'org.aspectj:aspectjtools:1.9.4' //For Aspectj
    }
}

allprojects {
    repositories {
        google()
        jcenter()
        mavenCentral() //For Aspectj
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Figura 4: codice aggiunto al build.grandle del progetto.

Per quanto riguarda il build.grandle del modulo invece bisogna aggiungere tra le dipendenze la seguente riga:

"implementation 'org.aspectj:aspectjrt:1.9.4' //For Aspectj"

Inoltre è necessario aggiungere il codice seguente (Figura 5):

```
final def log = project.logger
final def variants = android.applicationVariants
variants.all { variant ->
    if (!variant.buildType.isDebuggable()) {
        log.debug("Skipping non-debuggable build type '${variant.buildType.name}'.")
        return;
    }
    JavaCompile javaCompile = variant.javaCompile
    javaCompile.doLast {
        String[] args = ["-showWeaveInfo",
                         "-1.5",
                         "-inpath", javaCompile.destinationDir.toString(),
                         "-aspectpath", javaCompile.classpath.asPath,
                         "-d", javaCompile.destinationDir.toString(),
                         "-classpath", javaCompile.classpath.asPath,
                         "-bootclasspath", project.android.bootClasspath.join(File.pathSeparator)]
        log.debug "ajc args: " + Arrays.toString(args)
        MessageHandler handler = new MessageHandler(true);
        new Main().run(args, handler);
        for (IMessage message : handler.getMessages(null, true)) {
            switch (message.getKind()) {
                case IMessage.ABORT:
                case IMessage.ERROR:
                case IMessage.FAIL:
                    log.error message.message, message.thrown
                    break;
                case IMessage.WARNING:
                    log.warn message.message, message.thrown
                    break;
                case IMessage.INFO:
                    log.info message.message, message.thrown
                    break;
                case IMessage.DEBUG:
                    log.debug message.message, message.thrown
                    break;
            }
        }
```

Figura 5: codice aggiunto al build.grandle del modulo.

La configurazione è così terminata, eseguire un Rebuild Project e la libreria sarà funzionante.

1.3 Android

Android è un sistema operativo Open Source disponibile oggigiorno per molteplici piattaforme, dagli smartphone alle automobili (Android Auto), dalle TV (Android TV) agli smartwatch (Android WEAR).

Esso nasce come progetto di una piccola azienda, "Android, Inc." che venne acquisita successivamente, nel 2005, da Google Inc.

La presentazione ufficiale al mondo del nuovo sistema operativo è stata fatta il 5 Novembre 2007 mentre il primo smartphone venduto con tale sistema operativo a bordo è l'HTC Dream, lanciato sul mercato il 22 Ottobre 2008.
Da allora il sistema operativo è stato costantemente aggiornato, con vari piccoli aggiornamenti durante l'anno e una release più corposa circa una volta l'anno, con cambio di nomenclatura e numero di versione.

1.3.1 Permessi in Android

Un permesso in Android è la richiesta da parte di un'applicazione ad utilizzare un determinato servizio, sensore o funzionalità del sistema esterno all'applicazione stessa.
Tutti i permessi necessari all'applicazione devono essere dichiarati nel Manifest, un file Xml che contiene le informazioni principali dell'applicazione stessa come, ad esempio, la versione e il nome.
Si dividono in tre tipologie:

- Normal: utilizzato per tutti quei permessi che non servono a svolgere operazioni critiche sulle risorse. Questi sono concessi automaticamente all'applicazione.
- Dangerous: permessi che permettono in qualche modo di controllare il dispositivo o i dati presenti in esso. La concessione di questa tipologia di permessi deve essere data in modo esplicito dall'utente dopo aver ricevuto richiesta da parte dell'applicazione. L'utente può anche rifiutarne la concessione e, in questo caso, l'applicazione dovrebbe gestire il rifiuto.
- Signature: il permesso viene concesso solo a quelle applicazioni firmate con la stessa chiave dell'applicazione che definisce i permessi.

1.4 Design Pattern

Nell'ambito dell'ingegneria del software, con design pattern si intende una descrizione o modello logico da applicare per la risoluzione di un problema ricorrente.

1.4.1 Authenticator

Il design pattern Authenticator descrive un meccanismo generale per l'identificazione e l'autenticazione di un generico client in un server.

1.4.2 Authoriser

Il design pattern Authiriser aiuta l'ingegnere del software a risolvere due problemi:

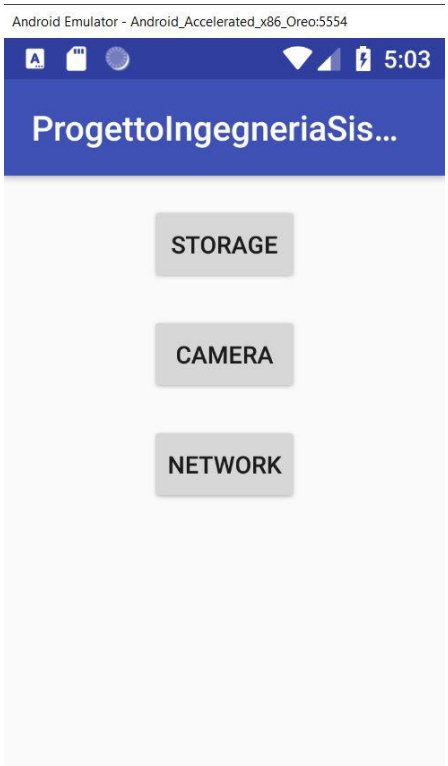
- Consistenza di permessi: ogni utente assume un ruolo e i ruoli sono standard almeno all'interno di un determinato contesto.
- Responsabilità: si deve sapere sempre chi ha fatto cosa, quindi si devono registrare gli accessi degli utenti alle risorse.

Esperimenti

2.1 Premessa

Per eseguire i vari esperimenti e studiare il workflow delle varie richieste è stata creata una semplicissima applicazione Android, con una sola Activity, la MainActivity contenente tre pulsanti:

1. **Storage**: consente la richiesta del permesso di tipo "Dangerous" per accedere alla memoria del dispositivo.
2. **Camera**: consente la richiesta del permesso di tipo "Dangerous" per utilizzare la fotocamera del dispositivo. Il permesso ***NON*** è dichiarato nel manifest.
3. **Network**: consente la richiesta del permesso di tipo "Normal" per utilizzare la rete.



Interfaccia applicazione Android.

Alla pressione di uno dei tre tasti viene invocata la funzione checkPermission (Figura 6) con il corretto Permission passato come input.

```
// Function to check and request permission.
public void checkPermission(String permission, int requestCode)
{
    if (ContextCompat.checkSelfPermission( context: MainActivity.this, permission)
            == PackageManager.PERMISSION_DENIED) {

        // Requesting the permission
        ActivityCompat.requestPermissions( activity: MainActivity.this,
                new String[] { permission },
                requestCode);
    }
    else {
        Toast.makeText( context: MainActivity.this,
                text: "Permission already granted",
                Toast.LENGTH_SHORT)
                .show();
    }
}
```

Figura 6: funzione per la gestione delle richieste di permessi.

Con l'aiuto della programmazione ad aspetti e il download del codice sorgente di Android si è provato a ipotizzare il percorso che le richieste fanno all'interno del codice.

2.2 Richiesta di un permesso Normal

Nel caso di richiesta di un permesso di tipo Normal dichiarato nel Manifest viene sempre restituito un "PERMISSION_GRANTED".
Dall'IDE di sviluppo, Android Studio, possiamo vedere che il flusso inizia con l'invocazione del metodo:

(MainActivity) ContextCompat.checkSelfPermission(MainActivity.this, permission)

per poi passare all'invocazione del metodo:

(ContextCompact) context.checkPermission(permission, Process.myPid(), Process.myUid())

```
D/13) Francionic - Android.Support:: call(int android.support.v4.content.ContextCompat.checkSelfPermission(Context, String))
D/14) Francionic - MainActivity:: call(void com.example.francy.progettoingegneriasistemidistribuiti.MainActivity.checkPermission(String, int))
```

Aspetti permesso Normal.

Da qui in poi, né con il Debug dell'IDE né con la programmazione ad aspetti si è riusciti a seguire il flusso. Quindi, dopo aver scaricato e studiato il codice sorgente completo, si è provato ad ipotizzarlo:
nella precedente chiamata, l'oggetto context è un oggetto di tipo ContextImpl, quindi il metodo invocato è il seguente:

```
@Override
public int checkPermission(String permission, int pid, int uid) {
    if (permission == null) {
        throw new IllegalArgumentException("permission is null");
    }

    try {
        return ActivityManagerNative.getDefault().checkPermission(
                permission, pid, uid);
    } catch (RemoteException e) {
        return PackageManager.PERMISSION_DENIED;
    }
}
```

Figura 7: ContextImpl.checkPermission

Da qui l'esecuzione prosegue, ammesso un permission diverso da "null", verso

ActivityManagerNative.getDefault().checkPermission(permission, pid, uid);

ActivityManagerNativce.getDefault() ritorna un'istanza di un oggetto di tipo *IActivityManager,* nel caso specifico si ipotizza sia di tipo *ActivityManagerService*, quindi il checkPermission ad essere invocato è il seguente:

```
public int checkPermission(String permission, int pid, int uid) {
    if (permission == null) {
        return PackageManager.PERMISSION_DENIED;
    }
    return checkComponentPermission(permission, pid, uid, -1, true);
}
```

Figura 8: ActivityManagerService.checkPermission

Il controllo dell'esecuzione passa a *ActivityManagerService.chekComponentPermission*

```
int checkComponentPermission(String permission, int pid, int uid,
        int owningUid, boolean exported) {
    // We might be performing an operation on behalf of an indirect binder
    // invocation, e.g. via {@link #openContentUri}.  Check and adjust the
    // client identity accordingly before proceeding.
    Identity tlsIdentity = sCallerIdentity.get();
    if (tlsIdentity != null) {
        Slog.d(TAG, "checkComponentPermission() adjusting {pid,uid} to {"
                + tlsIdentity.pid + "," + tlsIdentity.uid + "}");
        uid = tlsIdentity.uid;
        pid = tlsIdentity.pid;
    }

    // Root, system server and our own process get to do everything.
    if (uid == 0 || uid == Process.SYSTEM_UID || pid == MY_PID) {
        return PackageManager.PERMISSION_GRANTED;
    }
    // If there is a uid that owns whatever is being accessed, it has
    // blanket access to it regardless of the permissions it requires.
    if (owningUid >= 0 && uid == owningUid) {
        return PackageManager.PERMISSION_GRANTED;
    }
    // If the target is not exported, then nobody else can get to it.
    if (!exported) {
        Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid=" + owningUid);
        return PackageManager.PERMISSION_DENIED;
    }
    if (permission == null) {
        return PackageManager.PERMISSION_GRANTED;
    }
    try {
        return AppGlobals.getPackageManager()
                .checkUidPermission(permission, uid);
    } catch (RemoteException e) {
        // Should never happen, but if it does... deny!
        Slog.e(TAG, "PackageManager is dead?!?", e);
    }
    return PackageManager.PERMISSION_DENIED;
}
```

Figura 9: ActivityManagerService.checkComponentPermission

All'interno di questo metodo viene deciso se concedere o rifiutare il permesso a meno che in un caso, in cui l'esecuzione passa a (ipotizzato) *PackageManagerService.checkUidPermission*:

```
public int checkUidPermission(String permName, int uid) {
    synchronized (mPackages) {
        Object obj = mSettings.getUserIdLPr(uid);
        if (obj != null) {
            GrantedPermissions gp = (GrantedPermissions)obj;
            if (gp.grantedPermissions.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        } else {
            HashSet<String> perms = mSystemPermissions.get(uid);
            if (perms != null && perms.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        }
    }
    return PackageManager.PERMISSION_DENIED;
}
```

Figura 10: PackageManagerService.checkUidPermission

Questo sembra essere l'ultimo metodo invocato, qui infatti si prendono i setting dell'utente e i setting del sistema relativi ai permessi e si controlla se all'interno di essi è contenuto il permesso richiesto attualmente, se lo è allora il permesso è "Garantito", altrimenti il permesso è "Negato".

Essendo questo il caso di un permesso "Normal" esso si troverà, presumibilmente, all'interno dell'hashSet precedente inserito in automatico durante l'avvio dell'applicazione.

2.3 Richiesta di un permesso Dangerous

La richiesta di un permesso Dangerous inizia esattamente come il precedente, ossia controllando che l'utente non abbia già concesso il permesso e, fino a quando l'utente non lo confermerà, il workflow sarà il seguente:

dato il codice a Figura 6, il *ContextCompact.checkSelfPermission* restituirà un *PackageManger.PERMISSION_DENIED*, si entrerà quindi all'interno dell'if e verrà invocato il metodo *ActivityCompat.requestPermissions()*:

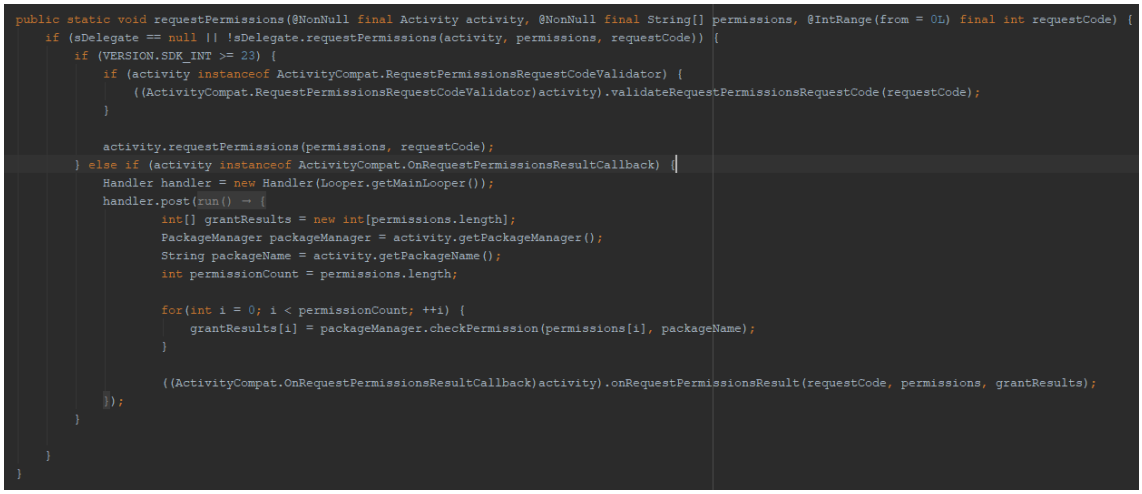


Figura 11: ActivityCompact.requestPermissions

Essendo la versione dell'SDK >= 23 il metodo seguente ad essere invocato (confermato anche dal debug) è *activity.requestPermissions*:

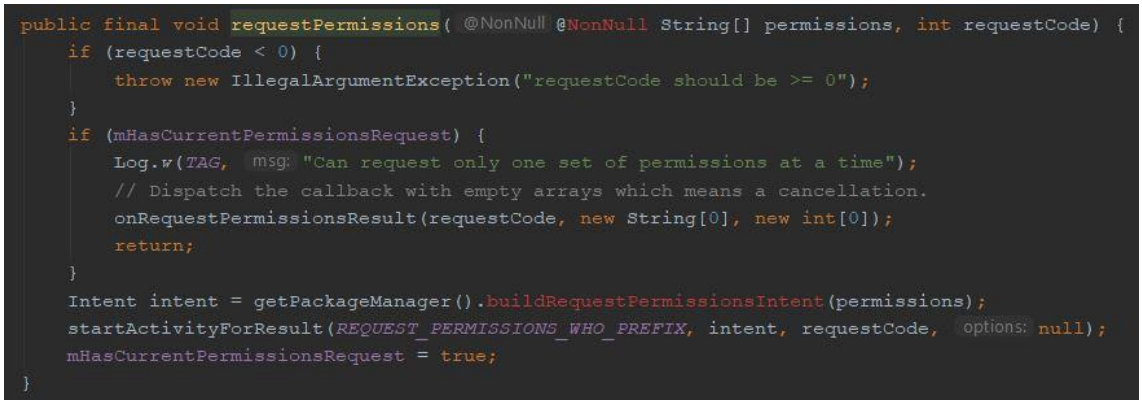


Figura 12: Activity.requestPermissions

Da questo punto in poi del codice, come per il caso precedente, né il debug, né gli aspetti riescono ad aiutarci per seguire il flusso, i prossimi passi sono quindi ipotesi derivanti dallo studio del codice sorgente:
dopo aver creato un "intent" specifico per il permesso richiesto l'esecuzione passa al metodo *Activity.startActivityForResult (Figura 13)*:

```
@Override
public void startActivityForResult(
        String who, Intent intent, int requestCode, @Nullable Bundle options) {
    Uri referrer = onProvideReferrer();
    if (referrer != null) {
        intent.putExtra(Intent.EXTRA_REFERRER, referrer);
    }
    options = transferSpringboardActivityOptions(options);
    Instrumentation.ActivityResult ar =
        mInstrumentation.execStartActivity(
            this, mMainThread.getApplicationThread(), mToken, who,
            intent, requestCode, options);
    if (ar != null) {
        mMainThread.sendActivityResult(
            mToken, who, requestCode,
            ar.getResultCode(), ar.getResultData());
    }
    cancelInputsAndStartExitTransition(options);
}
```

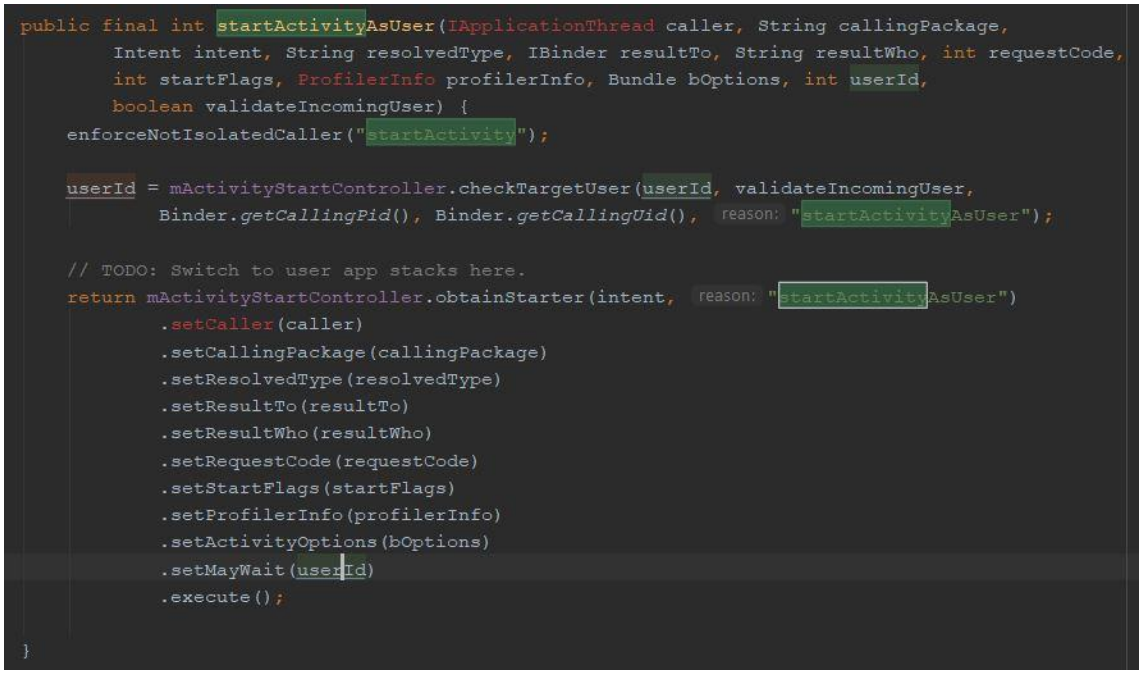
Figura 13: Activity.startActivityForResult.

In questo metodo verrà invocato *mInstrumentatio.execStartActivity*, che restituisce un'*ActivityResult (Figura 14)*:

```
    if (mActivityMonitors != null) {
        synchronized (mSync) {
            final int N = mActivityMonitors.size();
            for (int i=0; i<N; i++) {
                final ActivityMonitor am = mActivityMonitors.get(i);
                ActivityResult result = null;
                if (am.ignoreMatchingSpecificIntents()) {
                    result = am.onStartActivity(intent);
                }
                if (result != null) {
                    am.mHits++;
                    return result;
                } else if (am.match(who, null, intent)) {
                    am.mHits++;
                    if (am.isBlocking()) {
                        return requestCode >= 0 ? am.getResult() : null;
                    }
                    break;
                }
            }
        }
    }
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        int result = ActivityManager.getService()
            .startActivity(whoThread, who.getBasePackageName(), intent,
                    intent.resolveTypeIfNeeded(who.getContentResolver()),
                    token, target != null ? target.mEmbeddedID : null,
                    requestCode, 0, null, options);
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
```

Figura 14: Parte del metodo Instrumentation.execStartActivity.

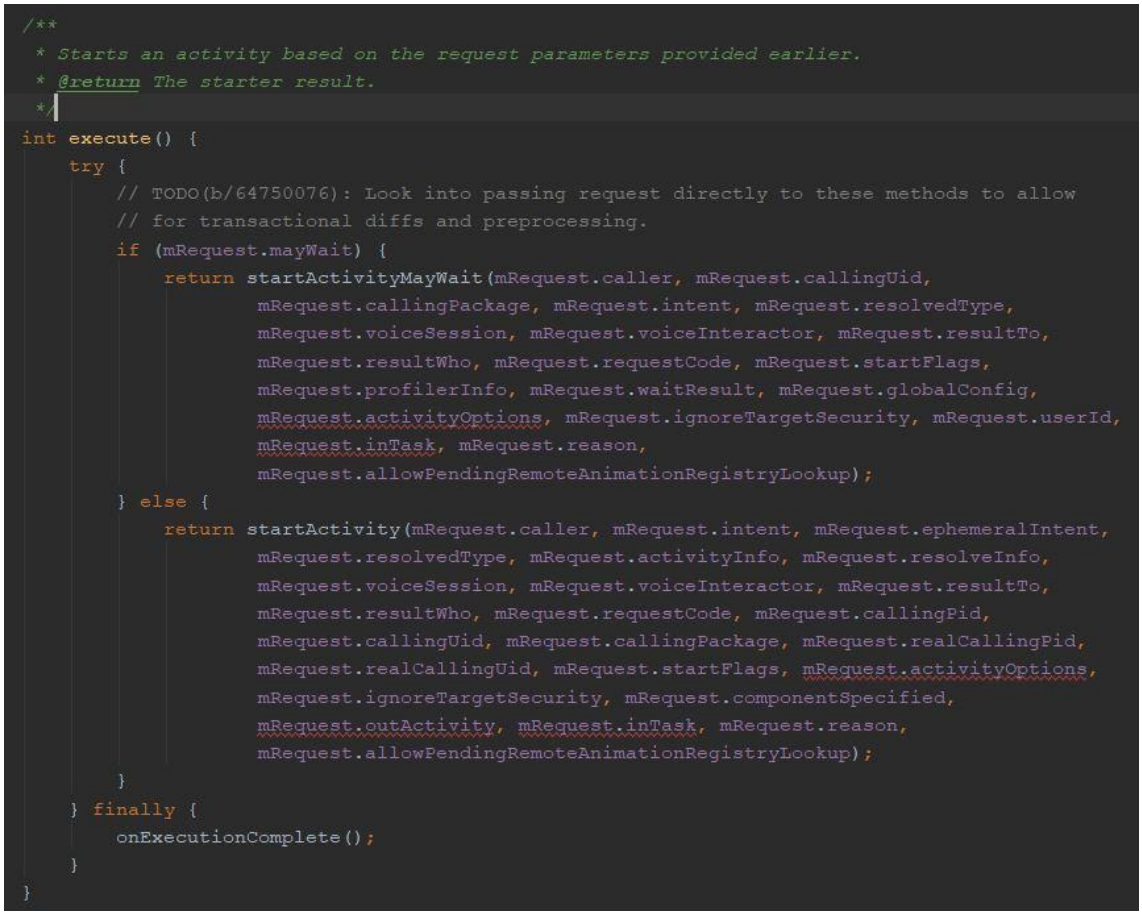
Da qui l'esecuzione passa al metodo *ActivityManagerService.startActivity* che, dopo varie chiamate interne, ci fa arrivare all'esecuzione del metodo *ActivityManagerService.startActivityAsUser (Figura 15):*



```
public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
        Intent intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
        int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int userId,
        boolean validateIncomingUser) {
    enforceNotIsolatedCaller("startActivity");

    userId = mActivityStartController.checkTargetUser(userId, validateIncomingUser,
            Binder.getCallingPid(), Binder.getCallingUid(),  reason: "startActivityAsUser");

    // TODO: Switch to user app stacks here.
    return mActivityStartController.obtainStarter(intent,  reason: "startActivityAsUser")
            .setCaller(caller)
            .setCallingPackage(callingPackage)
            .setResolvedType(resolvedType)
            .setResultTo(resultTo)
            .setResultWho(resultWho)
            .setRequestCode(requestCode)
            .setStartFlags(startFlags)
            .setProfilerInfo(profilerInfo)
            .setActivityOptions(bOptions)
            .setMayWait(userId)
            .execute();

}
```

Figura 15: ActivityManagerService.startActivityAsUser

La chiamata *mActivityStartController.obtainStarter(….)* e i seguenti "Add" fino al metodo *execute()* fanno sì, dopo aver configurato l'activity, che venga invocato il metodo *ActivityStarter.execute() (Figura 16)*:



```
/**
 * Starts an activity based on the request parameters provided earlier.
 * @return The starter result.
 */
int execute() {
    try {
        // TODO(b/64750076): Look into passing request directly to these methods to allow
        // for transactional diffs and preprocessing.
        if (mRequest.mayWait) {
            return startActivityMayWait(mRequest.caller, mRequest.callingUid,
                    mRequest.callingPackage, mRequest.intent, mRequest.resolvedType,
                    mRequest.voiceSession, mRequest.voiceInteractor, mRequest.resultTo,
                    mRequest.resultWho, mRequest.requestCode, mRequest.startFlags,
                    mRequest.profilerInfo, mRequest.waitResult, mRequest.globalConfig,
                    mRequest.activityOptions, mRequest.ignoreTargetSecurity, mRequest.userId,
                    mRequest.inTask, mRequest.reason,
                    mRequest.allowPendingRemoteAnimationRegistryLookup);
        } else {
            return startActivity(mRequest.caller, mRequest.intent, mRequest.ephemeralIntent,
                    mRequest.resolvedType, mRequest.activityInfo, mRequest.resolveInfo,
                    mRequest.voiceSession, mRequest.voiceInteractor, mRequest.resultTo,
                    mRequest.resultWho, mRequest.requestCode, mRequest.callingPid,
                    mRequest.callingUid, mRequest.callingPackage, mRequest.realCallingPid,
                    mRequest.realCallingUid, mRequest.startFlags, mRequest.activityOptions,
                    mRequest.ignoreTargetSecurity, mRequest.componentSpecified,
                    mRequest.outActivity, mRequest.inTask, mRequest.reason,
                    mRequest.allowPendingRemoteAnimationRegistryLookup);
        }
    } finally {
        onExecutionComplete();
    }
}
```

Figura 16: ActivityStarter.execute()

Dopo varie chiamate il flusso arriva al metodo (1) *ActivityStarter.startActivity (IApplicationThread caller, Intent intent, Intent ephemeralIntent,*

String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,

IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,

IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,

String callingPackage, int realCallingPid, int realCallingUid, int startFlags,

SafeActivityOptions options,

boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[] outActivity,

TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup)

del quale non si riporta il codice per semplicità. Questo metodo, dopo aver fatto tutti i controlli del caso, avvia l'Activity per la richiesta del permesso nel caso la richiesta sia legittima, restituisce un errore invece nel caso la richiesta sia "illegale".

Dopo i vari "Return" il controllo ritorna al metodo *Activity.startActivityForResult* (Figura 13) che, nel caso di ar != null invoca *mMainThread.sendActivityResult (Figura 17)*:

```
public final void sendActivityResult(
        IBinder token, String id, int requestCode,
        int resultCode, Intent data) {
    if (DEBUG_RESULTS) Slog.v(TAG, "sendActivityResult: id=" + id
            + " req=" + requestCode + " res=" + resultCode + " data=" + data);
    ArrayList<ResultInfo> list = new ArrayList<~>();
    list.add(new ResultInfo(id, requestCode, resultCode, data));
    final ClientTransaction clientTransaction = ClientTransaction.obtain(mAppThread, token);
    clientTransaction.addCallback(ActivityResultItem.obtain(list));
    try {
        mAppThread.scheduleTransaction(clientTransaction);
    } catch (RemoteException e) {
        // Local scheduling
    }
}
```

Figura 17: ActivityThread.sendActivityResult

Questo metodo non fa altro che schedulare una clientTransaction e dopo varie invocazioni di metodi il controllo arriva a *ActivityThread.deliverResults* (Figura 18):

```
private void deliverResults(ActivityClientRecord r, List<ResultInfo> results, String reason) {
    final int N = results.size();
    for (int i=0; i<N; i++) {
        ResultInfo ri = results.get(i);
        try {
            if (ri.mData != null) {
                ri.mData.setExtrasClassLoader(r.activity.getClassLoader());
                ri.mData.prepareToEnterProcess();
            }
            if (DEBUG_RESULTS) Slog.v(TAG,
                    "Delivering result to activity " + r + " : " + ri);
            r.activity.dispatchActivityResult(ri.mResultWho,
                    ri.mRequestCode, ri.mResultCode, ri.mData, reason);
        } catch (Exception e) {
            if (!mInstrumentation.onException(r.activity, e)) {
                throw new RuntimeException(
                        "Failure delivering result " + ri + " to activity "
                        + r.intent.getComponent().toShortString()
                        + ": " + e.toString(), e);
            }
        }
    }
}
```

Figura 18: ActivityThread.deliverResults

L'esecuzione passa così, a meno di errori, al metodo *r.activity.dispatchActivityResult* che, nel caso in cui ritrovi i corretti parametri invoca uno tra *Activity.dispatchRequestPermissionsResult (Figura 19, prima parte)* – *Activity.dispatchRequestPermissionsResultToFragment (Figura 19, seconda parte):*

```
private void dispatchRequestPermissionsResult(int requestCode, Intent data) {
    mHasCurrentPermissionsRequest = false;
    // If the package installer crashed we may have not data - best effort.
    String[] permissions = (data != null) ? data.getStringArrayExtra(
            PackageManager.EXTRA_REQUEST_PERMISSIONS_NAMES) : new String[0];
    final int[] grantResults = (data != null) ? data.getIntArrayExtra(
            PackageManager.EXTRA_REQUEST_PERMISSIONS_RESULTS) : new int[0];
    onRequestPermissionsResult(requestCode, permissions, grantResults);
}

private void dispatchRequestPermissionsResultToFragment(int requestCode, Intent data,
        Fragment fragment) {
    // If the package installer crashed we may have not data - best effort.
    String[] permissions = (data != null) ? data.getStringArrayExtra(
            PackageManager.EXTRA_REQUEST_PERMISSIONS_NAMES) : new String[0];
    final int[] grantResults = (data != null) ? data.getIntArrayExtra(
            PackageManager.EXTRA_REQUEST_PERMISSIONS_RESULTS) : new int[0];
    fragment.onRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Figura 19: Activity.dispatchRequestPermissionsResult e Activity.dispatchRequestPermissionsResultToFragment

Entrambi i metodi precedenti portano ad una chiamata del metodo *onRequestPermissionsResult,* un metodo di cui è necessario fare l'override quando si richiede un permesso.

Il controllo dell'esecuzione è quindi ritornato al programmatore che può gestire la risposta dell'utente alla richiesta del permesso tramite questo metodo per poi continuare con le proprie operazioni.

Lo screen dei messaggi derivanti dagli aspetti è il seguente:

```
D/5) Francionic - Android.Support:: call(int android.support.v4.content.ContextCompat.checkSelfPermission(Context, String))

    --------- beginning of system
D/6Francionic - checkSelfPermission:: call(void android.support.v4.app.ActivityCompat.requestPermissions(Activity, String[], int))
D/7) Francionic - MainActivity:: call(void com.example.francy.progettoingegneriasistemidistribuiti.MainActivity.checkPermission(String, int))
D/8) Francionic - OnRequestCallback:: execution(void com.example.francy.progettoingegneriasistemidistribuiti.MainActivity.onRequestPermissionsResult(int, String[], int[]))
```

Aspetti permesso Dangerous.

2.4 Richiesta di un permesso non definito nel Manifest.

È stato inserito nell'applicazione un pulsante per richiedere il permesso "Camera" che però non è stato definito nel Manifest.

Il flusso in questo caso è identico al precedente, almeno fino ad arrivare al punto (1), ossia al metodo *ActivityStarter.startActivity (Figura 20)*. Nel metodo specifico, infatti, vengono effettuati tutti i controlli e, presumibilmente, verrà restituito l'errore *ActivityManager.START_PERMISSION_DENIED*.

```
private int startActivity(IApplicationThread caller, Intent intent, Intent ephemeralIntent,
        String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
        IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
        IBinder resultTo, String resultWho, int requestCode, int callingPid, int callingUid,
        String callingPackage, int realCallingPid, int realCallingUid, int startFlags,
        SafeActivityOptions options,
        boolean ignoreTargetSecurity, boolean componentSpecified, ActivityRecord[] outActivity,
        TaskRecord inTask, boolean allowPendingRemoteAnimationRegistryLookup) {
    int err = ActivityManager.START_SUCCESS;
    // Pull the optional Ephemeral Installer-only bundle out of the options early.
    final Bundle verificationBundle
            = options != null ? options.popAppVerificationBundle() : null;

    ProcessRecord callerApp = null;
    if (caller != null) {
        callerApp = mService.getRecordForAppLocked(caller);
        if (callerApp != null) {
            callingPid = callerApp.pid;
            callingUid = callerApp.info.uid;
        } else {
            Slog.w(TAG, "Unable to find app for caller " + caller
                    + " (pid=" + callingPid + ") when starting: "
                    + intent.toString());
            err = ActivityManager.START_PERMISSION_DENIED;
        }
    }

    final int userId = aInfo != null && aInfo.applicationInfo != null
            ? UserHandle.getUserId(aInfo.applicationInfo.uid) : 0;

    if (err == ActivityManager.START_SUCCESS) {
        Slog.i(TAG, "START u" + userId + " {" + intent.toShortString(true, true, true, false)
```

Figura 20: ActivityStarter.startActivity.

Il suddetto errore verrà poi propagato all'indietro fino al metodo *Instrumentation.execStartActivity (Figura 14)* che proseguendo con l'esecuzione invocherà *Instrumentation.checkStartActivityResult (Figura 21)* che si occuperà di sollevare un'eccezione e quindi bloccare il flusso. In questo modo non verrà mostrata nessuna finestra di dialogo di richiesta all'utente e il permesso sarà considerato rifiutato.

2.5 Utilizzo dei design pattern

Il design pattern Authenticator non sembra essere utilizzato in nessuno dei workflow analizzati.

Per quanto riguarda il design pattern Authorizator invece, sembra essere stato utilizzato per la gestione dei permessi.

In Figura 10, metodo *PackageManagerService.checkUidPermission*, possiamo notare come per il permesso richiesto venga ricercato all'interno di due entità, una di tipo Object e una di tipo *HashSet<String>*.

Andiamo ad analizzare meglio i due casi:

- Nel primo caso abbiamo un'entità di tipo Object estratta con la seguente chiamata: *Object obj = mSettings.getUserIdLPr(uid) (Figura 21):*

```
public Object getUserIdLPr(int uid) {
    if (uid >= PackageManagerService.FIRST_APPLICATION_UID) {
        final int N = mUserIds.size();
        final int index = uid - PackageManagerService.FIRST_APPLICATION_UID;
        return index < N ? mUserIds.get(index) : null;
    } else {
        return mOtherUserIds.get(uid);
    }
}
```

Figura 21: Settings.getUserIdlPr.

L'oggetto restituito è di tipo *SparseArray<Object>* e, ipotizzando che per uid (userId) non si intenda quello dell'utente ma quello dell'applicazione che è usata in quel momento, quest'oggetto tiene traccia di tutti i permessi che sono stati autorizzati dall'utente per ogni applicazione.

- Nel secondo caso invece abbiamo un oggetto di tipo *HashSet<String>* estrapolata dalla seguente invocazione: *HashSet<String> perms = mSystemPermissions.get(uid) (Figura 22).*

  mSystemPermissions è un oggetto locale alla classe, e l'unico "put" in quest'oggetto viene eseguito in un metodo chiamato *readPermissionsFromXml.*

  Deduciamo da ciò, che questo è un oggetto che legge il Manifest e registra tutti i permessi dichiarati in esso.

Conclusioni

Lo studio è stato portato a termine con successo, il workflow di tutti gli esperimenti analizzati è stato completato grazie al codice sorgente, infatti né il debug né la programmazione orientata agli aspetti sono stati di grande aiuto per il completamento del progetto.