UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Realizzazione di un sistema IoT basato su Android per l'acquisizione multimediale di lezioni universitarie

Supervisore

Marco Ronchetti

Laureando

Matteo Contrini

Anno accademico 2018/2019

Ringraziamenti

La presente tesi è stata redatta nell'ambito di un tirocinio formativo svolto presso l'azienda AXIA Studio. Grazie a Tiziano, Chloe e al prof. Ronchetti per avermi seguito nel percorso.

Indice

Sommario

L'oggetto di questa tesi è lo sviluppo di un sistema di cattura e registrazione multimediale di lezioni universitarie, conferenze e seminari. Il sistema prodotto si basa su un dispositivo fisico dotato di apposito hardware per la cattura video, sul quale viene eseguito un apposito software su sistema operativo Android.

Il sistema è stato realizzato nell'ambito del progetto LODE (Lectures On DEmand) del professor Marco Ronchetti (Università di Trento), ed è stato elaborato durante un tirocinio formativo presso l'azienda AXIA Studio, con l'obiettivo finale di affinare la soluzione e di immetterla sul mercato delle soluzioni IoT accademiche.

Il progetto LODE, nel cui contesto questo lavoro si inserisce, è un progetto con l'obiettivo di fornire una soluzione low cost per l'acquisizione video delle lezioni universitarie. L'ultima versione del sistema prevede la possibilità di registrare flussi multimediali multipli, in particolare il video in uscita dal computer del docente, il video acquisito da una videocamera IP che riprende il docente e/o la lavagna, e l'audio di un microfono indossato dal docente. I contenuti sono poi elaborati e pubblicati su una pagina web accessibile dagli studenti del corso.

Una funzionalità del sistema prevede anche che durante le lezioni lo studente possa catturare degli screenshot di ciò che viene proiettato in quel momento, e inserire delle annotazioni testuali o disegnare sulle catture.

Nella sua ultima iterazione, LODE prevede l'uso di un dispositivo fisico, soprannominato LodeBox, che incorpora un "mini-computer" Raspberry Pi. Tramite apposite estensioni hardware, in particolare un modulo "HDMI to CSI-2", il dispositivo è in grado di acquisire un input HDMI come se si trattasse del video di una videocamera, e di sfruttare funzionalità come l'anteprima su schermo e la registrazione H.264 tramite encoder hardware. Il dispositivo prevede anche la possibilità di collegare un proiettore tramite uscita HDMI, un microfono tramite dongle USB, e una videocamera IP/RTSP (Real Time Streaming Protocol) tramite la rete locale.

Questa soluzione si scontra però con delle difficoltà che rendono difficile la realizzazione di un sistema affidabile e distribuibile su larga scala. Tra le problematiche principali si evidenziano la difficoltà nel gestire situazioni "plug and play", come la disconnessione e la riconnessione del cavo HDMI, la mancanza di storage integrato duraturo e ad alte prestazioni, le scarse garanzie sulla disponibilità del modulo hardware di conversione HDMI.

Ci si è orientati quindi verso la ricerca di SBC (Single-Board Computer) alternative più adatte per la realizzazione di applicazioni multimediali. Tra le soluzioni più accessibili dal punto di vista economico spiccano molte board basate sul sistema operativo Android, che offre i benefici di avere una piattaforma ben nota, documentata e con la possibilità di sostituire l'hardware riusando gran parte del codice scritto.

Gran parte del mio lavoro si è quindi concentrato sullo sviluppo di alcune applicazioni Java/Kotlin, con lo scopo di verificare la fattibilità di una versione di LodeBox con piattaforma Android.

La peculiarità fondamentale di molte board Android con input HDMI è che permettono di sfruttare direttamente le API di Android per l'acquisizione di video e immagini, permettendo di scrivere codice che non sia strettamente legato all'hardware. La piattaforma Android prevede due versioni delle API per l'accesso alla fotocamera, in particolare la classe `android.hardware.Camera` e le classi contenute in `android.hardware.camera2.*`. A partire da Android 5.0 (livello API 21), la classe `Camera` è deprecata ed è invece consigliato l'uso delle API `camera2`, più avanzate ma anche più complesse da usare. Ciò non ostante, non è garantito che ogni dispositivo con Android 5 o superiore supporti a pieno le API `camera2`.

È stato quindi necessario approfondire il funzionamento di entrambe le versioni delle API, per determinare se entrambe permettono di soddisfare i requisiti di LODE e quale versione conviene usare a seconda dell'hardware che si ha a disposizione.

Oltre all'input HDMI, si ha la necessità di registrare anche una videocamera esterna, che per contenere i costi è una videocamera IP raggiungibile nella rete locale tramite il protocollo di streaming RTSP. Di conseguenza, il flusso può essere registrato direttamente sulla board, utilizzando una versione della libreria `ffmpeg` compilata per funzionare su Android.

Un altro punto importante dello sviluppo di un'applicazione embedded per Android è la realizzazione di una modalità *kiosk*, cioè una situazione in cui l'utilizzatore del sistema può utilizzare solo ed esclusivamente una singola applicazione, senza poter accedere alle altre parti del sistema operativo o ad altre funzioni. Questo ha portato a sperimentare diverse soluzioni per assicurarsi che l'applicazione resti sempre a schermo intero, tra cui la modalità "lock task" di Android e l'avvio automatico come applicazione launcher.

Il risultato di questa fase è stato un insieme di applicazioni-prototipo che sono state sperimentate sul campo durante alcune lezioni presso l'Università di Trento, permettendo di raccogliere importanti riscontri sul funzionamento del sistema.

A questo punto la fattibilità delle funzionalità di base che il sistema deve fornire sono state verificate, e i passi successivi hanno riguardato altri aspetti per migliorare il funzionamento del software.

Uno di questi è la possibilità di mettere in pausa e riprendere la registrazione della lezione. L'idea considerata è stata quella di non sospendere la registrazione ma di escludere in post-produzione i segmenti di video corrispondenti alle pause. Questo è agevolmente ottenibile grazie a `ffmpeg` e con l'ausilio di uno script che generi il comando (potenzialmente lungo) per ritagliare i segmenti. Nel caso in cui non fosse possibile avere una registrazione unica, una tecnica simile può essere adottata per unire i segmenti video in fase di post-elaborazione.

Un'altra questione approfondita riguarda la sincronizzazione dei flussi, con lo scopo principale di evitare che l'audio risulti troppo sfasato rispetto al video, ed evitare che sia visibile lo sfasamento tra voce registrata e labiale del relatore. Il problema sorge principalmente per via della presenza del video RTSP, la cui latenza è difficilmente stimabile[1]. La soluzione sperimentale proposta prevede di incorporare l'audio nella registrazione video HDMI (in modo da ottenere una naturale sincronizzazione tra i due flussi acquisiti via cavo), e di occuparsi invece di sincronizzare video HDMI e RTSP. L'implementazione sperimentale è stata ottenuta mostrando su schermo un marcatore visivo che permetta di individuare con precisione un instante comune tra i due flussi video, e di conseguenza sincronizzarli.

Come accennato, un'altra funzione del sistema LODE prevede che lo studente possa prendere appunti in tempo reale durante lo svolgimento delle lezioni, catturando degli screenshot di ciò che vede in quel momento. Per evitare di inviare inutilmente screenshot al server nel caso in cui non ci sia stato nessun cambiamento percepibile nell'immagine, un sistema intelligente può rilevare le differenze tra i fotogrammi per determinare se l'immagine è nuova o invariata. Dopo aver appurato empiricamente che il confronto di tutti i pixel di due fotogrammi risulta computazionalmente dispendioso, un'alternativa considerata è quella di scegliere con un fattore di casualità un numero limitato di pixel "salienti". A livello intuitivo, il metodo sviluppato realizza una griglia con un numero oscillante di righe e colonne, le cui intersezioni determinano una quantità limitata di pixel che possono essere utilizzati per rilevare rapidamente le differenze tra i fotogrammi.

I capitoli di questa tesi approfondiscono più dettagliatamente le problematiche e le soluzioni che sono state sintetizzate in questo sommario.

1 Introduzione

Questo capitolo introduce il progetto LODE sviluppato presso l'Università di Trento, nel cui contesto questa tesi si inserisce. Sono approfonditi motivazioni e vantaggi, ed è introdotta l'ultima versione del sistema, cioè un dispositivo hardware dal nome LodeBox. Sono infine presentate le principali problematiche legate a LodeBox e l'hardware alternativo considerato per l'evoluzione del progetto.

[1] La latenza è influenzata dalla rete, dai tempi di codifica e dai buffer di trasmissione e ricezione. Non è quindi facile trovare un intervallo di tempo abbastanza preciso per sincronizzare il video con gli altri flussi.

1.1 LODE: introduzione e motivazioni

LODE (Lectures On DEmand) è un progetto realizzato dal professor Marco Ronchetti e collaboratori presso l'Università di Trento. Si presenta come una soluzione per l'acquisizione in formato video e audio di lezioni universitarie, con la particolarità di essere una soluzione a basso costo e facilmente manovrabile.[2]

Le lezioni registrate possono poi essere consultate tramite una pagina web appositamente generata, che combina il video del computer del docente, il video ripreso da una videocamera posizionata in aula e l'audio catturato da un microfono.

I vantaggi che questo sistema offre sono molteplici, tra cui: la possibilità per gli studenti-lavoratori di seguire le lezioni in remoto a qualsiasi orario; la possibilità di recuperare le lezioni in caso di assenze non volontarie (es. malattia); supporto per gli studenti che non comprendono bene la lingua del corso; la possibilità di rivedere porzioni specifiche di qualsiasi lezione in qualsiasi momento, e di verificare quindi la qualità dei propri appunti e il livello di comprensione.

Le versioni più recenti di LODE prevedono anche un'interfaccia web utilizzabile dagli studenti durante lo svolgimento della lezione. Questo strumento permette di catturare degli screenshot di quanto proiettato dal docente in quell'istante, e di scrivere o disegnare annotazioni sulle catture. In questo modo gli studenti sono coinvolti in modo meno passivo e seguono la lezione con più attenzione.

1.2 La soluzione LodeBox

L'ultima versione di LODE prevede l'utilizzo di una board Raspberry Pi per eseguire il software di acquisizione. Il dispositivo è inserito in una piccola scatola che espone dei connettori verso l'esterno. Tra questi sono presenti un ingresso HDMI per collegare un computer e un'uscita HDMI per collegare un proiettore o uno schermo. Le porte USB permettono di collegare un telecomando e un "dongle" per l'acquisizione dell'audio di un microfono con jack 3,5 mm.

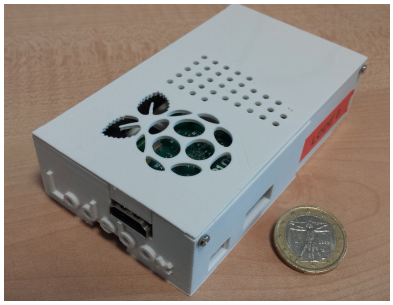

Figura 1.1: La scatola di LodeBox.

La scatola include un modulo "HDMI to MIPI CSI-2", che permette di convertire il segnale HDMI nel formato seriale della fotocamera. Il video HDMI è infatti acquisito utilizzando la libreria `PiCamera` per Python, che permette di acquisire il segnale come se si trattasse del video di una fotocamera. La libreria semplifica di molto lo sviluppo, perché espone delle funzioni di alto livello per svolgere operazioni come abilitare l'anteprima del video a schermo intero, la registrazione con un encoder H.264[1] con accelerazione hardware e la cattura di screenshot.

Sempre in ottica di riduzione dei costi, LodeBox prevede la ripresa del docente tramite una qualsiasi videocamera IP (wireless o cablata) che supporti il protocollo RTSP (Real Time Streaming Protocol). La registrazione avviene sul dispositivo utilizzando `ffmpeg`/`avconv`[2] in modalità copia (senza codifica), richiedendo quindi un uso molto basso di risorse.

Raspberry Pi non prevede spazio di archiviazione interno, e richiede quindi di utilizzare una scheda microSD per memorizzare le registrazioni. Le memorie di tipo flash hanno però vita limitata, a seconda di quanto intensivamente sono utilizzate, e questo riduce di conseguenza l'affidabilità a lungo termine del sistema.

Un altro svantaggio di questa soluzione è il livello di "fault tolerance" in caso di eventi come la disconnessione (volontaria o meno) del cavo HDMI in ingresso. Dato che l'input HDMI è mappato sull'interfaccia della fotocamera, non è previsto che la fotocamera possa essere improvvisamente scollegata. Secondo le prove effettuate, risulta molto difficile rilevare in modo affidabile la disconnessione del cavo HDMI. Nei casi in cui è possibile, non risulta invece fattibile il recupero dell'applicazione,

[1] H.264, conosciuto anche come AVC o MPEG-4 Part-10, è il più popolare codec di compressione video. Nato nel 2003, è ancora oggi lo standard de facto in numerosi ambiti, tra cui lo streaming video.

[2] *ffmpeg* è uno strumento estremamente popolare per l'elaborazione di video e audio tramite linea di comando. Supporta numerosi formati, codec e protocolli, e permette di effettuare operazioni quali il muxing, transmuxing, transcoding e altre funzioni più avanzate. *avconv* è un fork di ffmpeg nato per via di divergenze sui metodi di sviluppo, ed è stato incluso per un breve periodo in alcune distribuzioni Linux in sostituzione di ffmpeg.

perché le operazioni sulla `PiCamera` sollevano eccezioni o bloccano indefinitamente l'esecuzione del codice.

Infine, un altro problema è posto dalla presenza del modulo di conversione HDMI-CSI, la cui disponibilità e compatibilità a lungo termine non sono garantite.

1.3 Hardware alternativo

Per poter evolvere LodeBox in una soluzione più affidabile e adatta alla produzione e distribuzione, si sono quindi cercate SBC (Single-Board Computer) alternative, preferibilmente pensate per la realizzazione di applicazioni multimediali. I principali vincoli per la scelta di una nuova scheda erano la presenza dell'input HDMI, la possibilità di collegare un microfono, e il costo accessibile.

Molte board con sistema operativo Android soddisfano questi requisiti, e consentono inoltre di avere a disposizione una piattaforma nota, documentata, di facile sviluppo e per cui sono disponibile molte risorse. In molti casi l'input HDMI è reso disponibile tramite l'interfaccia della fotocamera CSI, da cui deriva la possibilità di acquisire l'input tramite le API di Android per l'accesso alla fotocamera.

2 Acquisizione video HDMI

Nelle sezioni che seguono vengono introdotti i principali metodi tipicamente disponibili su board Android per l'accesso all'ingresso HDMI. Si tratta in particolare delle due API per l'accesso alla fotocamera, a cui ci si riferisce spesso come "Camera1" e "Camera2", e talvolta di SDK forniti dal produttore.

2.1 Le API `android.hardware.Camera`

La classe `Camera` è stata introdotta nella prima versione di Android ed è il metodo più semplice per accedere alla fotocamera. Va notato però che la classe è stata messa in stato di deprecazione a partire da Android 5.0 (livello API 21), e Google consiglia di utilizzare invece le API `Camera2` per realizzare nuove applicazioni.[1] Tuttavia, non tutti i dispositivi con Android 5.0 o superiore supportano nativamente le API `Camera2`. Esiste infatti una modalità `LEGACY` che dà la possibilità di usare le API `Camera2` nonostante l'implementazione sia in realtà la stessa delle API Camera1.

Entrando nel merito, per mostrare il video acquisito sullo schermo è prima di tutto necessario aggiungere un nuovo controllo nella `Activity`[2] dell'applicazione Android. Si utilizza per questo un elemento chiamato `SurfaceView`, che permette di disegnare contenuti su schermo in modo ottimizzato, fluido e senza consumare le risorse del thread della grafica. Quando una `SurfaceView` diventa visibile su schermo, una callback informa del fatto che la sottostante `Surface` è stata creata. È possibile a questo punto accedere, tramite il metodo d'istanza `getHolder()`, a un'implementazione dell'interfaccia `SurfaceHolder`, che permette di configurare e controllare la superficie di disegno.

Ottenuto il `SurfaceHolder`, si può procedere con la creazione di un'istanza di `Camera` tramite il metodo statico `Camera.open()`. Si collega quindi la superficie alla `Camera` tramite il metodo `setPreviewDisplay(SurfaceHolder)`. Infine, una chiamata a `startPreview()` permette di avviare l'anteprima su schermo.

A questo punto, creando un'istanza della classe `android.media.MediaRecorder` è possibile registrare il video acquisito, mostrando allo stesso tempo l'anteprima su schermo. Più in dettaglio, si tratta di seguire una rigida sequenza di passaggi[3] per configurare correttamente il `MediaRecorder` e collegare l'input `Camera` al registratore, tramite il metodo `setCamera(Camera)`. La registrazione può infine essere avviata con il metodo `start()`.

[1] https://developer.android.com/reference/android/hardware/Camera

[2] Una `Activity` rappresenta una pagina dell'applicazione.

[3] Documentati qua: https://developer.android.com/guide/topics/media/camera#capture-video

La cattura di un singolo screenshot si può invece ottenere tramite il metodo `takePicture(...)`, esposto dalla classe `Camera`. Questo metodo permette di ricevere tramite delle callback i dati del fotogramma catturato, sia in formato non compresso che in JPEG.

Questo metodo presenta però un possibile effetto collaterale, riscontrato durante i test, ossia la possibilità che alla cattura di un fotogramma l'anteprima del video su schermo venga interrotta. Durante lo sviluppo si è quindi optato per una modalità alternativa per l'acquisizione degli screenshot: è possibile infatti aggiungere una callback alla `Camera` tramite il metodo `setPreviewCallback`, e ricevere così a flusso continuo una copia di ogni fotogramma in formato non compresso[4].

Sorge a questo punto un altro problema, cioè la necessità di convertire l'immagine in formato JPEG, per permettere l'upload dello screenshot sul server. La piattaforma Android offre una classe `YuvImage` con un metodo `compressToJpeg(...)`, che soffre però di un memory leak che porta dopo qualche minuto al crash dell'applicazione[5], come verificato nei test svolti. Il bug è stato risolto solo in Android 9.0[6], rendendo quindi necessaria un'alternativa. Si è optato per una soluzione che prevede la conversione in modo ottimizzato dal formato "non compresso" (NV21) a `Bitmap`[7], e successivamente in JPEG tramite il metodo `compress(...)` offerto da `Bitmap` (che non soffre di memory leak). Nonostante il numero di passaggi, il risultato si è rivelato molto soddisfacente dal punto di vista delle prestazioni.

Il seguente estratto di codice mostra la conversione da NV21 a JPEG. La variabile `context` è l'`Activity` dell'applicazione, mentre `data` contiene i dati dell'immagine sorgente. Alla riga 4 è possibile configurare la qualità dell'immagine JPEG prodotta (in questo caso 80 su 100).

```
1 RenderScript rs = RenderScript.create(context);
2 Bitmap bitmap = Nv21Image.nv21ToBitmap(rs, data, width, height);
3 ByteArrayOutputStream stream = new ByteArrayOutputStream();
4 boolean ok = bitmap.compress(Bitmap.CompressFormat.JPEG, 80, stream);
```

2.2 Le API `android.hardware.camera2.*`

A partire da Android 5.0, un nuovo insieme di classi permette una gestione più avanzata della fotocamera. L'equivalente della classe `Camera` è `CameraDevice`, che però non espone più i metodi visti nella sezione precedente.

Viene introdotto invece il concetto di pipeline, cioè flussi paralleli forniti dal sistema operativo per diversi scopi. Ad esempio, la pipeline di "preview" fornisce un flusso con latenza molto bassa ma trascurando leggermente la qualità dell'immagine. La pipeline per le foto fornisce un'immagine ad alta qualità, mentre la pipeline per la registrazione video offre qualità dell'immagine e scansione costante dei fotogrammi in uscita.[8]

Ottenuto il `CameraDevice`, la prima operazione da eseguire è creare una sessione di cattura, alla quale vanno collegati i *target* di acquisizione (metodo `createCaptureSession(...)`). Android utilizza questi target per configurare le pipeline interne e allocare i buffer necessari per la produzione dei fotogrammi.[3] I target possono essere `SurfaceView`, oppure istanze di `ImageReader`, che ricevono i fotogrammi singoli. La sessione creata è immutabile, e cioè non è possibile modificarne i target se non creandone una nuova.

Una volta configurata la sessione, il metodo `createCaptureRequest(int)` esposto dalla classe `CameraDevice` permette di inviare una richiesta di cattura alla fotocamera. Questo meccanismo permette di catturare in qualsiasi istante un fotogramma, che avrà caratteristiche diverse a seconda della pipeline scelta come parametro (per esempio `TEMPLATE_PREVIEW`, `TEMPLATE_STILL_CAPTURE`, `TEMPLATE_RECORD`, ecc.).

[4] Si tratta in realtà del formato NV21, cioè una variante di Android di YCbCr 4:2:0, che è una codifica del colore lossy anche se con una perdita di dettaglio solitamente impercettibile.

[5] https://codar.club/blogs/android-system-api-yuvimage-compresstojpeg-has-a-memory-leak-at-the-native-level.html

[6] Il commit del fix è disponibile qua: https://android.googlesource.com/platform/frameworks/base/+/1c3ded3

[7] Libreria `easyRS`: https://github.com/silvaren/easyrs

[8] *Build a universal camera app (Google I/O '18)* https://youtu.be/d1gLZCSLmaA

Per mostrare l'anteprima su schermo si può utilizzare il metodo `setRepeatingRequest(...)`, che si occupa di inviare richieste di cattura il più frequentemente possibile, con l'effetto di avere il video mostrato in tempo reale su schermo.

Quando si vuole iniziare la registrazione del video, è necessario interrompere la cattura di sessione e crearne una nuova, aggiungendo tra i *target* una `Surface` legata a un'istanza di `MediaRecorder` (il metodo `getSurface()` permette di ottenere l'oggetto). In questo modo la richiesta di cattura invierà i fotogrammi al `MediaRecorder`, che si occuperà di passarli all'encoder e di scriverli su file.

2.3 SDK alternativi

Alcuni produttori forniscono degli SDK (Software Development Kit) specifici per utilizzare l'ingresso HDMI su un determinato hardware. In questi casi, le API `Camera` di Android potrebbero non essere disponibili.

Inoltre, è spesso possibile sfruttare i `BroadcastReceiver` di Android per ricevere dal sistema eventi come la disconnessione e la riconnessione del cavo HDMI, e gestire quindi la sospensione e la ripresa dell'anteprima su schermo ed eventualmente della registrazione del video. Il seguente blocco di codice mostra come registrare un `BroadcastReceiver` per ricevere determinati eventi. Il `receiver` deve poi essere de-registrato alla sospensione dell'applicazione.

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    BroadcastReceiver receiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            // Lettura stato di connessione. Varia in base all'hardware
            boolean isConnected = intent.getBooleanExtra("state", false);
            // Gestire qua connessione/disconnessione
        }
    };

    IntentFilter filter = new IntentFilter("android.intent.action.HDMI_PLUGGED");
    super.registerReceiver(receiver, filter);
}
```

3 Acquisizione video RTSP

Come anticipato, il sistema LODE prevede la possibilità di registrare con una videocamera ciò che avviene in aula, in modo da rendere più coinvolgente la fruizione delle lezioni. Per limitare i costi, la videocamera è di tipo IP (Internet Protocol), e permette lo streaming locale in tempo reale tramite il protocollo RTSP.

3.1 Il protocollo RTSP

RTSP (Real Time Streaming Protocol) è un protocollo di rete per il controllo di flussi multimediali. Si dice che è *di controllo* perché non è usato per lo scambio di dati multimediali, ma di messaggi con lo scopo di richiedere determinate operazioni al server. Per fare qualche esempio, è possibile ottenere informazioni sui flussi disponibili, riprodurre un flusso specifico o metterlo in pausa.[1]

Il trasferimento vero e proprio dei dati multimediali avviene invece tramite RTP (Real-time Transport Protocol), un protocollo progettato per consentire il trasporto in tempo reale di contenuti video

e audio. Le implementazioni di RTP sono tipicamente basate su UDP[1], protocollo di livello trasporto non connesso e non affidabile particolarmente adatto per situazioni in cui la latenza è più importante dell'affidabilità, ma spesso offrono compatibilità anche con TCP[2].

3.2 Registrazione con `ffmpeg`

Un flusso video RTSP può essere facilmente registrato tramite lo strumento `ffmpeg`, un progetto open source che incorpora il supporto a numerosi protocolli, formati e codec per l'elaborazione del video e dell'audio. Il video acquisito può essere salvato anche in modalità copia, e cioè salvando il *bitstream* ricevuto (es. formato H.264) senza nessuna elaborazione. Questo permette di evitare la ricodifica del video e di risparmiare risorse hardware.

Può essere inoltre scelto un qualsiasi formato contenitore (es. MP4) compatibile con il codec del video. L'operazione di inserire un *bitstream* in un contenitore si chiama *muxing*, ed è molto leggera a livello di CPU. Una scelta conveniente per il contenitore può essere MPEG-2 Transport Stream, un formato pensato per sistemi di distribuzione che soffrono di perdita di dati (es. la televisione digitale terrestre) e tollerante a interruzioni forzate del muxing, come nel caso di mancanza di corrente.

Il comando seguente acquisisce il video da una videocamera IP e lo salva in un file `out.ts`:

```
> ffmpeg -i rtsp://admin:admin@192.168.178.30:88/videoMain
         -c:v copy -an out.ts -y
```

In alternativa a MPEG-2 TS si può scegliere anche MPEG-2 PS (Program Stream), che ha un overhead di muxing inferiore per via dell'assenza di controllo degli errori a livello di contenitore:

```
> ffmpeg -i rtsp://admin:admin@192.168.178.30:88/videoMain
         -c:v copy -an -f vob out.mpg -y
```

In una applicazione Android, è possibile utilizzare dei wrapper appositamente realizzati per sfruttare `ffmpeg` sulle architetture tipiche di Android (tra cui ARM). Per le sperimentazioni è stata presa in considerazione la libreria open source `MobileFFmpeg`[3], anche per via dell'ottimo stato di mantenimento e aggiornamento del progetto.

`MobileFFmpeg` è fornita in diverse varianti, a seconda delle versioni di Android che si desidera supportare e dei moduli di `ffmpeg` di cui si necessita. Per fare un esempio, il pacchetto identificato come `com.arthenica:mobile-ffmpeg-min:4.2.LTS` è una versione base (`min`) che non include il supporto a nessuna libreria esterna (nessun encoder/decoder), ma che comprende comunque il supporto a RTSP e al muxing. `LTS` sta a indicare che la versione minima di Android supportata è 4.1, a differenza della versione non LTS che richiede Android 7.0 o superiore.

4 Realizzazione della modalità "kiosk"

Nell'ambito dei sistemi *embedded* si utilizza spesso la locuzione *modalità kiosk* per indicare tutte le situazioni in cui il sistema deve comportarsi come un "chiosco digitale" e limitare l'utilizzo a specifiche funzioni. Su Android si possono adottare diverse tecniche per ottenere una modalità simile, nascondendo quindi la presenza del sistema operativo Android. In questo capitolo sono presentate le tecniche sperimentate, tra cui la modalità a schermo intero, la modifica dell'*overscan* di sistema, la modalità *lock task*, l'avvio automatico dell'applicazione e la modifica dell'animazione di avvio del sistema.

[1] User Datagram Protocol
[2] Transmission Control Protocol
[3] https://github.com/tanersener/mobile-ffmpeg

4.1 La modalità a schermo intero

A partire da Android 4.1, è stata introdotta una gestione granulare della modalità a schermo intero. Si tratta di base della possibilità di nascondere, utilizzando appositi *flag*, la barra di stato e la barra di navigazione di Android, presenti rispettivamente nella parte superiore e inferiore dello schermo.

```
private void goFullScreen() {
    getWindow().getDecorView().setSystemUiVisibility(
        View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_FULLSCREEN);
}
```

Questo metodo va chiamato sia all'avvio dell'applicazione che nei casi in cui la modalità a schermo intero potrebbe essere automaticamente disabilitata, ovvero quando l'applicazione perde e poi riacquisisce il "focus".

```
@Override
public void onResume(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    goFullScreen();
}

@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (hasFocus) {
        goFullScreen();
    }
}
```

Questa soluzione ha l'effetto di nascondere le due barre di sistema, ma solo fintantoché l'utente non preme un qualsiasi punto dello schermo. Per questo Google ha introdotto anche una "modalità immersiva", che permette di mantenere l'applicazione a schermo intero fino a quando l'utente non scorre dai bordi dello schermo.[1]

```
private void goFullScreen() {
    getWindow().getDecorView().setSystemUiVisibility(
        // Nasconde barra di navigazione e di stato
        View.SYSTEM_UI_FLAG_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_FULLSCREEN
        // Modalità immersiva
        View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY
        // Evita lo spostamento del layout della pagina
        | View.SYSTEM_UI_FLAG_LAYOUT_STABLE
        | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
        | View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN);
}
```

4.2 L'*overscan* di sistema

Come accennato, la modalità a schermo intero di Android permette comunque all'utente di usare le barre di sistema scorrendo dai lati dello schermo, e quindi potenzialmente di uscire dall'applicazione. In un sistema embedded questo non è desiderabile, ed esiste quindi un metodo alternativo per impedire che le barre di sistema vengano mostrate.

[1] https://developer.android.com/training/system-ui/immersive

La soluzione prevede l'utilizzo della funzione *overscan* del servizio di sistema `WindowManager`, il quale si occupa di gestire la visualizzazione delle finestre, la rotazione dello schermo, le animazioni, le transizioni, ecc.

L'overscan di sistema permette di modificare l'area di disegno delle finestre, o in altre parole i margini dello schermo. Impostando un margine negativo l'area di disegno dell'interfaccia di Android sarà estesa oltre l'area visibile dello schermo, con l'effetto di tagliare porzioni dell'interfaccia, come mostrato nella figura 4.1.

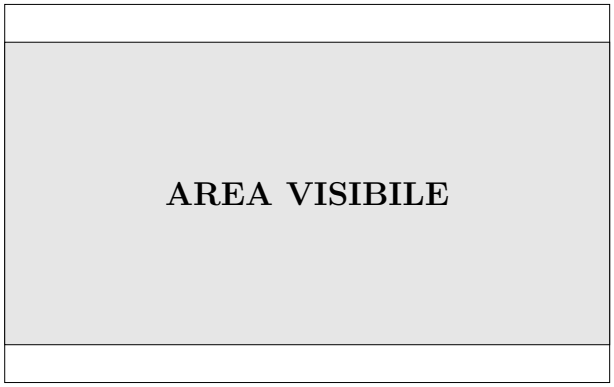

Figura 4.1: Rappresentazione dell'overscan di sistema. La finestra sconfina l'area visibile.

Nell'esempio seguente viene utilizzato il comando `wm` (`WindowManager`) della shell di Android per impostare un margine negativo al lato superiore e inferiore dello schermo, in modo da nascondere barra di stato e di navigazione, alte in questo caso 25dp e 48dp[2] (i valori possono essere diversi a seconda del dispositivo).

```
> adb shell wm
usage: wm [subcommand] [options]
wm size [reset|WxH|WdpxHdp]
wm density [reset|DENSITY]
wm overscan [reset|LEFT,TOP,RIGHT,BOTTOM]
wm scaling [off|auto]
wm screen-capture [userId] [true|false]

> adb shell wm overscan 0,-25,0,-48
```

Una volta configurato l'overscan, l'impostazione dovrebbe essere memorizzata in modo permanente e sopravvivere al riavvio del sistema. Tuttavia, durante le sperimentazioni si è verificato almeno una volta che l'overscan non fosse ripristinato in automatico. Potrebbe quindi essere opportuno eseguire il comando ad ogni avvio del sistema, anche avviando un processo dall'applicazione stessa.

Per completezza, l'overscan può essere disabilitato con questo comando:

```
> adb shell wm overscan reset
```

4.3 La modalità "lock task"

In aggiunta ai metodi illustrati nei capitoli precedenti, Android fornisce a partire dalla versione 5.0 una modalità chiamata *lock task*, che fa parte di un insieme più ampio di API per la realizzazione di dispositivi dedicati (in passato chiamati COSU, Corporate-Owned Single-Use). Queste API fanno a loro volta parte di Android Enterprise, che propone soluzioni specifiche per casi d'uso aziendali.

Quando la modalità lock task viene abilitata, il sistema operativo entra in una modalità rigida che permette di usare solo un insieme di applicazioni definite manualmente tramite una *whitelist*. Inoltre, la barra di stato viene disabilitata, le notifiche sono soppresse, e l'utilizzatore non può navigare nel sistema al di fuori delle app inserite in whitelist.[3]

[2] Density-independent Pixels, un'unità di misura che tiene in considerazione la densità dello schermo

[3] https://developer.android.com/work/dpc/dedicated-devices/lock-task-mode

Per implementare la modalità lock task, sono necessari due componenti: un Device Policy Controller (DPC) e una `Activity` da lanciare. Il DPC deve essere un'applicazione "proprietaria del dispositivo", e cioè deve avere dei privilegi speciali per modificare alcune funzioni di sistema. Ha infatti il compito di configurare la modalità lock task elencando i nomi dei *package* da inserire in whitelist.

La procedura per la realizzazione di un DPC è articolata ed è dettagliata nella documentazione di Android[4], ma è sufficiente sapere che viene fornito un metodo per scegliere quali applicazioni possono essere eseguite in modalità lock task:

```
DevicePolicyManager dpm =
    (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);
ComponentName adminName = getComponentName(this);
dpm.setLockTaskPackages(adminName, new String[] { "it.unitn.lode" });
```

Google fornisce inoltre un'applicazione chiamata "Test DPC" che implementa numerose funzionalità legate alle policy aziendali, tra cui la configurazione della whitelist. L'applicazione è particolarmente utile durante lo sviluppo, perché permette di effettuare test rapidamente senza sviluppare un DPC. L'installazione è semplice e prevede di impostare "Test DPC" come proprietario del dispositivo:[5]

```
> adb install TestDPC.apk
> adb shell dpm set-device-owner com.afwsamples.testdpc/.DeviceAdminReceiver
```

A questo punto, la `Activity` che vuole entrare in modalità lock task può semplicemente chiamare il metodo `startLockTask()`:

```
@Override
public void onResume() {
    super.onResume();

    DevicePolicyManager dpm =
        (DevicePolicyManager) getSystemService(Context.DEVICE_POLICY_SERVICE);

    if (dpm.isLockTaskPermitted(getPackageName())) {
        startLockTask();
    }
}
```

[IMMAGINE LOCK TASK MODE]

Se ben configurata, questa soluzione, combinata con le tecniche mostrate nei capitoli precedenti, permette di avere un'applicazione a schermo intero da cui è impossibile uscire.

4.4 L'avvio automatico dell'applicazione

Una `Activity` può essere configurata per sostituire la schermata "home" di Android, ed essere quindi la prima ad essere mostrata all'avvio del sistema.

Il blocco di codice seguente mostra una porzione del file `AndroidManifest.xml`, tramite il quale è stato forzato il fatto che possa esistere una sola istanza alla volta di `MainActivity`, e che questa debba agire come attività "home" predefinita:

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

[4] https://developer.android.com/guide/topics/admin/device-admin.html#developing
[5] https://codelabs.developers.google.com/codelabs/cosu/index.html#6

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

4.5 L'animazione di avvio

Come ultimo aspetto, un requisito di un sistema embedded potrebbe essere quello di non mostrare i marchi di Android o del produttore dell'hardware. Fortunatamente, Android permette in modo abbastanza facile di sostituire l'animazione di avvio del sistema con una a piacere.[6]

In fase di avvio, il sistema legge il file `/system/media/bootanimation.zip` ed estrae una descrizione dell'animazione (`desc.txt`) e un insieme di file PNG rappresentanti i fotogrammi da mostrare in sequenza su schermo.

Per fare un esempio, si prenda come riferimento questo file di descrizione:

```
400 200 10
c 0 0 part0 #ffffff
```

La prima riga indica parametri generali dell'animazione, in particolare la larghezza, l'altezza e il numero di fotogrammi al secondo da renderizzare.

La seconda riga indica invece un blocco di fotogrammi che devono essere eseguiti secondo precise regole. In particolare:

- la lettera `c` indica che l'animazione verrà eseguita fino al suo completamento, anche nel caso in cui il sistema sia pronto prima;
- la prima occorrenza del numero `0` indica quante volte l'animazione deve essere ripetuta, in questo caso infinite volte, mentre la seconda il numero di fotogrammi di attesa prima della riproduzione del blocco successivo;
- `part0` è il nome della cartella in cui trovare la lista di file che compongono l'animazione;
- infine, l'ultimo parametro determina il colore di sfondo nel caso in cui l'animazione sia trasparente o non copra l'intero schermo.

I file ottenuti possono quindi essere inseriti in un archivio ZIP senza compressione, con un comando simile a questo:

```
zip -0qry -i \*.txt \*.png @ ../bootanimation.zip *.txt part*
```

Infine, il file `bootanimation.zip` va caricato sul dispositivo Android tramite `adb` e la modalità debug con accesso root:

```
adb root
adb remount
adb push bootanimation.zip /system/media
adb reboot
```

[6] https://android.googlesource.com/platform/frameworks/base/+/master/cmds/bootanimation/FORMAT.md

5 Sospensione della registrazione

Lorem ipsum dolor sit amet.

6 Sincronizzazione di video e audio

Lorem ipsum dolor sit amet.

6.1 Definizione del problema

Lorem ipsum dolor sit amet.

6.2 Proposta di soluzione

Lorem ipsum dolor sit amet.

6.3 Implementazione e sperimentazione

Lorem ipsum dolor sit amet.

7 Rilevamento delle differenze tra fotogrammi

Lorem ipsum dolor sit amet.

7.1 In post-produzione

Lorem ipsum dolor sit amet.

7.2 In tempo reale

Lorem ipsum dolor sit amet.

7.2.1 Implementazione e sperimentazione

Lorem ipsum dolor sit amet.

8 Acquisizione audio (forse)

Lorem ipsum dolor sit amet.

9 Conclusione

Lorem ipsum dolor sit amet.

Bibliografia

[1] Real Time Streaming Protocol (RTSP). RFC 2326, RFC Editor, aprile 1998.

[2] Ronchetti Marco. Strumenti per favorire l'apprendimento nei primi anni di università. Parte 2: Il sistema LodeBox. In Raffone Alessandra, editor, *La città educante. Metodologie e tecnologie innovative a servizio delle smart communities*, pages 137–146. Liguori, 2018.

[3] Wahltinez Oscar. Understanding Android camera capture sessions and requests. https://medium.com/androiddevelopers/understanding-android-camera-capture-sessions-and-requests-4e54d9150295, settembre 2018. Ultimo accesso 25/06/2019.

Allegato A Titolo primo allegato

Lorem ipsum dolor sit amet.