ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UN SISTEMA DI VIDEO STREAMING IN AMBITO EYEWEAR E WEARABLE COMPUTING

Elaborato in
PROGRAMMAZIONE DI SISTEMI EMBEDDED

Relatore
Prof. ALESSANDRO RICCI

Co-relatore
Ing. ANGELO CROATTI

Presentata da
DAVIDE GIACOMINI

Seconda Sessione di Laurea
Anno Accademico 2016 – 2017

PAROLE CHIAVE

Wearable computing

Video streaming

MPEG-DASH

FFMpeg

RTSP

Alla mia famiglia

Indice

Introduzione

L'introduzione di sistemi portabili, quali smartphone e tablet, sempre più performanti e sempre più ricchi in termini di funzionalità, ha aperto nuovi modi di utilizzare la tecnologia. Ad ora è uno dei settori più attivi ed in costante crescita, anche grazie al fatto che questa tipologia di dispositivi è sempre più richiesta ed utilizzata. Negli ultimi anni però, sta prendendo piede una nuova categoria, quella del **Wearable computing**, i cosiddetti "dispositivi indossabili", che presentano, oltre alle stesse caratteristiche dei dispositivi attuali, nuove possibilità di interazione per l'utente, rendendoli molto più "smart". Un esempio di ciò è il fatto che, grazie alla grande qualità di sensori, gli utenti possono impartire ordini al dispositivi in modo *hands-free*, ovvero senza dover per forza interagire fisicamente col dispositivo, attraverso gestures o comandi vocali. Grazie a ciò, diventa possibile utilizzare i sistemi wearable mentre vengono eseguite altre attività. Inoltre, facendo interagire questi ultimi con altri sistemi embedded o distribuiti, si concretizza il concetto di **Ubiquitous computing**, ovvero l'idea secondo cui la computazione possa essere resa pervasiva e costante in ogni attività quotidiana.

Queste potenzialità possono quindi essere molto utili in ambito medico, dove la reattività, la velocità di intervento e l'esperienza risultano essere aspetti cruciali, soprattutto per quanto riguarda il primo soccorso. Ed è proprio in questo ambito che l'informatica può fornire un importante contributo al fine di rendere più efficaci ed efficienti tutti i processi ospedalieri, fornendo sistemi in grado di supportare gli operatori sanitari in tutte le attività critiche in cui sono coinvolti. Proprio per questo scopo è nato il progetto *Trauma Tracker*, sviluppato dall'Università di Bologna grazie alla collaborazione dell'ospedale M. Bufalini di Cesena, che verrà utilizzato come caso di studio del sistema che dovrà essere sviluppato.

Nello specifico, si andranno a considerare dispositivi appartenenti ad una precisa categoria, quella dei cosiddetti *eyewear*, come per esempio gli **smartglasses**, che offrono la possibilità di registrazione audio e video tramite videocamera e microfono. L'obiettivo preposto infatti è quello di sfruttare questa particolare caratteristica, creando un framework per il video streaming che supporti questi dispositivi, con capacità limitate, fornendo un meccanismo per

la visione live o posticipata delle registrazioni ad utenti remoti. Esso verrà progettato tenendo in considerazione le richieste specifiche richieste dal team di sviluppo del progetto TraumaTracker e dai medici coinvolti, tuttavia si cercherà di renderlo il più generico possibile in modo tale che possa essere utilizzato in un qualsiasi scenario.

Capitolo 1

Wearable Computing e Streaming

In questo capitolo verrà introdotto il concetto di *wearable computing*, con particolare enfasi sulla sottocategoria **eyewear**, analizzandone le caratteristiche e introducendo alcuni dispositivi di questa categoria, i loro scenari di utilizzo e i loro principali pregi e difetti. Verrà inoltre analizzato il significato del termine streaming e webcasting, facendo anche una panoramica sui protocolli maggiormente utilizzati in questo ambito.

1.1 Introduzione

Grazie al continuo sviluppo elettronico ed informatico sempre più dispositivi sono ora in grado di elaborare informazioni dal mondo reale, comunicare tra di loro e fornirci supporto in molteplici ambiti. Con la continua miniaturizzazione dei circuiti integrati è stato possibile estenderne la capacità di elaborazione e fornire nuovi sensori e attuatori che permettono ai dispositivi di percepire i fenomeni nella realtà e agire in essa.

Si è iniziato quindi a parlare di *Ubiquitous computing* che è anche descritto come *Pervasive computing*, per indicare appunto il modello di interazione uomo-macchina in cui l'elaborazione delle informazioni è stata interamente integrata in oggetti ed attività di tutti i giorni.

1.1.1 Eyewear e Wearable Computing

Un esempio concreto delle nozioni descritte in precedenza può essere il **Wearable computing**, che si può definire come l'insieme di tutti quei *sistemi embedded* "indossabili", il cui utilizzo introduce nuovi concetti e paradigmi

di sviluppo. Si può inoltre estendere questa definizione aggiungendo che questi sistemi possono essere visti anche come estensione dell'utilizzatore, grazie ai quali è possibile arricchire la percezione e l'interazione con il mondo circostante, sovrapponendovi per esempio informazioni digitali, come nel caso dell' *Augmented reality* [5].

Di particolare interesse per questo progetto risulta essere la parte dell'**Eyewear computing**, che si può definire come quella categoria dei sistemi *head-mounted*, nello specifico gli smartglasses, la maggior parte dei quali integrano videocamera e microfono, grazie ai quali è possibile registrare contenuti audio e video ad alta definizione, aspetto fondamentale per questo progetto che si pone appunto l'obiettivo di abilitare, per tutti questi sistemi, lo streaming efficiente di questi contenuti nel web.

1.1.2 Webcasting e Streaming

Prima di introdurre qualche esempio di dispositivo, si illustri il significato dei termini webcasting e streaming.

Nell'ambito delle telecomunicazioni, il termine **webcast** descrive la trasmissione di segnale audio e/o video sul web, attraverso la rete Internet, in modalità client-server. L'invio di questi dati può essere eseguito in tempo reale o ritardato.

D'altro canto, lo **streaming** identifica un flusso di dati audio e/o video trasmessi da una sorgente a più destinazioni tramite una rete telematica. I dati trasmessi in questo caso sono riprodotti man mano che giungono alla destinazione. In generale per la trasmissione vengono utilizzati i protocolli RTP e RTSP a livello di applicazione, mentre per il livello di trasporto si utilizza molto di frequente il protocollo UDP, poiché favorisce la velocità di trasferimento sacrificando l'affidabilità, che in questa particolare applicazione, risulta meno rilevante.

La distribuzione dei dati di streaming avviene tramite uno dei seguenti metodi:

- *Unicast*: i dati vengono inviati dal server ad un solo dispositivo, che deve occuparsi di ritrasmetterli agli altri client
- *Multicast*: i dati vengono inviati dal server a più dispositivi simultaneamente
- *HTTP*: i dati vengono inviati dal server al client sfruttando il protocollo a livello applicativo HTTP
- *P2P*: peer-to-peer, anche i client possono fungere da server, a patto che abbiano già ricevuto i dati

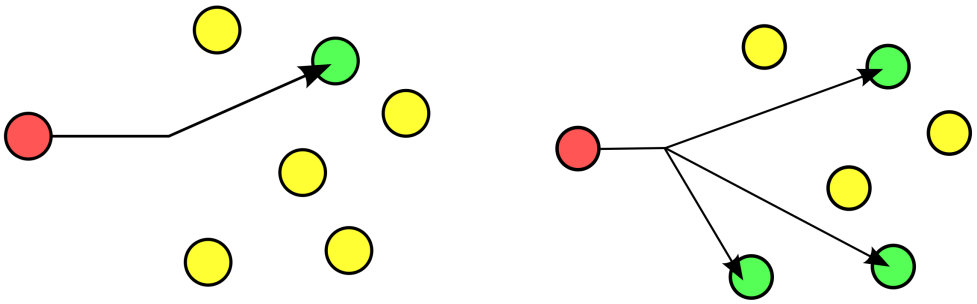


Figura 1.1: Schema comunicazione di tipo Unicast (a sinistra) e Multicast (a destra)

Uno streaming può essere di due tipologie, **on demand**, se il contenuto multimediale è completo e disponibile in un server come file, oppure **live**, nel caso in cui il contenuto sia ancora in fase di generazione da una sorgente esterna. In entrambi i casi però, il client non ha necessità di scaricare l'intero contenuto per poterlo riprodurre.

Generalmente, per diminuire il traffico dati, il contenuto è compresso, e quando il client ha necessità di riprodurlo, deve prima operare una decompressione utilizzando il codec specifico del formato in cui sono compressi i dati.

Per evitare l'interruzione del playback, causati dalla latenza della rete, la riproduzione è quasi sempre posticipata, in attesa che il *buffer* del player venga sufficientemente riempito, ovvero che si ottengano abbastanza dati in memoria tali che, riproducendoli si abbia tempo a sufficienza per permettere il download di quelli successivi.

1.2 Protocolli di streaming

Si andrà ora ad analizzare i protocolli maggiormente utilizzati per lo streaming di contenuti multimediali.

Nella seguente immagine è presente una panoramica delle tecnologie attualmente disponibili per lo streaming in relazione alla latenza offerta.

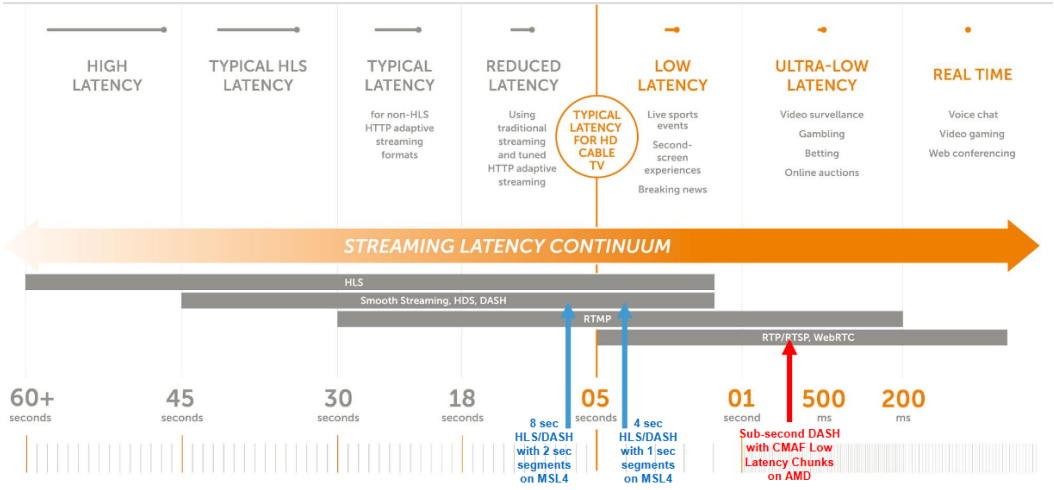


Figura 1.2: Panoramica protocolli per lo streaming multimediale

Come si vede chiaramente dalla figura [fig:1.1], per poter fornire contenuto con bassissima latenza o real-time, è necessario utilizzare i seguenti protocolli:

- *WebRTC*: tecnologia open source, sviluppato principalmente per audio e video chat, con una comunicazione di tipo browser-to-browser e peer-to-peer tramite websocket. È principalmente basato su Javascript e HTML5 [23]
- *RTMP*: protocollo proprietario di Adobe, necessita di Adobe Flash Player e lavora direttamente su TCP. Divide i dati in frammenti, la cui dimensione viene definita dinamicamente dal client e il server, a seconda delle condizioni della rete
- *RTSP*: utilizzato per la gestione di sessione di streaming tra server e client, utilizza RTP per il trasporto dei dati tramite UDP

Tuttavia questi protocolli non permettono di scalare adeguatamente al crescere dei client, soprattutto per quanto riguarda il playback dei contenuti, essendo pensati per uno scenario in cui è coinvolto un solo client che comunica ad un unico server. In questo caso quindi, per ottenere uno streaming scalabile

si ha la necessità di utilizzare tecniche di *adaptive bitrate streaming* (ABR), tutte basate esclusivamente su protocollo HTTP e progettate per distribuire dati in modo efficace ed efficiente su internet attraverso la frammentazione del contenuto multimediale.

Le principali sono:

- *HDS*: HTTP Dynamic Streaming, implementazione di Adobe, sfruttato da Flash Player e Flash Media Server
- *HLS*: Apple HTTP Live Streaming, implementazione di Apple, sfrutta il formato MPEG2-TS per dividere il contenuto in segmenti, accessibili attraverso
- *Smooth streaming*: estensione dei servizi multimediali IIS di Microsoft che abilita l'utilizzo dello streaming adattivo
- *MPEG-DASH*: Standard internazionale, creato per permettere un deployment universale per soluzioni ABR, presenta quasi tutte le funzionalità delle controparti proprietarie

La frammentazione del contenuto e la loro gestione rendono però questa tecnologia molto meno performante in termini di latenza. Tuttavia adottando alcune pratiche specifiche, è possibile raggiungere latenze accettabili di un paio di secondi.

Sono comunque già in sviluppo altre iterazioni dello standard MPEG-DASH, chiamato FDH, o DASH over Full Duplex HTTP-based Protocols, che si pone come obiettivo di fornire streaming a bassa latenza in grande scala, a conferma del fatto che la tecnologia DASH sembra essere molto promettente [24].

Idealmente quindi, la scelta del protocollo, per quanto riguarda il wearable computing sarebbe orientata verso RTSP, per quanto riguarda il webcasting dei contenuti da dispositivo wearable a server, dove è richiesta bassa latenza e la comunicazione risulta essere solo punto-punto, e DASH, per lo streaming dei dati dal server ai client web, poichè offre un'ottima scalabilità e una latenza accettabile.

In seguito sono descritte più nel dettaglio queste due tecnologie.

1.2.1 RTSP

Il Real Time Streaming Protocol, o RTSP, è un protocollo di rete pensato per il controllo della riproduzione di contenuto da server multimediali. Non definisce schemi per la compressione del flussi e delega le modalità di incapsulamento del bitstream al protocollo RTP. Non ha un protocollo di trasporto

definito (può utilizzare TCP o UDP indifferentemente) e non pone limitazioni su come il player debba riprodurre i flussi [09] [10].

In generale una interazione client-server su RTSP si compone dei seguenti eventi:

- **DESCRIBE**
  - *Client*: richiede al server di creare una nuova sessione
  - *Server*: risponde inviando la descrizione della sessione multimediale (in formato SDP)
- **SETUP**
  - *Client*: comunica in che porte resterà in ascolto per ricevere dati multimediali
  - *Server*: risponde comunicando quale porta userà per inviare i dati multimediali
- **PLAY**: comando inviato dal client per segnalare al server di avviare il trasferimento dei pacchetti di dati (RTP)
- **TEARDOWN**: comando inviato dal client per terminare la sessione

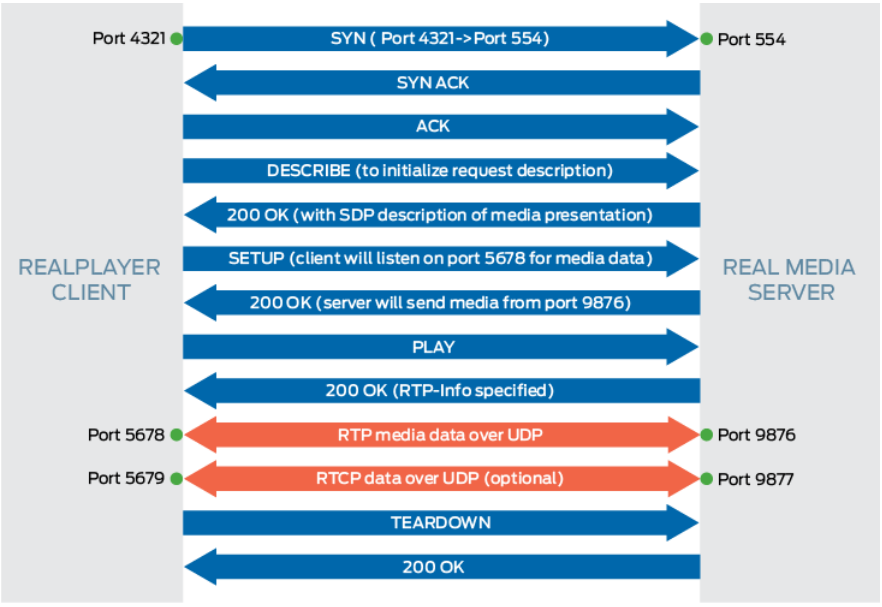


Figura 1.3: Un esempio di conversazione RTSP standard

1.2.1.1 SDP

SDP è l'acronimo di Session Description Protocol ed è un formato realizzato per descrivere sessioni multimediali. È utilizzato per descrivere i flussi audio e video che formano la sessione e le informazioni relative ciascuno di essi, contiene inoltre gli indirizzi di destinazione dei diversi stream.

Di seguito è riportato un esempio di SDP in una tipica sessione del framework sviluppato, in questo caso il server ha ricevuto la descrizione della sessione e dei flussi che il client è pronto ad inviare, nello specifico una stream video compressa in formato H.264 e una audio compressa in formato AAC [12].

```
v=0
o=- 0 0 IN IP4 127.0.0.1
s=Unnamed
i=N/A
c=IN IP4 192.168.1.2
t=0 0
a=recvonly
m=audio 5000 RTP/AVP 96
a=rtpmap:96 mpeg4-generic/16000/2
a=fmtp:96 streamtype=5; profile-level-id=15; mode=AAC-hbr;
    config=1410; SizeLength=13; IndexLength=3; IndexDeltaLength=3;
a=control:trackID=0
m=video 5002 RTP/AVP 96
a=rtpmap:96 H264/90000
a=fmtp:96 packetization-mode=1;
    sprop-parameter-sets=Z0KAH9oBQBboBtChNQ==,aM4G4g==;
a=control:trackID=1
```

1.2.1.2 RTP

Il Real Time Transport Protocol è un protocollo del livello di applicazione utilizzato per servizi di comunicazione real-time unicast o multicast. Si basa sul protocollo UDP e definisce il formato del pacchetto standard per inviare audio e video tramite reti IP ed è usato in congiunzione al protocollo RTCP (RTP Control Protocol), che ha essenzialmente la funzione di monitoraggio e controllo dei pacchetti RTP, il cosiddetto QoS, Quality of Service.

Il lato trasmittente di un' applicazione multimediale, il server di streaming, nel passaggio dal livello "Applicazione" a quello "Trasporto" crea un pacchetto RTP, con un blocco dati ed una intestazione, il quale viene a sua volta incapsulato in un pacchetto UDP prima dell'affidamento al protocollo IP

per l'instradamento. Il lato ricevente, il client, esegue l'operazione contraria estraendo il pacchetto RTP e inviando i dati contenuti al player [11].

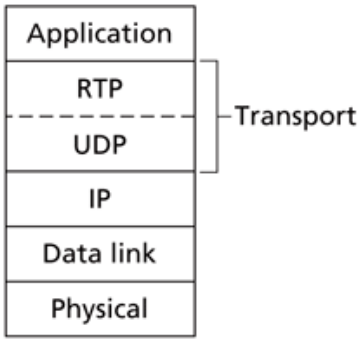


Figura 1.4: Collocazione del segmento RTP all'interno della pila ISO-OSI.

L'intestazione RTP contiene elementi importanti per ricostruire il file multimediale e per evitare ritardi nella trasmissione.

I quattro principali sono:

1. *Payload type*: indica la codifica utilizzata per la compressione dei dati
2. *Sequence number*: utilizzato per rilevare la perdita di pacchetti
3. *Timestamp*: l'istante di campionamento del primo byte nel pacchetto dati RTP, utile per sincronizzare il playback dei flussi audio e video e compensare il jitter (la continua variazione del ritardo di ricezione dei pacchetti)
4. *Source identifier*: indica la sorgente del flusso

1.2.2 MPEG-DASH

MPEG-DASH (Dynamic Adaptive Streaming over HTTP) è una tecnologia per lo streaming adattivo via HTTP. In passato le uniche tecnologie esistenti per questo tipo di servizi erano per lo più proprietarie e poco supportate, come Adobe HDS, Apple HLS e Microsoft Smooth Streaming. Così alcuni enti di standardizzazione, quali MPEG e ISO, iniziarono un processo di armonizzazione, che ha prodotto nel 2012 l'approvazione di un nuovo standard internazionale, chiamato appunto MPEG-DASH [18][19][20][21][22].

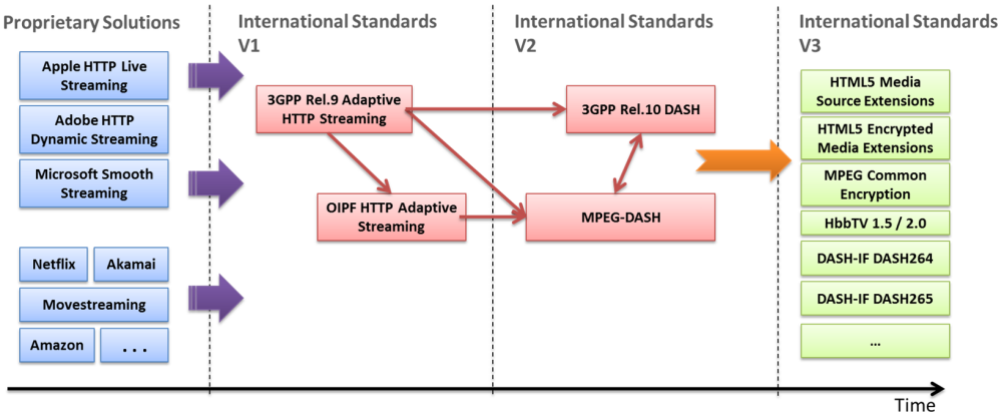


Figura 1.5: Processo di standardizzazione di MPEG-DASH.

1.2.2.1 Obiettivi

Gli obiettivi principali considerati durante la creazione di questo nuovo standard, che si sono poi effettivamente concretizzato, sono:

- Riduzione del ritardo di avvio e del buffering o stallo durante la riproduzione
- Costante adattamento alla capacità di banda del client
- Logica di streaming client-based che abilita la massima scalabilità e flessibilità
- Bypass efficiente di NAT e Firewall sfruttando HTTP
- Agnosticismo nei confronti dei codec utilizzati

Negli ultimi anni, MPEG-DASH è stato integrato in altre tecnologie in corso di standardizzazione, come il Media Resource Extension di HTML5, che ha abilitato il playback di contenuto DASH attraverso il tag HTML video e audio.

Ora, grazie all'adozione di questo tipo di tecnologia da parte di aziende quali Netflix e Google, MPEG-DASH costituisce più del 50% del traffico globale su internet.

1.2.2.2 Architettura

L'idea di base consiste nel dividere i flussi multimediali in segmenti che possono essere codificati con diverso bitrate o risoluzione. Questi segmenti

sono resi disponibili da un server web e possono essere scaricati attraverso richieste HTTP GET.

Mentre il contenuto è riprodotto, il client può automaticamente selezionare dalle alternative per il prossimo segmento da scaricare e riprodurre, sulla base delle condizioni correnti della rete. Di default il client seleziona il contenuto con maggior bitrate che può essere scaricato in tempo per il playback senza causa stalli o eventi di re-buffering.

MPEG-DASH sfrutta come protocollo di trasporto TCP ed è, come già detto, agnostico nei confronti del formato dei flussi, il che gli permette di utilizzare qualsiasi tipo di codifica disponibile e lo rende molto più funzionale rispetto alle controparti proprietarie.

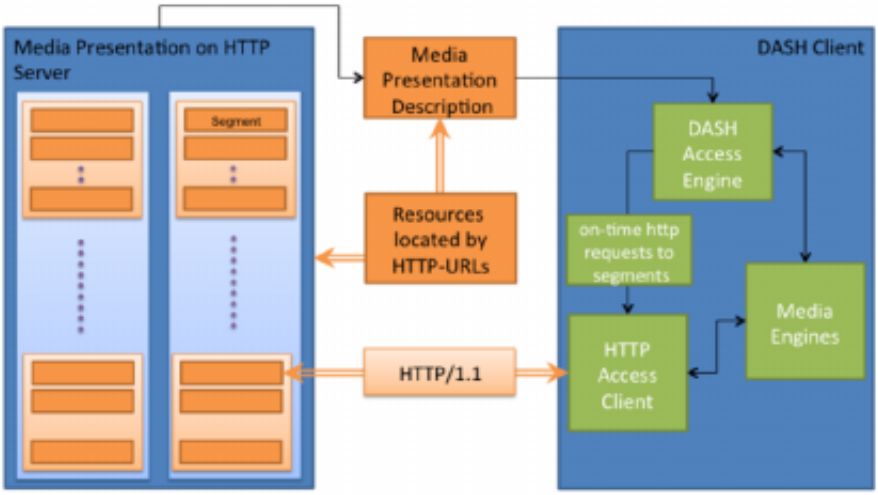


Figura 1.6: Architettura ad alto livello di MPEG-DASH.

1.2.2.3 MPD Manifest

Per poter descrivere la relazione temporale e strutturale dei segmenti, è stato introdotto il Media Presentation Descriptor. MPD è un file xml che rappresenta tutte le differenti "qualità" disponibili del contenuto multimediale, e, per ciascuna di esse, una URL HTTP per ottenerne i segmenti.

Il Media Presentation Descriptor di MPEG-DASH è modello gerarchico che si compone di cinque principali entità:

- *Media Presentation*: Rappresenta l'intero contenuto multimediale, può contenere uno o più periodi
- *Period*: Contiene le informazioni inerenti ai singoli componenti multimediali, intesi come stream audio, video oppure sottotitoli e capitoli

- *AdaptionSets*: Raggruppamento di differenti componenti multimediali, che sono logicamente connessi. Grazie a questo, il client può eliminare un range di componenti che non soddisfano i suoi requisiti
- *SubSet*: meccanismo che abilita restrizioni nelle combinazioni possibili degli AdaptationSets
- *Representation*: contiene le versioni interscambiabili del rispettivo contenuto, come per esempio diverse risoluzioni o bitrate
- *Segments*: definiti da una URL, o da un byte range, rappresentano i frammenti del contenuto multimediale

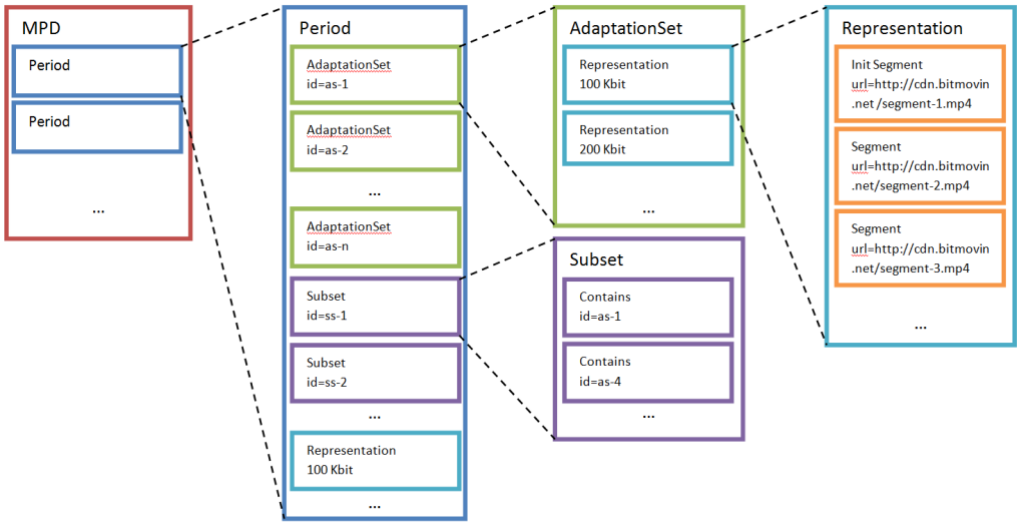


Figura 1.7: Modello del Media Presentation Descriptor

1.2.3 WEBM-DASH

WebM-DASH è una estensione di MPEG-DASH, basato sul WebM container, formato multimediale sponsorizzato da Google e completamente open source, principalmente pensato come alternativa royalty-free per essere usato nei tag HTML5 audio e video.

Risulta essere una valida alternativa a MPEG-DASH principalmente per il fatto che supporta nativamente codec molto più efficienti di H.264 e AAC (codec di default utilizzati da MPEG-DASH), quali VP8, VP9, Vorbis e Opus. Una descrizione più dettagliata di ciascuno di questi codec verrà fatta nella sezione inerente ai Codec audio e video nel terzo capitolo.

Figura 1.8: Logo di WebM

1.3 Dispositivi

Si continui ora la panoramica sul *wearable computing* analizzando i tipi di dispositivi e le loro caratteristiche. Tra i dispositivi *wearable* più noti e diffusi si trovano gli *smartwatch*, che negli ultimi anni sono stati largamente diffusi nel mercato consumer con l'immissione di prodotti di questo ambito da parte di aziende importanti a livello consumer, quali Samsung e Apple.

Sono pensati per essere utilizzati insieme ad uno smartphone, molto spesso collegati tramite bluetooth e sono utilizzati principalmente per le notifiche, come sensori per il monitoraggio dell'attività fisica, cardiaca dell'utilizzatore.

Figura 1.9: Apple Watch, lo smartwatch prodotto da Apple

Oltre agli smartwatch si possono trovare altri tipi di activity tracker, come le smartband, bracciali studiati per monitorare principalmente l'attività fisica, generalmente privi di un display.

Infine troviamo ciò che concerne maggiormente il progetto che verrà illustrato, ovvero gli *smartglasses*. A differenza degli altri sistemi presentano una capacità computazionale notevole e molte più funzionalità, come la possibilità di eseguire applicazioni native, registrare video attraverso una videocamera, connettersi ad internet utilizzando la rete Wi-Fi, piuttosto che quella cellulare. Queste caratteristiche permettono una esecuzione indipendente da altri dispositivi, con cui tuttavia possono comunicare attraverso Bluetooth.

Ad alcuni di essi possono essere impartiti ordini vocali, in questo caso si parla di sistemi *hands-free*, oppure attraverso l'uso di bottoni tattili. Gli smartglasses sono largamente utilizzati anche per l'*augmented reality*, ovvero quella tecnica utilizzata per sovrapporre al mondo reale informazioni digitali. Possono inoltre essere dotati di display in grado di mostrare elementi digitali e nello stesso tempo permettere all'utente di vedere attraverso, nel caso dei Google Glass, oppure possono adottare un approccio più standard, come nel caso dei Vuzix M300, dove lo schermo monoculare ostruisce in parte la visione dell'utente [2][5].

Figura 1.10: Smart glasses Vuzix M300

1.4 Scenari di utilizzo

Di seguito vengono riportati le applicazioni e gli scenari tipici di utilizzo dei dispositivi wearable [1]. Tra i più importanti ci sono:

- *Augmented memory*: attraverso i cosiddetti remembrance agents, è possibile ricordare all'utente di potenziali informazioni rilevanti in base allo stato fisico corrente e al contesto virtuale
- *Finger tracking*: tramite l'utilizzo della videocamera vengono tracciati i movimenti del dito, che possono essere utilizzati come input per il sistema
- *Face recognition*: sempre per mezzo della videocamera, in combinazione ad un software appropriato, possono essere utilizzati per il riconoscimento facciale
- *Navigazione*: Se dotati di GPS potrebbero fornire istruzioni all'utenti su come raggiungere una destinazione desiderata partendo da quella attuale

- *Istruzioni*: grazie all'utilizzo di marker, potrebbe essere possibile visualizzare oggetti in 3D, per fornire una guida dettagliata su come riparare un particolare oggetto

- *Assistenza remota*: potrebbe fornire all'utente una assistenza remota, attraverso l'uso di audio e immagini, per far sì che il personale non addestrato possa comunque portare a compimento semplici attività di riparazione

- *Militare*: possono fornire dati tattici alle truppe, oltre che supportarli nella distinzione tra forze ostili e alleati o consigliare la giusta strategia in situazioni di pericolo

- *Medico*: utilizzati soprattutto per il tracciamento dei parametri vitali di un paziente durante interventi o in caso di soccorso, oppure in ambito *healthcare* per il monitoraggio remoto, utile soprattutto per la prevenzione

- *Sport e Fitness*: possono mostrare le prestazioni degli atleti, gli obiettivi da perseguire o più semplicemente registrare dati quali la distanza percorsa e le calorie consumate

- *Turismo*: potrebbero essere utilizzati per fornire ai visitatori informazioni dettagliate sulle opere, riconoscendo automaticamente a quale opera riferirsi in base, per esempio, alla posizione

Si specifica che questi sono solo alcuni possibili scenari di utilizzo, si prevede infatti che con l'aumentare delle funzionalità, dell'ergonomia e delle prestazioni si vedrà probabilmente l'adozione di questi dispositivi anche in altri settori.

1.5 Pregi e difetti

Si va infine a delineare i pregi e i difetti che presentano questi tipi di dispositivi, che dovranno essere presi in considerazione durante la fase di analisi e progettazione del progetto.

I principali vantaggi per questo tipo di dispositivi sono:

- **Consistenza**: non è necessario spegnerli poiché c'è una costante interazione tra uomo e dispositivo

- **Multitasking**: offrono supporto computazionale anche quando l'utente è attivamente coinvolto nel mondo fisico per altre attività e la sua attenzione è rivolta altrove

- **Mobile**: dispositivi sempre accesi che possono essere utilizzati in qualunque momento o luogo
- **Libertà**: È possibile svolgere altre attività mentre si stanno attivamente utilizzando questo tipo di dispositivi, si parla infatti di paradigmi di interazione *hands-free* e *hands-limited*
- **Context-awareness**: sono consapevoli dell'ambiente in cui operano, multimodali e dotati di diversi sensori per poter percepire l'ambiente circostante
- **Senza distrazioni**: non provocano distrazione, poiché, a differenza di altri dispositivi, non bloccano all'utente la vista del mondo esterno
- **Comunicazione semplificata**: possono essere usati come mezzo di comunicazione senza la necessità di sospendere le attività correnti, attivando magari una conversazione per mezzo della voce o tramite gestures

Uno dei punti fondamentali del wearable computing è appunto lo stretto rapporto tra il dispositivo e il mondo reale. È esemplificativo il grafico seguente in cui viene mostrata la differenza tra i tipi di interazione con sistemi tradizionali e wearable [4].

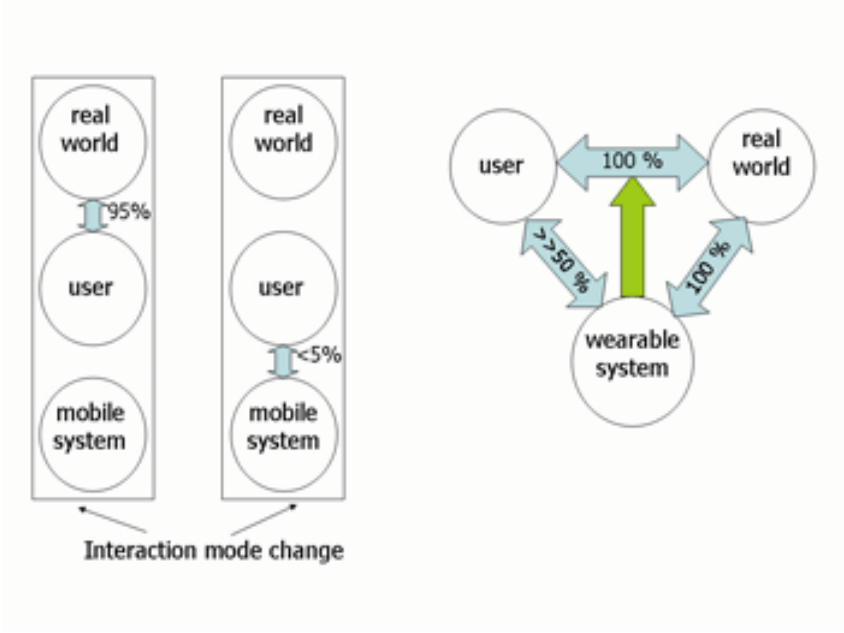


Figura 1.11: Differenze nella modalità di interazione tra i comuni sistemi mobile e wearable.

Dopo averne introdotto tutti i pregi, si andrà ora ad analizzarne meglio gli svantaggi e i loro difetti. Tra i principali è possibile citare:

- **Costo**: nonostante sia in forte calo con la sempre maggiore adozione da parte degli utenti consumer, i prezzi rimangono ancora abbastanza elevati, a causa dell'utilizzo di tecnologie sofisticate
- **Peso**: alcuni modelli risultano essere ancora troppo ingombranti e pesanti, a causa della grande quantità di componenti
- **Sicurezza**: potrebbero costituire un problema di sicurezza, soprattutto in ambito aziendale, poiché gestiscono informazioni sensibili ma non implementano mezzi di sicurezza avanzati
- **Limitate prestazioni**: a causa di limiti in termini di consumo energetico e difficoltà in termini di dissipazione del calore non possono offrire prestazioni molto elevate
- **Batteria limitata**: essendo alimentati a batteria, che, a causa dello spazio limitato, deve essere di dimensioni ridotte, la loro durata potrebbe non essere sufficiente per certi tipo di applicazioni

Nei capitoli successivi verranno analizzati alcuni di questi difetti e si esporrà in che modo sono stati risolti a livello implementativo.

Capitolo 2

Il progetto Trauma Tracker

2.1 Introduzione

In questo capitolo si andrà ad introdurre il sistema *TraumaTracker*, scelto come caso di studio per il framework sviluppato, che verrà introdotto nel capitolo successivo. Ne verrà quindi descritta l'architettura generale, dopodiché si esporranno le motivazioni per hanno portato a richiede l'aggiunta di un servizio di Video streaming e i vantaggi che ne potrebbero risultare.

2.2 Architettura

Il sistema TraumaTracker, sviluppato dall'Università di Bologna grazie alla collaborazioni di docenti, studenti e dottorandi, nasce in risposta alle esigenze dei medici e dei chirurghi dell'ospedale M.Bufalini di Cesena, i quali necessitavano di un sistema informatico di supporto durante il loro lavoro.

Nello specifico il sistema è pensato per assistere il cosiddetto *Trauma Team*, ovvero quel gruppo di medici e infermieri che si occupano di effettuare le prime cure in modo da stabilizzare le condizioni di un paziente, non appena raggiunge il pronto soccorso.

Il sistema quindi è stato pensato principalmente per occuparsi del tracciamento e memorizzazione dei vari eventi, dell'acquisizione di dati per finalità statistiche, in modo da individuare eventuali problemi procedurali [3].

I macro componenti dell'architettura del sistema sono i seguenti:

- Applicazione su smartphone
- Applicazione su smartglasses
- Microservizio per la gestione dei report

- Microservizio per il recupero dei parametri vitali e la gestione dei monitor salvati nel sistema
- Web app per la gestione dei report
- Web app per la gestione dei monitor salvati nel sistema e per visualizzare da remoto i parametri vitali

Per comprendere meglio l'architettura si riporta lo schema dell'intero sistema TraumaTracker con le connessioni e le interazioni di ciascun componente [fig:2.1].

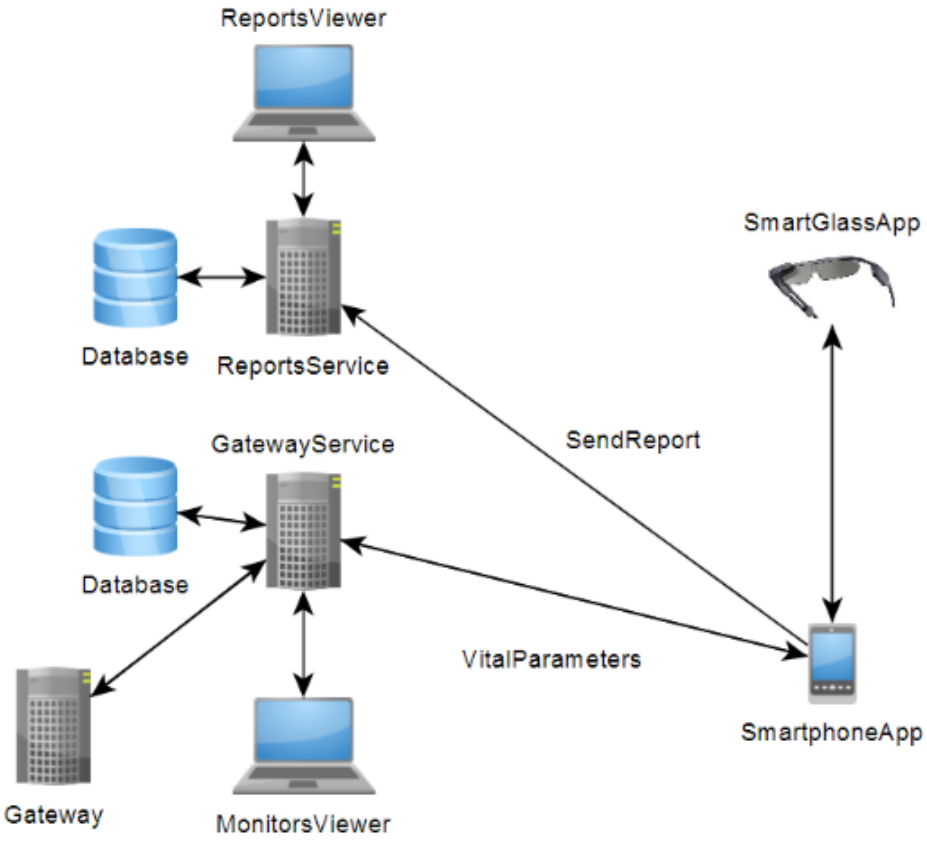


Figura 2.1: Componenti architetturali di Trauma Tracker e le loro interazioni

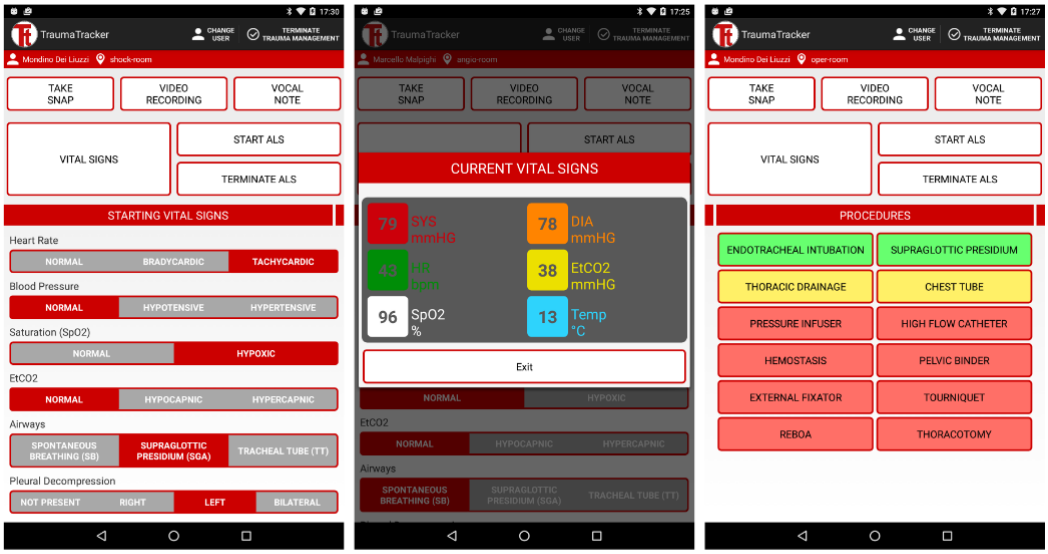
Per ciascuno dei componenti sopracitati si andranno ora ad illustrare più nel dettaglio le caratteristiche.

2.2.1 SmartphoneApp

L'applicazione eseguita sullo smartphone Android, si occupa principalmente di riconoscere ed elaborare i comandi vocali, memorizzare manovre effettuate, farmaci somministrati e parametri vitali, leggere i dati dai beacon per fornire informazioni accurate circa la posizione dell'utente. Offre inoltre l'interfaccia grafica [fig:2.2] per l'inserimento manuale delle operazioni effettuate, come metodo alternativo ai comandi vocali.

L'applicazione presenta principalmente due tipi di stati:

- **Intervento fermo**: stato di partenza del sistema, nel quale è in attesa del comando vocale "inizia intervento"
- **Intervento attivo**: terminato tramite l'utilizzo del comando vocale "fine intervento", si compone di altri due stati, *riconoscimenti attivo*, nel quale il sistema riconosce comandi medici e *riconoscimento fermo* nel quale qualunque tipo di comando vocale viene ignorato.



(a) Starting Vital Signs (b) Current Vital Signs from Monitoring System (c) Procedures

Figura 2.2: Alcuni esempi di schermate inerenti all'applicazione Android

2.2.2 SmartglassesApp

L'applicazione eseguita sugli smartglasses, fornisce un'interfaccia grafica al sistema facilmente accessibile. Permette principalmente la visualizzazione di feedback visivi sui comandi vocali, come per esempio la modalità corrente del sistema, lo stato del riconoscimento vocale oppure quali termini sono stati riconosciuti. Inoltre permette di visualizzare i parametri vitali su richiesta e di scattare foto.

Le informazioni sopracitate vengono ottenute attraverso una connessione Bluetooth con lo smartphone. La presentazione delle informazioni a livello grafico avviene tramite quattro regioni distinte, ciascuna adibita alla visualizzazione di una particolare dato o elemento [fig:2.3].

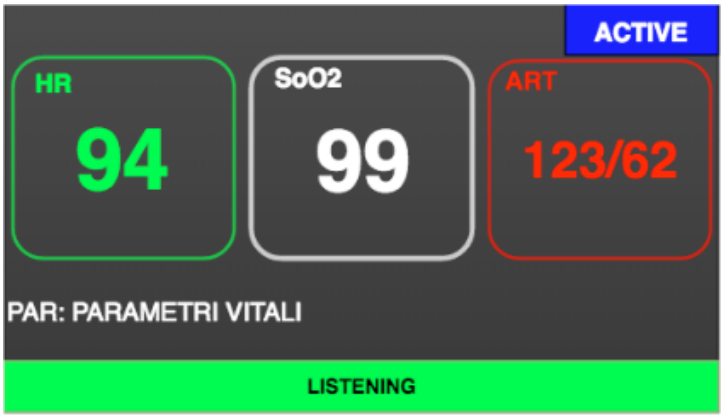


Figura 2.3: Un esempio di visualizzazione dei parametri vitali nell'applicazione per smartglasses

2.2.3 ReportsService e ReportsViewer

Il microservizio ReportsService che espone API per la manipolazione dei report, è possibile infatti aggiungere, modificare e cancellare i report, oltre ovviamente a recuperarli.

Nell'ultimo caso per la visualizzazione del report, il sistema si appoggia alla web app ReportsViewer per permettere all'utente di eseguire le operazioni sopracitate e di esportare i dati dei report in diversi formati.

2.2.4 GatewayService e MonitorsViewer

Un ulteriore microservizio che si occupa di fornire API per il recupero dei parametri vitali del paziente, che sono ottenuti grazie al gateway server di Draeger il quale rende disponibile esternamente i dati dei monitor. Questi dati

sono restituiti in un formato specifico (HL7) che viene ottenuto ed interpretato attraverso un sottosistema chiamato *HL7SubSystem*.

Oltre a questo, il servizio espone metodi per la gestione dei monitor nel sistema, nello specifico aggiunta, recupero, modifica e cancellazione.

Come nel caso del ReportsService, si ha a disposizione una web app, che permette all'utente di manipolare i monitor del sistema e visualizzare i parametri vitali rilevanti.

2.2.5 Infrastruttura GT^2

Il team di sviluppo ha pensato inoltre di implementare un livello che si ponesse sopra tutti i dispositivi esistenti e che si occupasse di interfacciarsi con essi, in modo tale da semplificare l'integrazione di nuove applicazioni all'interno del sistema. I servizi descritti nelle sezioni precedenti implementano appunto questo livello intermedio che permette a qualsiasi applicazione di ottenere dati dai dispositivi ospedalieri in un formato univoco e facilmente gestibile, quale JSON.

In figura [fig:2.4] viene rappresentata l'architettura di TraumaTracker con particolare enfasi sui tipi di protocolli utilizzati per la comunicazione tra i nodi del sistema.

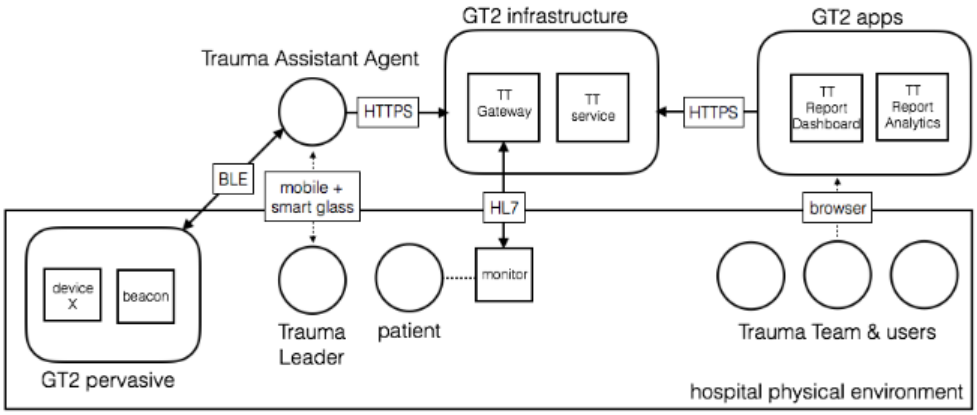


Figura 2.4: Architettura ad alto livello di TraumaTracker

2.3 Vantaggi del video streaming

Di seguito vengono esplicitati i vantaggi principali che il framework di video streaming potrebbe portare al sistema TraumaTracker, alcuni dei quali erano già stati identificati dal team di sviluppo, che ha infatti richiesto questa nuova funzionalità.

Attualmente viene offerto agli utenti la possibilità di conoscere solamente i dati dei monitor e gli eventi registrati dal Trauma Team. Grazie invece a questa nuova funzionalità di video streaming si abiliterebbe un nuovo tipo di visualizzazione che, unito ai dati mostrati dai viewer, permetterebbe ad un utente esterno di avere un quadro migliore sulla situazione del paziente, e, nel caso, fornire un supporto migliore e più accurato.

Riassumendo, i principali vantaggi che si potrebbero ottenere sono i seguenti:

- **Reattività**: si potrebbero comunicare le condizioni del paziente in modo molto più rapido, poiché non ci sarebbe la necessità di inserire nel sistema alcun tipo di dati, ma solamente avviare la registrazione
- **Condivisione**: potrebbe essere uno strumento utile per aggiornare, nel minor tempo possibile, una persona non presente e con maggiore esperienza sulla situazione corrente e ottenere quindi supporto in tempi molto brevi
- **Accuratezza**: si diminuirebbe il rischio di errori di inserimento da parte di un utente, in quanto si potrebbero comunque verificare visivamente i dati riportati dal sistema
- **Livello di dettaglio**: si potrebbe capire con un livello di dettaglio molto più profondo e completo le reali condizioni del paziente combinando i dati forniti dalle apparecchiature e il feedback live offerto dallo streaming
- **Addestramento**: con la possibilità di salvare e riprodurre su richiesta le registrazioni, si potrebbero rilevare molto più facilmente i problemi nelle procedure adottate, e, se necessario, migliorarle. Inoltre potrebbe rivelarsi uno strumento molto utile per addestrare nuovo personale tramite esempi concreti di una corretta esecuzione delle operazioni di soccorso
- **Sicurezza**: potrebbe aprire nuove possibilità per quanto riguarda le attività in ambito legale, ovvero per facilitare le investigazioni riguardo ad eventuali errori gravi commessi o mancanze di chi ha effettuato le operazioni di soccorso

Il framework da implementare cercherà di soddisfare i requisiti principali definiti dal team di sviluppo di TraumaTracker e dai medici coinvolti, rimanendo comunque staccato dal sistema, e sarà progettato in modo da essere applicato in qualunque ambito, anche estraneo al caso di studio di TraumaTracker.

Capitolo 3

Java Video Streaming Framework

In questo capitolo si andrà ad introdurre il framework di Video Streaming sviluppato, analizzando prima i possibili scenari di utilizzo, passando poi alla definizione dei requisiti e alla descrizione delle tecnologie utilizzate. Dopodiché verrà illustrata l'architettura in generale, il tutto integrato dalla descrizione delle varie scelte progettuali ed implementative. Infine verrà illustrato il funzionamento generale del sistema.

Il sistema è stato sviluppato utilizzando il metodo di sviluppo software AGILE, che si è dimostrato particolarmente utile grazie allo sviluppo iterativo, incrementale e la pianificazione adattiva. Infatti, a causa della natura del progetto, fortemente dipendente da fattori esterni, si è fatto largo uso di prototipi incrementali che permettessero di effettuare real-world testing, che hanno permesso di riscontrare e risolvere problemi non predicibili dalla sola analisi preliminare. Inoltre, tra gli obiettivi principali, si è cercato di rispettare il più possibile le regole di buona programmazione e dell'ingegneria.

3.1 Analisi

Il problema proposto, a cui si è cercato di trovare una soluzione, era quello di permettere a dispositivi con capacità limitate, in grado di produrre flussi audio e video real-time, potessero servire un numero indefinito di utenti. In particolare, si è considerata nello specifico la categoria di dispositivi di *Wearable computing*, con particolare enfasi sull'ecosistema *Glassware* o *Eyewear*.

Da una prima analisi sulle capacità dei dispositivi, già descritte nel primo capitolo, e in relazione al problema considerato, si sono messi in risalto i seguenti punti critici:

- Limitata capacità computazionale
- Problematica gestione della batteria
- Insufficiente capacità di rete
- Assenza di scalabilità

I punti riportati sopra introducono notevoli limitazioni che permettono di definire i requisiti che il sistema dovrà essere in grado di soddisfare per ottenere risultati apprezzabili e in linea con le specifiche richieste.

Detto ciò si vede chiaramente la necessità di fare in modo che i dispositivi che si occupano di generare i flussi comunichino al più con un singolo sistema. Una valida soluzione sarebbe quella di creare un sistema intermedio tra la fonte del flusso e l'utente finale. Di questo se ne discuterà più avanti, nella sezione dedicata alla descrizione dell'architettura del sistema.

Proseguendo con l'analisi del problema, si rende necessario definire in che modo un ipotetico utente debba ottenere accesso ai flussi dai dispositivi sorgente. Da specifiche, il sistema dovrebbe rendere disponibile questa parte del sistema sul web, preferibilmente tramite una web app.

Le caratteristiche che questa parte dovrebbe richiedere sono riassumibili nei seguenti punti:

- Facilità di utilizzo
- Estendibilità
- Possibilità di accesso da qualunque dispositivo

Per poter garantire le caratteristiche sopracitate, si dovrebbe considerare, ancora una volta, una centralizzazione di tutta la logica di gestione di questa parte front end del sistema.

Ciò, legato al discorso precedente, enfatizza la necessità di gestire la maggior parte della logica del sistema sotto una singola entità. Vedremo quindi nelle sezioni successive in che modo questo si traduce nella progettazione del sistema.

3.1.1 Requisiti

Dopo un'analisi preliminare sulle caratteristiche e sui problemi, si è passato alla definizione dei requisiti del sistema. Contestualizzandolo ad uno scenario d'uso tipico sono stati riscontrati i seguenti requisiti necessari per fare il modo che il sistema rispondesse adeguatamente alle necessità:

- Si dovrebbe considerare accuratamente il consumo per le sorgenti dei flussi, delegando le operazioni computazionalmente onerose ad un altro dispositivo in grado di supportare tali carichi e senza la necessità di tenere in considerazione possibili problemi legati all'alimentazione.
- Necessità di una scalabilità elevata per poter inviare i flussi ad un numero elevato di utenti
- Una buona reattività dovrebbe essere garantita per fare in modo che le comunicazioni tra i sottosistemi siano veloci e che i contenuti possano essere serviti con una latenza minima

Oltre a quanto detto sopra, altri punti critici sono:

- L'adattamento del sistema a tutti i possibili casi di utilizzo, si dovrebbe perciò considerare il rapporto qualità/compressione dei flussi audio e video, favorendo un adattamento dinamico a seconda delle condizioni
- Una buona configurabilità dovrebbe essere garantita per poter modificare, senza ulteriore sforzo, il comportamento del sistema
- Il sistema dovrebbe essere, per quanto possibile, estendibile e aperto alla possibilità di introdurre nuove componenti per estendere o migliorare le funzionalità

In relazione ai requisiti sopracitati e all'analisi eseguita nella sezione precedente, si procede nel definire gli attori che interagiranno con il sistema e le loro possibili azioni. Pensando ad uno scenario tipico, si può facilmente riconoscere che il sistema avrà principalmente due tipi di utilizzatori:

- Produttore: l'utente munito di dispositivo mobile, tendenzialmente smartglasses, tablet o smartphone, il quale potrà creare nuove sessioni di streaming, con la possibilità di definirne anche il tipo, per favorire la qualità del contenuto o per evitare limitazioni del proprio dispositivo o della rete
- Consumatore: l'utente che potrà visualizzare la lista di sessioni disponibili, rimuoverle, se necessario, e avviare il playback di una sessione da lui scelta

Si intuisce poi che il server sarà il sottosistema principale dell'intero sistema che dovrà da un lato estendere le capacità limitate del produrre e dall'altro fornire le stream ai consumatori in un formato facilmente riproducibile. Si presuppone già che il sistema potrà avere contemporaneamente più produttori e più consumatori attivi nello stesso istante.

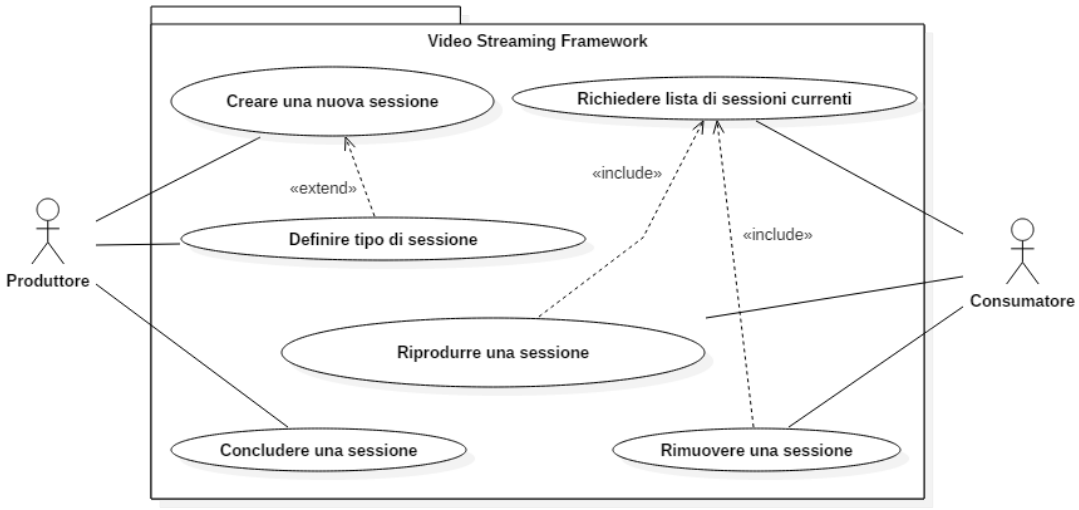


Figura 3.1: Diagramma dei casi d'uso

3.2 Tecnologie

Dopo la prima parte di analisi, di seguito sono riportate tutte le tecnologie che sono state scelte per l'implementazione del framework e che, combinate, soddisfano i requisiti riportarti nella sezione precedente.

3.2.1 Vert.x

Vert.x è un toolkit poliglotta orientato agli eventi che viene eseguito dalla Java Virtual Machine (JVM). Si basa su un modello ad event loop ed è adatto per implementare servizi di backend. Processa tutte le operazioni di IO usando Netty, un framework low level, così da offrire prestazioni elevate [6].

Di seguito vengono elencate le sue caratteristiche principali:

- *Semplice modello di concorrenza*: essendo basato su un modello ad eventi, tutto il codice viene eseguito in modalità single threaded, così che non sia necessario per il programmatore occuparsi di tutti i problemi legati al multi threading.

- *Event bus distribuito*: gli eventi possono propagarsi anche tra server diversi, funzionando sia con meccanismo *punto-punto* sia con quello *publish-subscribe*. Sfruttando questa funzionalità è possibile far interagire tecnologie diverse tra loro.

- *Modello di programmazione asincrono*: consente di implementare applicazioni scalabili e non bloccanti

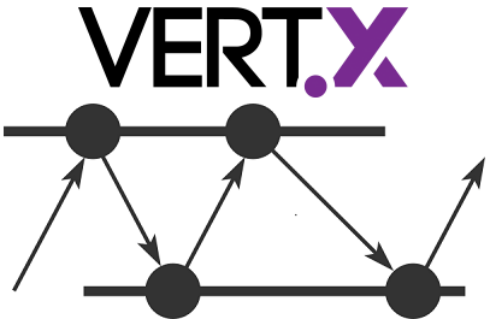


Figura 3.2: Logo di Vert.x

3.2.1.1 Event loop e Reactor pattern

Vert.x, per evitare l'overhead causato dal context switch dei thread, utilizza un modello event loop, in cui un singolo thread si occupa di tutta la gestione degli eventi. Il pattern di riferimento è il cosiddetto *Reactor Pattern*, dove l'unico thread in esecuzione invoca gli event handler associati agli eventi.

Le fasi del loop sono:

1. L'applicazione genera una richiesta non bloccante all'Event Demultiplexer attraverso una nuova operazione di IO

2. Al termine dell'operazione viene inserito un nuovo evento nell'Event Queue

3. L'Event Loop processa gli eventi nella coda richiamando il corrispondente handler, che non appena termina restituisce il controllo all'Event Loop.

4. Quando nell'Event Queue non rimangono più eventi da processare, il loop si ripete bloccandosi nell'Event Demultiplexer

Nello schema successivo vengono riportati gli elementi che compongono il *Reactor pattern*, le varie fasi e il loro ordine di esecuzione [fig:3.3].

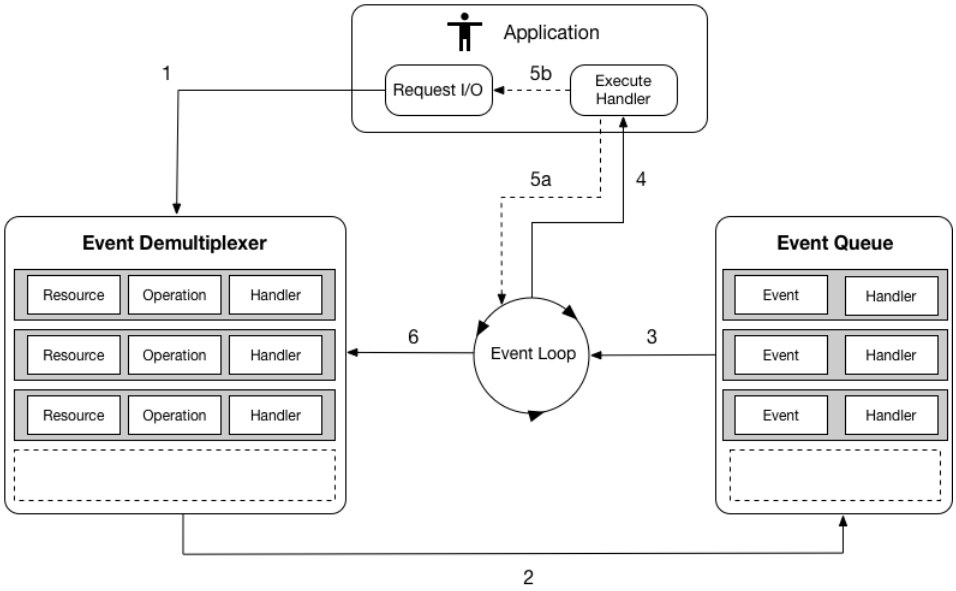


Figura 3.3: Schema del Reactor Pattern

3.2.1.2 Architettura

L'unità di esecuzione principale di Vert.x è il Verticle dei quali possono esserne eseguiti molteplici in una singola istanza di Vert.x. I Verticle sono eseguiti nei thread dell'Event Loop. Diverse istanze di Vert.x possono essere eseguite in diversi host o in un unico host remoto. In questo caso i Verticle o i moduli comunicano utilizzando l'Event Bus. Per riassumere quindi, una applicazione Vert.x consiste nella combinazione di Verticle, o moduli, e nella comunicazione tra essi, resa possibile grazie all'Event Bus [7][8].

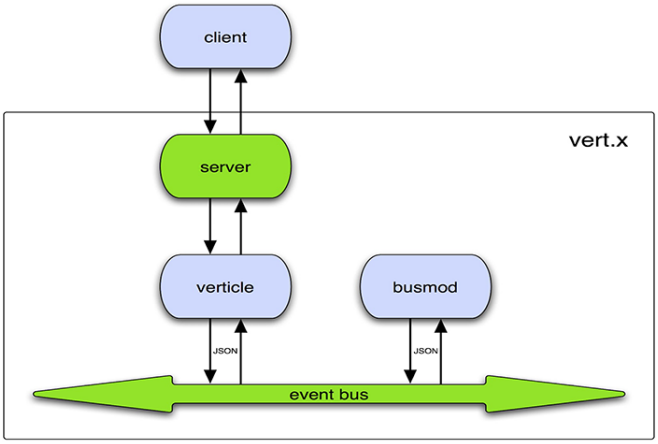


Figura 3.4: Grafico architetturale di vert.x

3.2.1.3 Componenti

Di seguito sono riportati i componenti principali di Vert.x:

- *Verticle*: Unità di esecuzione principale, comunica tramite Event Bus
- *Verticle Instance*: un Verticle è eseguito dentro una istanza di Vert.x, ciascuna di queste, a sua volta, è eseguita nella sua istanza dentro la JVM
- *HazelCast*: è una struttura dati che risiede in RAM, e che può essere distribuita tra più server. Utilizzata principalmente per condividere dati tra Verticle o istanze di Vert.x
- *Event Bus*: principale mezzo di comunicazione tra istanze di Verticle
- *HTTP Server*: utilizzato per registrare gli handler inerenti a richieste HTTP. Supporta anche WebSocket.
- *Net Server*: utilizzato per gestire gli eventi dell'Event Bus e per la costruzione di server TCP
- *Event Loops*: gruppo di thread che eseguono gli Event Loop, uno per ogni processore
- *Background Thread Pool*: gruppo di thread a disposizione nel caso in cui un handler preveda un'operazione bloccante o di lunga durata
- *Acceptor Thread Pool*: gruppo di thread per eseguire l'accept delle socket, uno per ogni porta
- *NioWorker*: handler non bloccanti, eseguiti senza l'utilizzo di thread dal Background Thread Pool
- *Blocking Job*: handler bloccanti, eseguiti utilizzando thread dal Background Thread Pool
- *Worker Mode*: istanze di Verticle che eseguono esclusivamente task bloccanti
- *Shared Data*: componente utilizzato per la condivisione di dati tra Verticle o istanze di Vert.x
- *Shared Server*: server condiviso tra tutti i Verticle

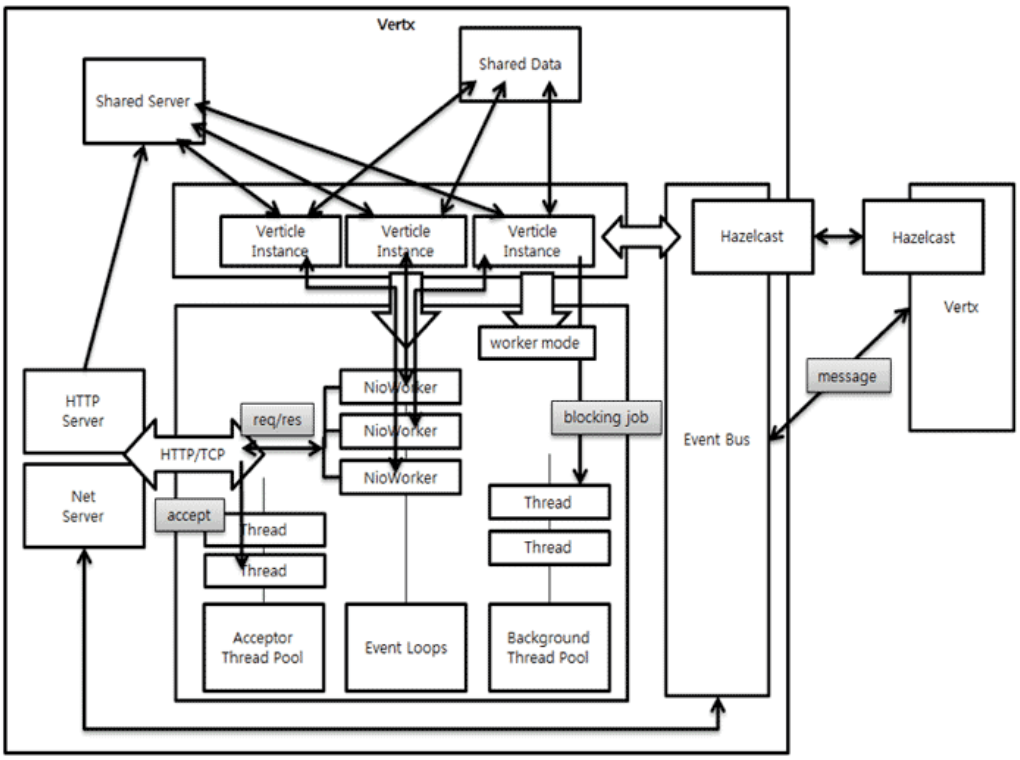


Figura 3.5: Diagramma dei componenti di vert.x

3.2.2 REST

REST, acronimo di REpresentation State Transfer, sono linee guida per la realizzazione di un tipo di architettura software per servizi di backend. L'utilizzo più diffuso dell'architettura REST è su HTTP, dove, con la definizione di una struttura degli url e l'utilizzo di metodi specifici del protocollo, per la selezione dell'azione da eseguire, è possibile accedere alle risorse di un Web Service.

I principi di progettazione sono:

- Risorse autodescrittive
- Collegamenti tra risorse
- Identificazione delle risorse
- Utilizzo esplicito dei metodo HTTP
- Comunicazione senza stato

3.2.2.1 REST e Vert.x-Web

Vert.x-Web utilizza ed espone le API da Vert.x core, è stato pensato per essere potente e completamente incorporabile. È possibile utilizzarlo per la creazione di applicazioni web server-side, servizi RESTful o server push.

I concetti fondamentali sono:

- *Router*: È l'entità base, un oggetto che mantiene zero o più routes. Si occupa di smistare ogni richiesta HTTP alla prima route che corrisponde ad essa.

- *Routes*: Sono entità che permettono di definire la suddivisione dei vari handler dichiarando dei path specifici che verranno utilizzati dal Router per processare le richieste

- *Handlers*: entità che contengono le azioni da compiere per gestire le richieste

Di seguito è riportato un esempio di codice per la creazione di una API:

```
Router router = Router.router(vertx);

router.route().handler(BodyHandler.create());

router.get("/products/:productID").handler(this::handleGetProduct);
router.put("/products/:productID").handler(this::handleAddProduct);
router.get("/products").handler(this::handleListProducts);

vertx.createHttpServer().requestHandler(router::accept).listen(8080);
```

In questo caso viene creato un router, al quale poi viene aggiunto un Body Handler, per gestire l'eventuale corpo delle richieste. Dopodiché vengono definite delle routes, dichiarando per ciascuna il metodo HTTP corrispondente, la path relativa e il codice che dovrà essere eseguito, tramite l'handler. In questo caso all'handler è passato un riferimento ad un metodo implementato nella stessa classe. Infine, viene creato un server HTTP, in ascolto sulla porta 8080, il quale è configurato per gestire tutte le richieste HTTP in ingresso tramite il router creato in precedenza.

3.2.3 JSON

JavaScript Object Notation (JSON) è un formato utilizzato per lo scambio di dati. Essendo facilmente interpretabile sia da umani che dalle macchine e

semplice da gestire, è diventato uno standard de facto per quanto riguarda lo scambio di dati sul Web [13].

Si compone di due strutture principali:

- *Object*: insieme di coppie chiave/valore

```
{
   "name" : "John Doe",
   "age" : 30
}
```

- *Array*: elenco ordinato di valori

```
[
   "Red",
   "Black",
   "Green"
]
```

L'unico tipo accettato per i nomi (o chiavi) è stringa, mentre per i valori sono: number, object, array, boolean, null e string. Essendo la sintassi di dichiarazione delle strutture molto semplice il parsing JSON risulta molto veloce.

Si specifica che per questo progetto è stata utilizzata la libreria Java "Jackson" inclusa nel framework Vert.x.

3.2.4 FFmpeg

FFMpeg, acronimo di Fast Forward Motion Pictures Expert Group è il principale framework ad alte prestazioni, cross platform e open source, per la manipolazione di contenuti multimediali. Supporta la codifica e decodifica della maggior parte dei codec conosciuti [14].

Le librerie più note che contiene sono libavcodec, una collezione di codec audio/video (es. H264, VP8, MP3, AAC) , libavformat, una collezione di contenitori audio/video (es. MP4, WEBM). Il framework FFMpeg offre uno strumento a riga di comando, chiamato ffmpeg utilizzato per la conversione di flussi audio video, l'elaborazione di flussi real-time e l'applicazione di filtri a flussi audio e/o video [16].

ffmpeg è stato scelto per lo sviluppo della parte backend di questo progetto soprattutto per le sue performance, il supporto a quasi tutte le tecnologie coinvolte e la quasi illimitata configurabilità.

Figura 3.6: Il logo di FFmpeg.

3.2.5 Codec audio e video

Un codec è un programma o dispositivo che si occupa di codificare digitalmente segnali di tipo audio o video, così che possa essere salvato su disco.

I codec effettuano una compressione dei dati per poter ridurre lo spazio di memorizzazione occupato, aumentandone così la portabilità, trasmissibilità e, in generale, facilitandone la loro gestione.

I codec si dividono in base al tipo di modalità utilizzata per la compressione:

- *Lossy*: nel caso in cui dalla compressione derivi una perdita di informazione
- *Lossless*: quando la compressione viene eseguita senza perdita di informazione

La compressione può consiste nella riduzione della precisione dei colori per i singoli pixel, al salvataggio delle sole differenze di un frame da un altro o eliminazione di ridondanze, nel caso di frame video, o di frequenze da riprodurre, nel caso di flussi audio [17].

Per quanto riguarda questo progetto, e lo streaming in generale, si prediligono le modalità di compressione lossy, poiché minimizzano il numero di dati rendendone più agevole il trasporto.

Tra i codec video più usati ci sono:

- *H.264*: consente una buona qualità video a basso bitrate e supporta compressioni lossless e lossy. Essendo supportata la codifica/decodifica hardware da molti dispositivi è ancora largamente utilizzato
- *H.265*: successore di H.264, offre una diminuzione di oltre il 40% del bitrate a parità di qualità permettendo così la fruizione di contenuti 4K
- *VP8*: alternativa royalty-free, open source di H.264, di cui ne condivide approssimativamente le caratteristiche

- *VP9*: in competizione con H.265, offre una diminuzione del 50% del bitrate rispetto a VP8, utilizzato correntemente da YouTube

Mentre per quanto riguarda la categoria audio:

- *AAC*: adotta un approccio modulare, a seconda della complessità del flusso da codificare, delle prestazioni e il risultato che si vuole ottenere. Ne esistono 3 profili: AAC LC, il più semplice, più utilizzato e supportato, soprattutto in situazioni di mancanza di risorse; AAC, utilizza tutte le funzionalità del codificare; AAC SSR, utilizzabile solo per la codifica di tracce stereo

- *Vorbis*: algoritmo libero per la compressione lossy di audio digitale, permette una maggiore compressione a parità di qualità rispetto a MP3 grazie a tecniche di psicoacustica

- *Opus*: uno tra i migliori codec disponibili per la compressione audio, utilizzato da moltissime organizzazioni quali Microsoft, Mozilla e Google. Sviluppato specificatamente per comunicazioni in tempo reale, è in grado di offrire una qualità accettabile, in condizioni di mancanza di banda, ma anche molto elevata, in caso di una rete stabile, pur utilizzando un bitrate relativamente basso

In generale, all'aumentare dell'efficienza aumenta anche la quantità di tempo impiegata per la codifica e anche la decodifica. Proprio per questo motivo, per quanto riguarda questo progetto, è stato deciso di aggiungere supporto a tutti i codec sopra indicati, offrendo così una soluzione flessibile, che permettesse di favorire compatibilità da un lato o la qualità e l'efficienza dall'altro.

3.3 Progettazione

Riflettendo su quanto emerso dall'analisi del problema, e dopo aver confrontato le varie tecnologie disponibili per trovare la soluzione migliore al problema dato, si possono facilmente riconoscere tre principali sottosistemi necessari per il funzionamento del sistema: Client producer, il Server di broadcasting e il Client consumer.

Il *client producer* è l'entità che si occupa di produrre i flussi audio e video. È da considerare con risorse limitate, in movimento e con capacità di rete limitata. In questo caso quindi sarebbe opportuno limitare al massimo le operazioni. Alla luce di questo, il primo sottosistema si dovrebbe quindi limitare alla sola compressione e invio del flusso in ingresso dalla videocamera e dal microfono.

Oltre a ciò, nella fase di analisi si era rilevato un altro problema abbastanza limitante per quanto riguarda questo sottosistema, ovvero la durata della batteria. Nonostante sia possibile ovviare a questo problema per mezzo delle cosiddette Power Bank, batterie esterne portatili, che offrirebbero un incremento notevole della durata della batteria (ma un maggiore impedimento per l'utente finale), è comunque necessario studiare un metodo per massimizzare l'efficienza.

Presumibilmente quindi, utilizzando un codec per la compressione che abbia supporto hardware nel SoC (System-on-a-Chip) del dispositivo, si dovrebbe massimizzarne l'efficienza e minimizzarne i consumi.

Un altro aspetto importante inerente a questa parte è la strategia che dovrà essere adottata per la comunicazione tra server e client producer. In questo caso per offrire maggiore adattabilità il client producer potrebbe definire il comportamento del server, comunicando magari parametri specifici interpretabili dal server.

Una possibile strategia di handshaking (inizializzazione della connessione) tra client e server potrebbe quindi essere la seguente:

1. Il client invia una richiesta al server specificando i parametri di configurazione della sessione

2. Il server prepara la sessione, si mette in ascolto, in attesa della stream, su una specifica path che invia al client

3. Il client avvia la registrazione e lo streaming del flusso RTSP al server che inizia la conversione

4. Il client può in qualunque momento decidere di terminare la registrazione, appena il server rileva che la stream RTSP è stata chiusa, termina la conversione

Di seguito è riportato il diagramma del sequenze con una descrizione dettagliata di tutte le comunicazioni tra i due sistemi.

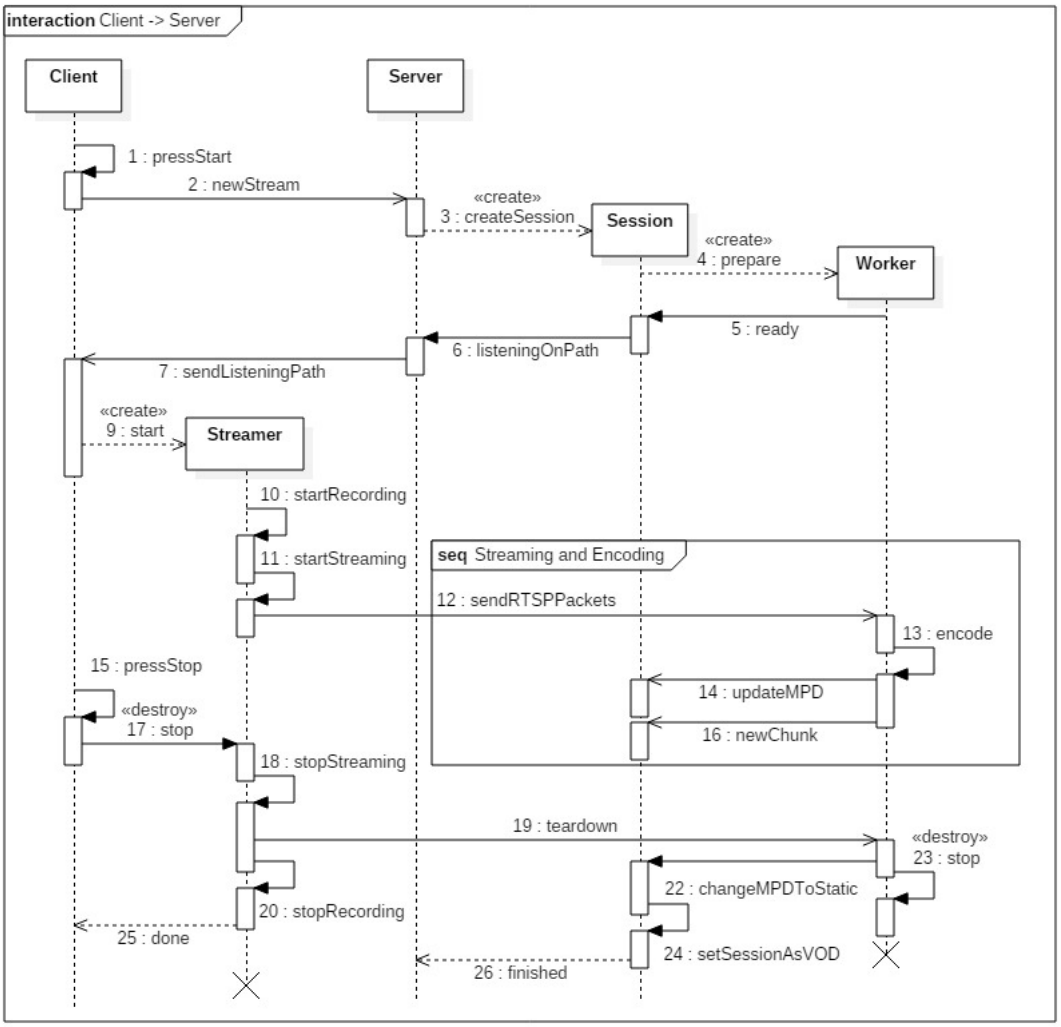


Figura 3.7: Diagramma delle sequenze tra client producer e server

Per quanto riguarda il client consumer, la strategia di comunicazione con il server risulta essere molto più banale e non richiede particolari attenzioni. Di seguito vengono citate le possibili iterazioni:

1. Il client richiede la lista di stream attive, il server risponde con un json array contenente tutte le stream attive al momento, rimuovendo eventualmente quelle che hanno generato errori

2. Il client invia una richiesta di cancellazione di una stream con un id preciso, il server, controlla la validità dell'ID, termina eventuali worker attivi sulla specifica stream e segnala al client il fallimento o il successo dell'operazione

3. Il client decide di iniziare il playback di una stream, per prima cosa richiede al server le informazioni sulla stream, riceve quindi un json object dal quale estrae la url del manifest, che utilizza per inizializzare il player

Nell'elenco precedente è esclusa la comunicazione che viene effettuata tra il player lato browser e il server, poiché gestita internamente e definita dallo standard. Si specifica comunque che normalmente, durante la riproduzione, il player scarica progressivamente i chunk (segmenti) dal server attraverso semplici richieste HTTP, selezionandoli in base al loro timecode, che viene inteso come la sequenza di codici numerici che rappresentano lo specifico momento del contenuto riprodotto.

Nella figura [3.14] viene mostrato il diagramma delle sequenze per quanto riguarda l'interazione tra queste due entità.

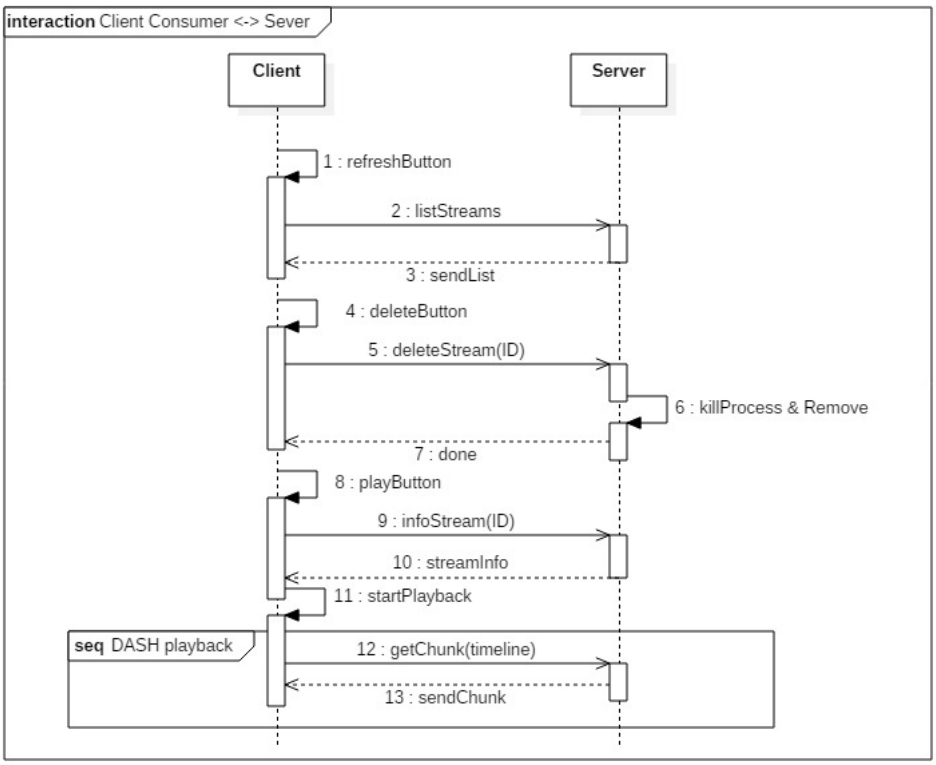


Figura 3.8: Diagramma delle sequenze tra client producer e server

Infine, per quanto riguarda il sottosistema di backend, si dovrà tenere in considerazione il fatto che verrà implementato utilizzando Vert.x. In questo modo tutte le richieste dei client saranno processate in modo asincrono. Si dovrà quindi gestire adeguatamente la modifica della lista globale di tutte le stream attive, per evitare problemi di concorrenza.

Poiché il server potrà dover gestire più richieste parallele sia in entrata che in uscita, la gestione dei processi ffmpeg, o in generale di qualunque altra operazione bloccante, dovrà essere eseguita in background per evitare di bloccare l'intero sistema. Oltretutto dovrà essere preso in considerazione il fatto che sarà necessario ottenere informazioni importanti sulle stream dall'output ffmpeg, come la durata corrente, o intercettare messaggi di errore o warning.

3.4 Architettura

Avendo completato la fase di progettazione del sistema è arrivato il momento di introdurne l'architettura.

Come graficato in figura [3.15] il sistema presenta tre tipi di entità:

- **Client Producer**: può essere un qualsiasi dispositivo wearable con capacità di registrazione, al suo interno presenta una classe "Streamer", che incapsula tutta la logica di comunicazione, registrazione, codifica e streaming del flusso RTSP.

- **Server**: si occupa della gestione di tutti i flussi in ingresso, la loro conversione e il mantenimento di tutte le informazioni rilevanti del sistema in un determinato istante (come la lista di stream attive). Espone i metodi di accesso ai dati con una logica di tipo REST e un server http, che permette, attraverso una path specifica, di accedere alla *web app* utilizzata dal client consumer. Grazie all'utilizzo di MPEG-DASH non è necessaria alcuna gestione del playback, poiché il player può ottenere i dati facendo richieste HTTP GET per ottenere i segmenti inerenti al timecode da riprodurre. La conversione dei flussi è gestiti tramite i Worker, classi che consentono di eseguire il processo ffmpeg in background, con la possibilità di gestirne eventi come la terminazione, l'aggiornamento delle statistiche di conversione o la comunicazione di errori.

- **Client Consumer**: grazie all'utilizzo di Vert.x e all'implementazione di una REST API lato server, il client non fa altro che eseguire richieste http su path remote specifiche per ottenere le informazioni o effettuare delle azioni tramite l'interfaccia web fornita dalla web app.

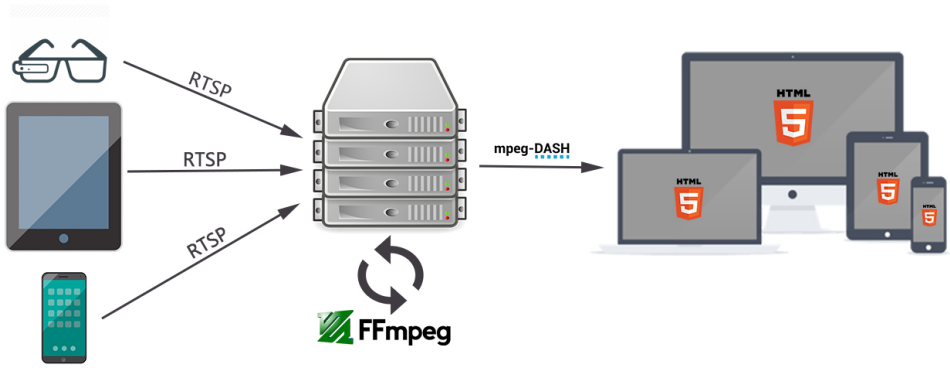


Figura 3.9: Architettura del sistema

Raggiunto questo punto è necessario analizzare più nello specifico l'architettura del Web Service per comprenderne meglio il funzionamento. Principalmente il servizio si basa tutto sulla gestione asincrona degli eventi scatenati da Vert.x dopo aver ricevuto una richiesta HTTP.

Essendo stato implementato tramite REST, la classe principale crea un oggetto router, definisce tutte le route necessarie e lo utilizza come metodo di accept di un nuovo server http. Alcune di queste route sono definite statiche e puntano ad una cartella specifica del file system, per fare in modo che il client possa scaricare tutti i file necessari per il funzionamento della web app e soprattutto per permettere il download del manifest DASH e dei relativi chunk. Il servizio descrive le caratteristiche delle sessioni utilizzando una classe interna specifica denominata "Stream". Grazie ad una collezione di questi oggetti il "MainVerticle" tiene traccia di tutte le sessioni attive e può accedere alle informazioni conoscendo l'identificativo univoco della stream.

La Stream offre dei metodi per inizializzare ed avviare la classe Encoder, che incapsula tutta la logica di gestione di ffmpeg, che, a seconda del tipo di Encoding (metodo di codifica) richiesto dall'utente potrebbe aver necessità di utilizzare la classe Analyzer, che provvede ad avviare ffprobe, un tool supplementare di FFmpeg, per ottenere le informazioni sui flussi audio e video.

La generazione dei comandi è gestita da una classe di supporto chiamata CommandsGenerator, la quale, data una Stream, un oggetto StreamInfo e il tipo di Encoding, genera comandi sia per avviare la codifica delle tracce in ingresso sia per la generazione del manifest. L'oggetto StreamInfo viene creato attraverso la classe Analyzer, e rappresenta tutte le tracce audio e video nel flusso RTSP e la loro posizione relativa. È necessario questo tipo di approccio perché ffmpeg presenta un sistema interno per la mappatura di tutte le tracce dell'input, grazie al quale è possibile definire parametri specifici per ciascuna di esse.

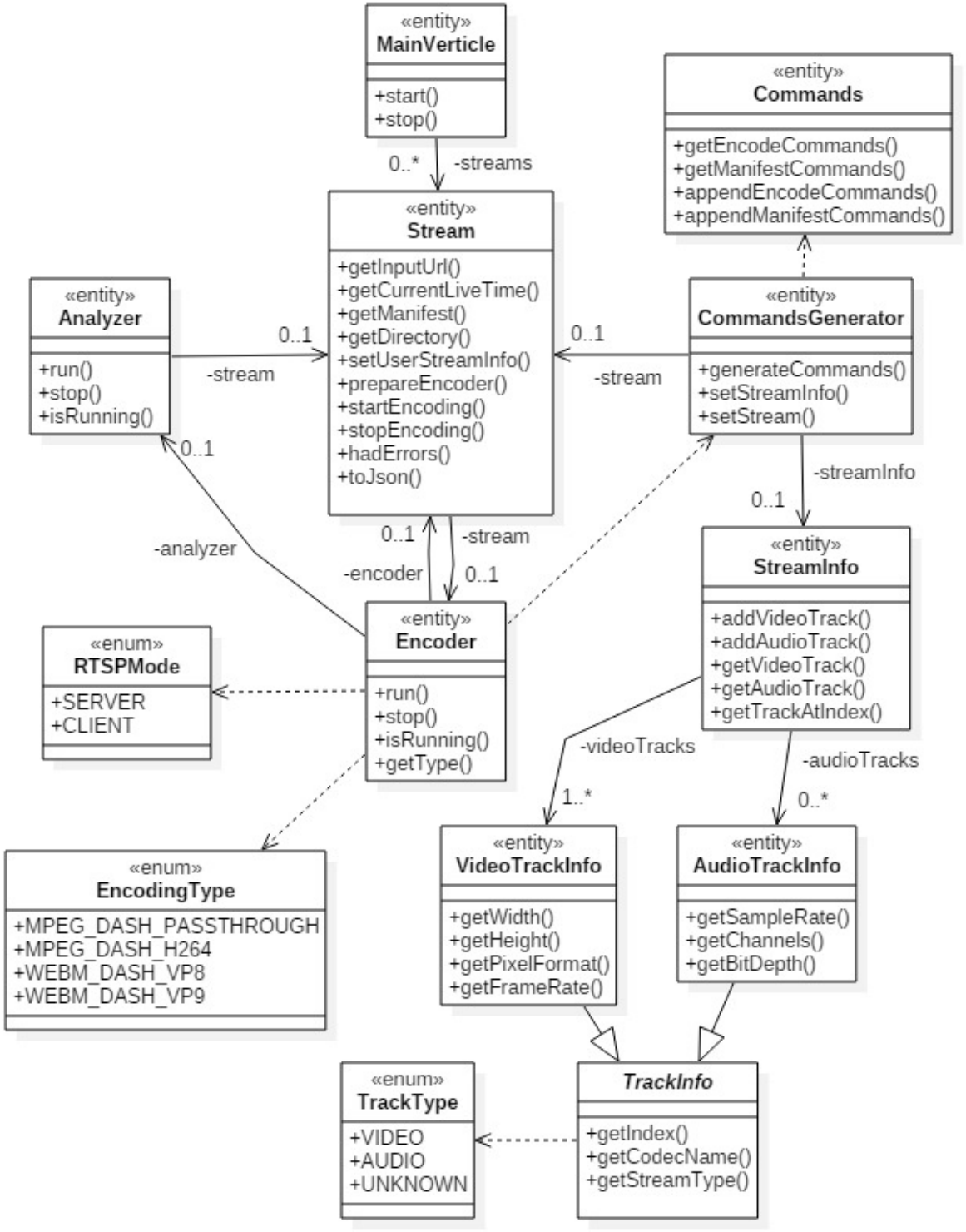


Figura 3.10: Diagramma delle classi inerente alla gestione delle stream

I tipi di encoding disponibili sono definiti da una enumerazione chiamata EncodingType e contiene quattro valori:

- *MPEG–DASH (passthrough)*: Il profilo più veloce e meno costoso in termini computazionali, in quanto non effettua la ricodifica dei flussi audio e video, ma si occupa semplicemente di frammentarle. Tuttavia, nel caso in cui la codifica dei flussi lato client producer non sia configurata correttamente si potrebbero avere problemi di latenza
- *MPEG–DASH (H264/AAC)*: profilo molto veloce e poco esoso di risorse, effettua la riconversione del flusso video e audio e offre la massima compatibilità
- *WEBM–DASH (VP8/Vorbis)*: simile al precedente metodo, sfrutta però il formato webm e codifica i flussi video in VP8 e quelli audio in Vorbis, offre una buona compatibilità
- *WEBM–DASH (VP9/Opus)*: formato simile al precedente, sfrutta però codec più recenti ed efficienti per la codifica dei flussi, il profilo col maggior costo computazionale, offre una limitata compatibilità

In alcuni casi, oltre al tipo di Encoding, l'Encoder deve tener conto anche della modalità RTSP richiesta dal client. Le possibili modalità e la gestione richiesta per ciascuna sono elencate in seguito:

- *Server*: ffmpeg deve essere configurato in modo che rimanga in ascolto su una porta e route specifica, in attesa di una richiesta RTSP di ANNOUNCE da parte del client. Non avendo la possibilità di analizzare preventivamente la stream, se è selezionato un profilo di encoding che richiede una riconversione dei flussi, il client si deve occupare di segnalare al server le informazioni generali su ciascuna stream. Utilizzando questa modalità è anche necessario che il client comunichi esplicitamente la chiusura della stream, per far sì che il servizio possa terminare il processo ffmpeg, che altrimenti resterebbe attivo ed in ascolto per altre eventuali richieste
- *Client*: ffmpeg viene utilizzato come client RTSP, riceve la url da contattare per prelevare i flussi. In questo caso quindi non è necessario che rimanga in ascolto ed è possibile acquisire le informazioni sulla stream utilizzando prima di tutto la classe Analyzer, che, utilizzando ffprobe, ottiene l'analisi completa di ciascuna traccia tramite array json. Grazie all'ObjectMapper della libreria Jackson di Vert.x, dal codice json vengono create nuove classi VideoTrackInfo o AudioTrackInfo, che sono aggiunte all'oggetto StreamInfo e utilizzate successivamente dal CommandsGenerator

3.5 Implementazione

Entrando più nello specifico si esponga come è stata implementata, nelle classi Encoder e Analyzer, la gestione dei processi ffmpeg e ffprobe. Si è proceduto inizialmente creando un classe astratta Worker che implementasse l'interfaccia Runnable di Java.

L'eseguibile ffmpeg riporta tutte le informazioni riguardo alla conversione, gli errori o i warning nello *standard error*. Sulla base di questo, il metodo run di ciascun worker è stato implementato per leggere dai file descriptor del processo. All'interno del metodo è presente un loop che viene eseguito fintanto che il metodo readLine del BufferedReader non ritorna un valore null, che presumibilmente coincide con la chiusura del processo.

Dentro al ciclo si filtrano le linee cercando delle parole ricorrenti all'inizio delle sotto stringhe e, quando viene trovata una corrispondenza, si eseguono operazioni specifiche per estrarne le informazioni.

L'esecuzione del loop sopra citato non causa problemi di busy waiting perché il metodo readLine utilizzato, che viene eseguito all'interno di un thread in background, è bloccante e ritorna solo quando viene letto un carattere new line.

Di seguito vengono riportati gli eventi previsti dai worker implementati:

- *OnProgress*: permette al sistema di aggiornare costantemente la durata corrente di una stream in fase di encode. L'evento viene scatenato nel momento in cui viene trovata una corrispondenza tra l'inizio della linea e la stringa "frame=". Dopo l'acquisizione della stringa tramite espressioni regolari della rappresentazione del timecode corrente, ne viene eseguito il parsing e creato un oggetto java di tipo Duration. Questo evento torna un oggetto di tipo ProgressEventArgs che contiene due campi opzionali, uno di tipo Duration che è utilizzato per comunicare la durata corrente della stream e un secondo campo di tipo stringa per la segnalazione di eventuali messaggi di warning.

  Un esempio di stringa di progresso riportata da ffmpeg è il seguente:

```
frame= 169 fps=105 q=29.0 size= 99543kB time=00:01:25.70
   bitrate=3449.3kbits/s speed=3.56x
```

- *OnComplete*: questo evento è scatenato prima che il metodo run termini e ritorna un oggetto di tipo CompletedEventArgs, il quale contiene un campo opzionale intero per la comunicazione del codice di uscita del processo (exit code). Inoltre è presente un secondo campo di tipo Object per permettere al worker di comunicare qualsiasi altro tipo di dato che potrebbe variare in base all'implementazione.

Per assicurarsi che il processo sia terminato e per ottenere il suo codice di uscita, è necessario richiamare il metodo bloccante waitFor dell'oggetto Process una volta che il metodo readLine ha interrotto il loop restituendo null dopo aver letto EOF (End-of-file).

```
public void run() {
  BufferedReader br = null;
  Process process = null;
  Optional <Integer> exitCode = Optional.empty();

  ProcessBuilder pb = new ProcessBuilder();
  //dichiarazione degli argomenti del processo
  pb.command(commands);
  //modifica directory di lavoro del processo
  pb.directory(new File(workingDir));
  //si reindirizza lo standard error per poterlo leggere
  pb.redirectError(ProcessBuilder.Redirect.PIPE);
  //lo standard output e' gestito dal processo
  pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
  //avvio del processo
  process = pb.start();

  //decoratori per facilitare la lettura della stream
  br = new BufferedReader(new
      InputStreamReader(process.getErrorStream()));

  String line = null;
  //lettura standard error fintanto che il processo non e' terminato
  while ((line = br.readLine()) != null) {
     //si scatena l'evento OnProgress una volta acquisita la durata
     if (line.startsWith("frame=")) {
        String time = parseProgressInfoLine(line);
        Duration duration = DurationUtils.parseDuration(time);
           progress(new ProgressEventArgs(Optional.of(duration),
               null));
     }
  }
  //metodo bloccante, ritorna l'exit code alla chiusura del processo
  exitCode = Optional.of(process.waitFor());
  //alla chiusura del processo si scatena l'evento OnComplete
  completed(new CompletedEventArgs(exitCode, null));
}
```

Si specifica che è stato necessario utilizzare questo tipo di approccio ad eventi poiché il sistema doveva essere in grado gestire e sincronizzare delle operazioni asincrone. Grazie a questa implementazione è stato possibile definire handler che venissero eseguiti una volta terminate altre operazioni.

La conversione WEBM-DASH è un esempio in cui sono stati sfruttati questi meccanismi. Infatti in questo tipo di conversione è stato necessario sincronizzare tre diverse operazioni. Nello specifico, una volta avviato il processo di encode del flusso RTSP è possibile creare il manifest MPD solo dopo la prima comunicazione di progresso, poichè significa che tutti i file necessari per la creazione sono disponibili.

Oltre a questo, una volta terminato il processo di codifica, è necessario modificare il manifest MPD per renderlo statico e aggiungere alcuni tag necessari per trasformare la stream da live a on demand. In questo caso quindi si è creato un nuovo listener per l'evento di completamento nel quale si crea ed avvia un nuovo worker della classe MPDManifestUpdater adibito alla creazione o modifica di tag o nodi XML.

Di seguito è riportato lo schema gerarchico delle varie implementazioni dei worker presenti nel framework.

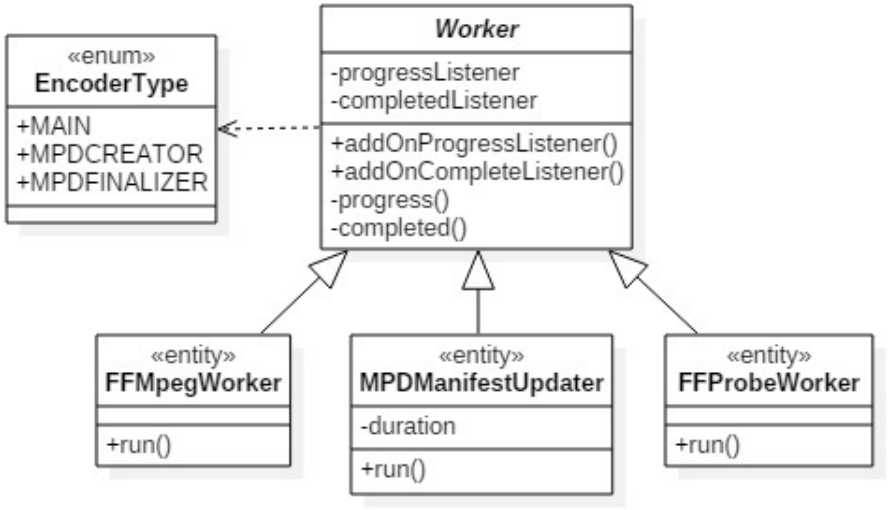


Figura 3.11: Diagramma delle classi inerente ai workers

Proseguendo la descrizione dell'implementazione si può considerare la classe ConfigManager. Per soddisfare i requisiti di flessibilità e configurabilità è stato infatti previsto che il sistema potesse importare all'avvio un file di configurazione, nel quale fossero dichiarati alcuni parametri valori con cui configurare il comportamento del sistema.

Nel file è possibile configurare la path base di tutte le route della REST API, l'IP e la porta di ascolto del server http, oltre ad alcuni parametri di default, pattern utilizzati per il parsing dell'output dei processi di conversione e, infine, parametri aggiuntivi di ffmpeg per tutti i tipi di encode.

La gestione della configurazione si basa sull'uso della classe ConfigManager implementata utilizzando il pattern Singleton per rendere disponibile in qualunque parte del codice un metodo per ottenere i valori di configurazione dichiarati dall'utente. I dati sono contenuti dentro un oggetto della classe Configurazione, che espone metodi *getter* per ottenerne i valori. In figura [3.18] è rappresentato il diagramma delle classi inerente alla parte di gestione della configurazione.

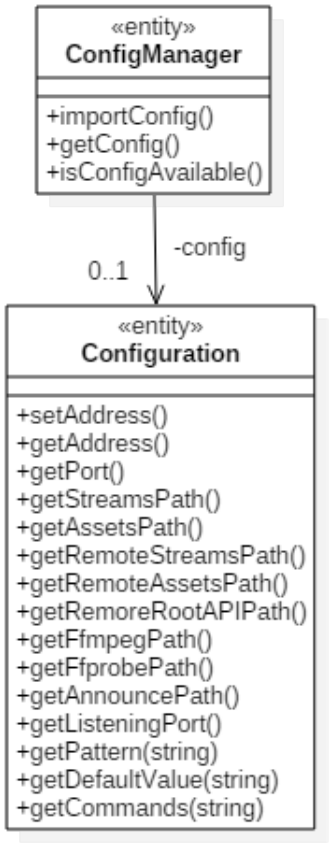


Figura 3.12: Diagramma delle classi inerente alla gestione della configurazione del servizio

Altre considerazioni possono essere fatte in merito al salvataggio dello stato del sistema alla chiusura. Esso infatti tiene traccia di tutte le stream corrente-

mente attive, ma anche quelle che sono state convertite in flussi VOD (Video on demand), quando è stata rilevata la chiusura da parte del client producer della stream RTSP.

Sono stati quindi implementati due metodi nel MainVerticle. Il primo che si occupa, alla chiusura, di serializzare su disco, in formato JSON, il contenuto della collezione contenente tutte le stream correntemente gestite dal servizio; mentre il secondo, che si occupa di deserializzare, se presente, il file contenente le stream salvate in precedenza e di controllare se tutti i file necessari al playback di ciascuna siano ancora presenti su disco.

Altro dettaglio implementativo riguarda la scelta del player per il playback dei flussi DASH lato web app. È stato scelto in questo caso shaka-player, un player Javascript open-source manutenuto da Google, specifico per la riproduzione di contenuti in formato adattivo.

È comunque opportuno citare anche il player dash.js, che presenta caratteristiche e performance molto simili. Dai test effettuati, entrambi possono essere utilizzati indistintamente senza particolari problemi. Infine, una ulteriore nota

deve essere fatta per quanto riguarda la gestione di un problema di concorrenza nel MainVerticle. Essendo tutti i metodi eseguiti in modo asincrono sulla base degli eventi della libreria Vert.x si potrebbe verificare una inconsistenza nella memoria dovuta a delle race conditions per quanto riguarda la collezione di stream.

Si è reso quindi necessario dichiarare *synchronized* tutti quei che metodi che presentavano un accesso a questa risorsa, in modo tale da rendere atomiche le operazioni di lettura e scrittura della collezione.

3.6 Funzionamento

In questa sezione si andrà a descrivere brevemente il funzionamento del sistema in un tipico scenario di utilizzo. La condizione iniziale del sistema è quella presentata in [figura:3.13], dove la web app interroga il servizio in una path specifica richiedendo la lista di stream disponibili.

In questo caso si nota che il servizio presenta alcune stream in diversi formati.

La web app ogni 10 secondi (valore modificabile) esegue una nuova richiesta al server per poter ottenere dati aggiornati e mostrare all'utente eventuali stream appena create.

Le prime quattro stream in figura sono delle stream di tipo VOD (Video on demand), questo rappresenta tutte quelle stream passive che sono già state

interamente processate dal server. La riproduzione di queste stream è la semplice visualizzazione della registrazione completa di un particolare client che non è più attivo.

Discorso diverso deve essere fatto per quanto riguarda gli ultimi due elementi. In questo caso le stream permettono di visualizzare un flusso live che è correntemente generato da un client producer ancora attivo e in fase di registrazione. Riproducendo questo tipo di stream, il player, riconoscendo tramite il manifest DASH che il contenuto è live, avvia il playback ricercando automaticamente l'ultimo chunk disponibile in modo tale da riprodurre il contenuto con la minore latenza possibile.

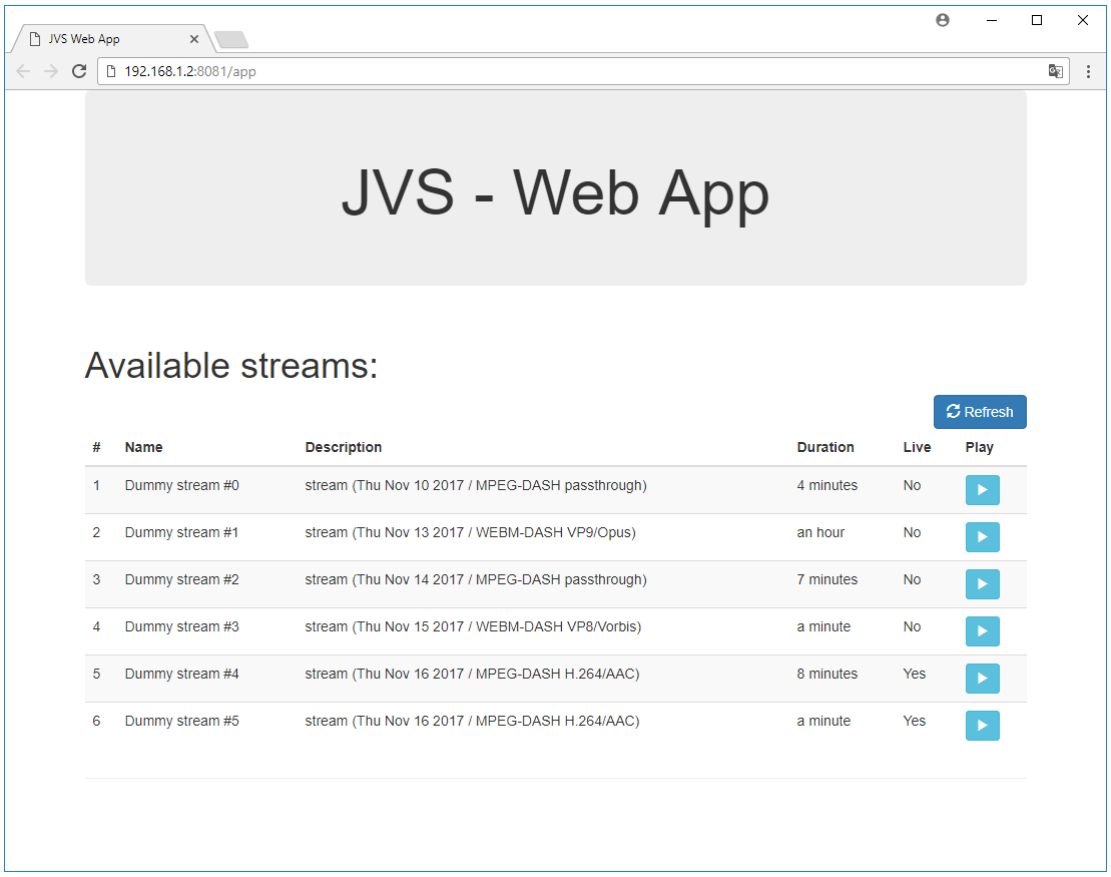


Figura 3.13: Esempio di visualizzazione sessioni disponibili tramite web app

Per avviare il playback di una stream è sufficiente premere sul bottone alla fine di ogni elemento della tabella presente all'interno della pagina. Una volta premuto la web app richiede al server dettagli specifici per la stream. Dopo aver ottenuto la risposta dal server inizializza il player, mostra l'elemento all'interno del DOM collegato al player e avvia la riproduzione [figura:3.14].

Per concludere la riproduzione di una stream è sufficiente utilizzare il bottone in alto a destra per ritornare alla visualizzazione predecente.

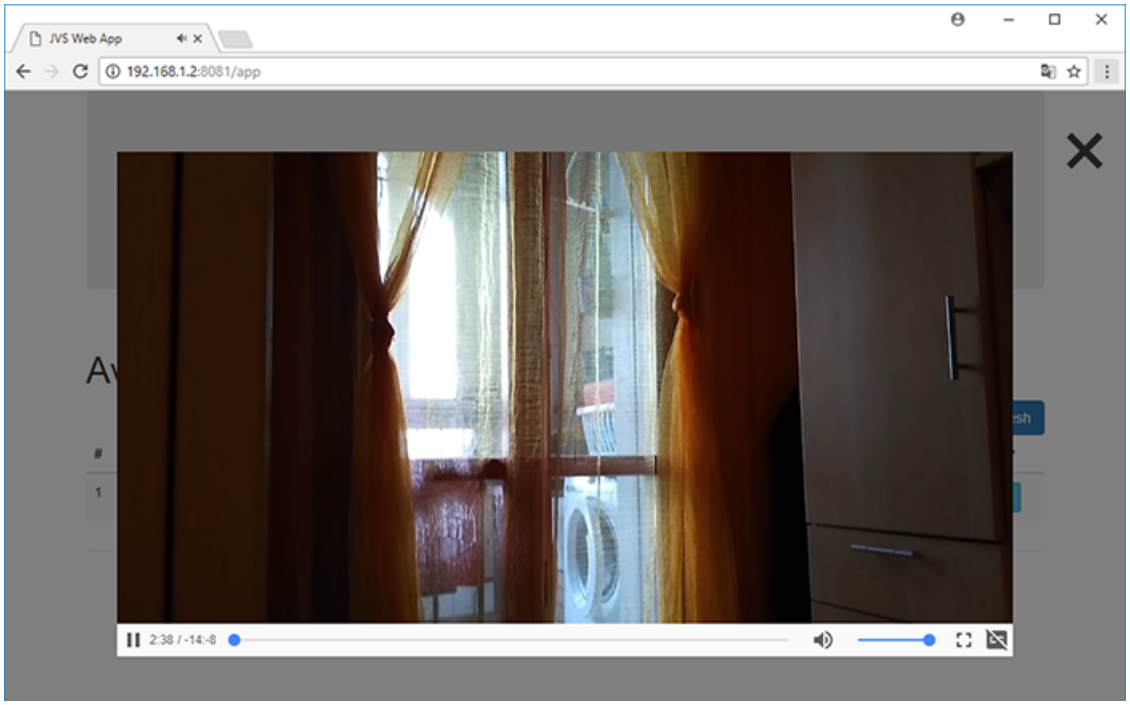


Figura 3.14: Esempio di playback sessione live tramite web app

Capitolo 4

Embedding in Trauma Tracker

In questo capitolo si descriverà come è stato eseguito l'embedding del framework all'interno del progetto Trauma Tracker. Di seguito se ne riporta lo schema dell'architettura logica con evidenziati i nuovi componenti del framework sviluppato.

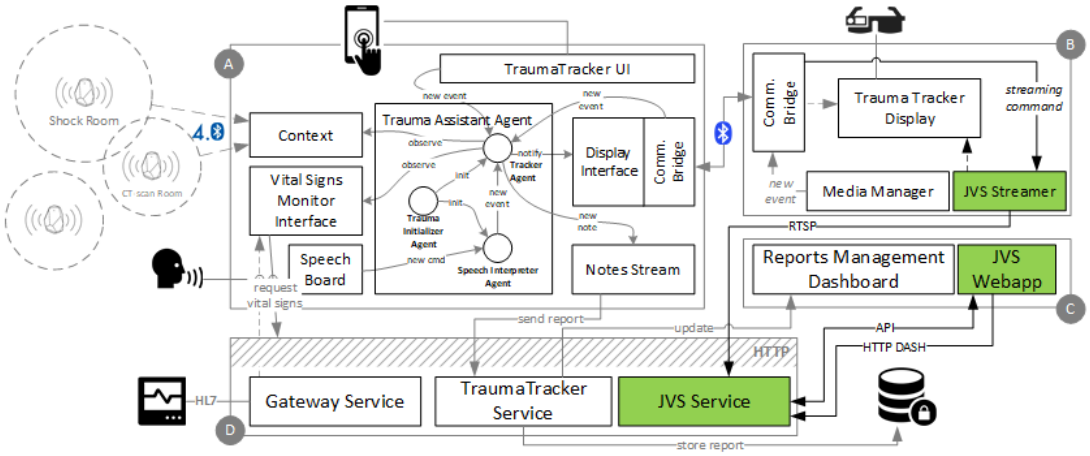


Figura 4.1: Schema dettagliato sull'architettura di TraumaTracker

4.1 Server

Come analizzato nel secondo capitolo, TraumaTracker presenta la cosiddetta Intrastruttura GT^2 (parte D della [fig:4.1]), un livello intermedio che si pone tra le applicazioni e i monitor all'interno dell'ospedale, costituito dai vari service che espongono le API per accedere e modificare i dati del sistema. Il servizio server del framework è quindi stato collocato in questa parte.

In questo modo il servizio rende disponibile le nuove funzionalità di video streaming a tutti gli utenti e a tutti i membri del TraumaTeam, nella Local Area Network (LAN) dell'ospedale, isolata dall'esterno e dunque protetta.

Essendo i servizi attualmente attivi su Trauma Tracker basati sulla libreria Vert.x, la stessa libreria utilizzata per il servizio di video streaming, ed essendo stato predisposto per accettare nuovi servizi, il sistema non dovrebbe avere problemi nell'accettare questo nuovo servizio, dopo aver configurato correttamente le porte utilizzate da ciascun service e aver controllato che le path di accesso alle API siano univoche.

Oltre al deploy, questo componente non ha necessità di ulteriori modifiche per poter funzionare tranne per la modifica del file di configurazione per la dichiarazione dell'indirizzo su cui fare il binding, la porta da utilizzare per permettere l'accesso all'api e quella per ricevere stream RTSP.

4.2 Client Producer

Per quanto riguarda l'applicazione Android su smartglasses (parte B della [fig:4.1]), per il client producer è stata sviluppata una classe apposita per la comunicazione con il server e l'incapsulamento di tutta la gestione della registrazione ed invio dei pacchetti RTSP. È stata implementata utilizzando il pattern singleton ed espone due semplici metodi per avviare e terminare la registrazione e lo streaming oltre alla relativa comunicazione al servizio di backend.

In questo caso, per evitare un consumo eccessivo di batteria e perché non era richiesto dalle specifiche del sistema, in quanto la maggior parte del display dei glasses è già occupata da altre informazioni, non è stato previsto nessun elemento grafico per mostrare la preview della registrazione.

Considerando come il progetto TraumaTracker è strutturato, con uno smartphone o tablet in comunicazione con gli smartglasses tramite bluetooth, la maggior parte degli input dell'utente vengono processati direttamente nell'applicazione principale lato dispositivo mobile (parte A in [fig:4.1]).

Presumibilmente quindi il sistema potrebbe prevedere nuove gestures o comandi vocali, interpretabili come nuovi comandi specifici per il servizio di video streaming che potrebbero essere gestiti inviando un nuovo tipo di messaggio tramite bluetooth agli glasses. Questo risulterebbe molto comodo per l'utilizzatore, che avrebbe in questo modo la possibilità di attivare lo streaming senza la necessità di agire fisicamente nel dispositivo.

In alternativa, sfruttando i *beacon*, il sistema potrebbe decidere di avviare automaticamente lo streaming, notificando l'utente e proponendogli nel ca-

so l'annullamento dell'operazione, nel caso in cui l'utente entrasse in un'area specifica.

Sarebbe inoltre opportuno prevedere una sistema per comunicare all'utente l'inizio o la fine di uno streaming, preferibilmente nell'interfaccia dei glasses, banalmente per mezzo di un testo o un'icona.

Infine, per gestire adeguatamente l'utilizzo della batteria e prevenire lo spegnimento completo dei glasses, il sistema potrebbe monitorare costantemente la percentuale di carica e, ove lo ritenesse necessario, bloccare lo streaming, sempre permettendo all'utente di annullare l'operazione.

4.3 Client Consumer

Per quanto riguarda la parte inerente alla riproduzione delle stream, TraumaTracker prevede già una web app dove sono presenti diverse tab per visualizzare i vari report o l'elenco degli eventi registrati. Per integrare il sistema di video streaming, sarebbe quindi sufficiente aggiungere una nuova tab, nel quale mostrare la pagina con l'elenco delle stream disponibili e i vari bottoni per avviarne la riproduzione.

Anche in questo caso non ci sarebbe la necessità di operare modifiche al codice poiché il suo funzionamento dipende esclusivamente dall'API del framework e da normali richieste HTTP, nel caso del playback delle stream.

Come già si era discusso nel secondo capitolo, si è cercato grazie a questo framework di soddisfare le richieste dei medici coinvolti, i quali chiedevano principalmente un modo più pratico e potente per registrare gli eventi durante le attività del TraumaTeam, sia per scopi legali, sia soprattutto per poter ricevere assistenza remota in tempi molto brevi da altri medici non presenti fisicamente all'interno della shock room in situazioni critiche.

Si prevede inoltre che il progetto TraumaTracker possa essere esteso anche nelle attività antecedenti il pronto soccorso, ovvero nelle attività di prelevamento dell'infortunato nel luogo dell'incidente. Anche in questo caso il servizio di video streaming potrebbe rilevarsi molto utile per permettere al TraumaTeam in ospedale di verificare le condizioni del paziente ancora prima del suo arrivo nella struttura medica e, nel caso, pianificare in anticipo eventuali attività da eseguire.

Capitolo 5

Validazione e discussione

In questo capitolo si andranno a descrivere le metodologie di testing utilizzate per l'acquisizione delle statistiche del sistema, grazie alle quali è stato successivamente possibile eseguire la validazione del sistema, comparandoli con i requisiti per capire se, e in che misura, sono stati soddisfatti. L'obiettivo è principalmente quello di capire come il sistema si comporta in uno scenario reale e comprendere a pieno le sue potenzialità. Si specifica che il testing non è stato eseguito all'interno del progetto TraumaTracker, perché l'obiettivo principale era quello di creare un servizio generico.

5.1 Scenario di utilizzo

Come scenario di test si è cercato di emulare un tipico caso di utilizzo del framework in combinazione al progetto TraumaTracker. Si presuppone quindi che il client producer sia rappresentato da un qualunque dispositivo Android Wearable, come per esempio un paio di smartglasses Vuzix M300 piuttosto che un tablet o uno smartphone su cui è in esecuzione l'applicazione di test (o indifferentemente l'app TraumaTrackerCore) che sfrutti la libreria Android sviluppata.

Il servizio del framework e tutti i client risultano essere nella stessa intranet (rete locale e privata generalmente isolata dalla rete internet esterna), come se fossero all'interno dell'ospedale. Si considerano inoltre tutti i client producer collegati tramite rete Wi-Fi interna, mentre una parte di quelli consumer potrebbero avere anche la disponibilità di una rete cablata, sempre all'interno dell'ospedale.

Sono quindi state avviate due sessioni simultanee di streaming, una proveniente dagli smartglasses Vuzix e l'altra da uno Smartphone Huawei 5X, entrambi con sistema operativo Android 6.1 Marshmallow. Dopodiché sono state aperte tre istanze della web app in tre dispositivi diversi, due collegati

tramite rete wireless ed uno tramite rete cablata, ciascuno dei quali ha iniziato a riprodurre una delle due stream live disponibili.

Entrambi i client sono stati configurati in modo da inviare un flusso video a 1280x720 pixel, con un framerate di 30 frame al secondo, un bitrate fissato di circa $2000 \cdot 2^{10}$ bit/sec, codificato, con supporto hardware dal SoC integrato, in H.264.

Per quanto riguarda il flusso audio è stato scelto un flusso stereo codificato in AAC-LC, con un sample rate di 16000 Hz e un bitrate costante di $64 \cdot 2^{10}$ bit/sec.

Infine si specifica che per entrambi è stata utilizzata la configurazione MPEG-DASH (H.264/AAC) per quanto riguarda la riconversione lato server.

5.2 Risultati

Nessuno dei tre client consumer ha subito particolari ritardi per quanto riguarda il playback, la latenza media di riproduzione, rispetto al tempo reale, si è aggirata attorno ai 5 secondi, latenza intrinseca del sistema dovuta al tempo minimo di buffering del player. La qualità video è risultata essere molto buona con un audio sufficientemente chiaro e senza particolari distorsioni.

Lato server, ciascuna istanza di ffmpeg ha occupato circa il 17% del tempo CPU (quadcore), mentre il trasferimento di dati tramite network ha avuto dei picchi massimo di 3 Mbps.

Le misurazioni per quanto riguarda i client producer sono state ottenute tramite il tool integrato all'IDE Android Studio 3.0, chiamato Android Profiler.

In figura [fig:5.1] viene mostrato lo screenshot della finestra di monitoraggio del tool durante il quale il client stava effettuando la registrazione, conversione e trasferimento del flusso RTSP.

Come si vede chiaramente, il consumo di CPU rimane fisso al 15%, l'utilizzo della memoria poco sopra gli 80 MB, mentre l'attività di rete presenta picchi fino a 600 Kbps ad intervalli regolari di circa un secondo, ovvero il momento nel quale viene inviato un pacchetto nuovo RTSP.

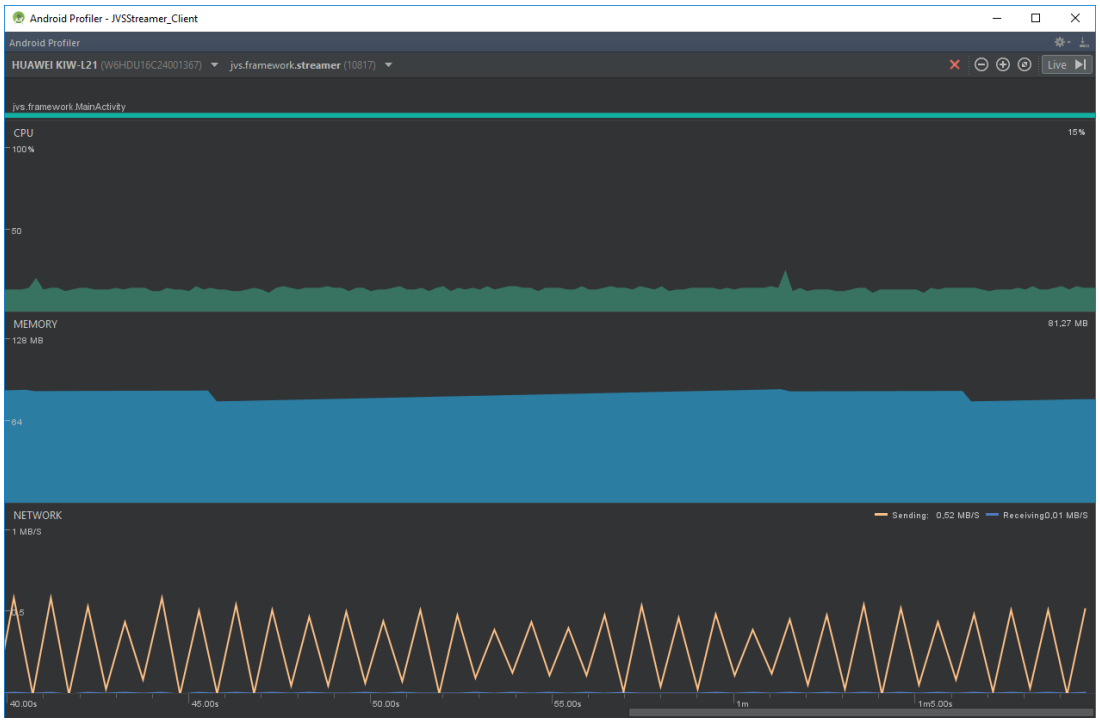


Figura 5.1: Statistiche di Android profiler durante lo streaming

Per quanto riguarda il consumo di batteria si sono eseguite tre misurazioni e si è fatta la media dei valori ottenuti. È risultato che, nella media, il sistema rileva un calo di un punto percentuale ogni 2 minuti e 30 secondi, nelle condizioni descritte in precedenza.

5.3 Validazione

Esposti quindi i risultati, si passa ora alla validazione del sistema. Uno dei requisiti fondamentali del sistema era la scalabilità. In questo caso il sistema si è comportato adeguatamente con multipli e concorrenti client di entrambe le tipologie, si può intendere quindi questo requisito pienamente soddisfatto.

Si è inoltre raggiunto anche una buona efficienza per quanto riguarda i client, che, limitandosi alla connessione punto-punto con il server e con la possibilità di sfruttare accelerazioni hardware per la codifica dei flussi, hanno registrato un consumo accettabile.

Altro aspetto positivo è il fatto che il sistema permette una estesa configurabilità per poter essere ottimizzato in diversi scenari. Infatti, grazie alla possibilità per ciascun di client di scegliere i profili di codifica dei flussi lato server, si riesce ad aggiungere l'alta adattabilità richiesta.

Il sistema risulta essere anche molto estendibile poiché, grazie all'utilizzo del tool ffmpeg, è possibile modificare il sistema per accettare nuovi tipi di protocolli o tecnologie, presupponendo che siano stati implementati dalla libreria FFmpeg, semplicemente cambiando o estendendo le classi adibite alla generazione dei comandi.

L'ultimo requisito mancante da analizzare è la reattività del sistema, che è stato soddisfatto parzialmente. Il sistema riesce a raggiungere una latenza minima di massimo 5 o 6 secondi per quanto riguarda le sessioni live, questo perché la latenza risulta essere inversamente proporzionale alla scalabilità, come già riportato nel primo capitolo. È stato infatti scelto MPEG-DASH poiché risulta il protocollo che permette un buon compromesso tra entrambe le caratteristiche.

Per diminuire al minimo la latenza del sistema si è cercato di abbassare la dimensione dei segmenti creati e di forzare il codec a generare *keyframe* all'inizio di ciascuno di essi mantenendo la grandezza del GOP (il numero di frame) pari al framerate (numero di frame per secondo) della stream per la dimensione del segmento.

Per comprendere il motivo per cui si è adottato questo tipo di approccio, è necessario capire come viene eseguita la compressione dei dati. Nello specifico, un keyframe è un tipo di fotogramma nella compressione video, noto anche come I-frame, utilizzato dai codec per eseguire la compressione delle immagini che costituiscono una video stream. I frame possono essere di tre tipologie e vengono sono raggruppati in strutture GOP, group of pictures, le quali ne specificano l'ordine e le loro relazioni.

Le tipologie sono:

- *I-frame*: Intra-coded frame, è un'immagine completamente descritta, utilizzata come rifermento per gli altri due tipi di frame

- *P-frame*: Predicted frame, contiene sono la parte dell'immagine che è stata modificata rispetto ai fotogrammi precedenti. In questo modo è possibile risparmiare spazio, anche se può essere decodificato solo quando sono stati ottenuti tutti frame precedenti fino al primo I frame

- *B-frame*: Bi-predictive picture, è il frame che contiene meno informazioni di tutti poiché codifica la differenza tra i frame precedenti e quelli successivi. Anche in questo caso quindi non è possibile decodificarlo senza prima aver ottenuto tutti i frame all'interno del GOP in cui è contenuto

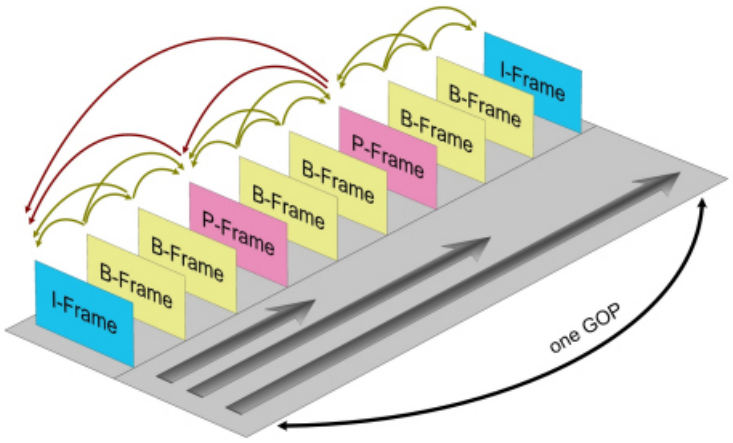


Figura 5.2: Struttura di un GOP e relazione tra i frame

I codec video cercano quindi di codificare il maggior numero di frame nel tipo P e B, per ottenere la compressione massima del contenuto. Tuttavia in ambito di streaming questo particolare comportamento risulta essere un problema, poiché essendo il contenuto frammentato e riprodotto separatamente, se in un segmento mancasse un GOP completo, il player non riuscirebbe a decodificare correttamente tutti i frame presenti nel pacchetto, il che risulterebbe in artefatti durante la riproduzione o la necessità di scaricare altri pacchetti, aumentandone quindi il ritardo.

5.3.1 Validazione in WAN

Si vuole ora dedicare una piccola parte per la descrizione dei risultati ottenuti utilizzando una WAN (wide area network). La differenza sostanziale rispetto alla intranet è il tipo di interconnessione tra i dispositivi. In questo caso quindi il server utilizza un ip statico (o un DDNS e ip dinamico) e si presuppone che le porte utilizzate dal servizio per l'accesso alla API e la porta in ascolto per flussi RTSP non siano bloccate.

I client conoscono l'indirizzo del server e lo utilizzano per accedere alle funzionalità del servizio. In questo caso il client producer potrebbe avere delle difficoltà se fosse utilizzato come server, quindi è necessario, in questo scenario, di utilizzare la modalità in cui è il client ad eseguire announce del suo flusso RTSP al servizio lato backend configurato come server RTSP.

Se tutte queste condizioni sono soddisfatte il sistema funziona correttamente e presenta, almeno lato server, le stesse caratteristiche dello scenario descritto in precedenza.

Per quanto riguarda i client consumer, si presuppone che siano collegati tramite rete dati (presumibilmente in LTE o 3G) e che siano in movimento.

In questo scenario alcuni pacchetti RTSP potrebbero essere persi causando, per un brevissimo periodi di tempo, un blocco per mancata bufferizzazione del pacchetto successivo e artefatti durante la riproduzione. La magnitudine di questi effetti è in relazione alla quantità di pacchetti persi. Detto ciò, essendo il flusso RTSP ottimizzato per la latenza e il recupero in caso di errori di rete, configurando il codec per generare un numero adeguato di keyframe, questi fenomeni potrebbero essere ridotti significativamente.

Relativamente allo scenario appena riportato, è stato condotto un test, dal quale si sono ottenuti risultati abbastanza in linea con quelli riscontrati nello scenario su intranet. Le divergenze principali sono nel consumo di batteria dei dispositivi mobile relativi ad entrambe le tipologie di client, causato principalmente dall'utilizzo di rete dati, che è notoriamente la rete più esosa in termini di batteria.

I principali problemi sono quindi causati dalla rete, che, essendo indipendenti dal sistema, non ne pregiudica il soddisfacimento dei requisiti, tuttavia si potrebbe presentare un nuovo problema legato alla sicurezza del servizio. Essendo il sistema accessibile da chiunque, sarebbe opportuno un meccanismo di crittografia e di controllo degli accessi, o un sistema di account.

5.4 Possibili miglioramenti

In questa sezione si vanno ad esporre alcuni possibili miglioramenti del sistema che, pur essendo stati pensati, non sono stati implementati nel sistema. Di seguito ne vengono citati alcuni tra i più importanti:

- *Ripristino sessioni*: in situazioni di scarsa copertura o di gravi problemi alla rete, sarebbe opportuno implementare un meccanismo per fare in modo che una sessioni possa essere ripresa, nel caso in cui un client perdesse la connessione col servizio. Una possibile soluzione a questo problema potrebbe essere quella di assegnare ad ogni coppia client-stream un identificativo univoco da fornire al client come risposta alla prima richiesta effettuata. In questo modo il client, in caso di perdita di connessione, potrebbe decidere di inviare una nuova richiesta al servizio comunicando anche questo identificativo univoco, che il server potrebbe gestire come modifica di una sessione già presente

- *Crittografia*: nel caso di studio considerato la sicurezza della connessione e la protezione dei dati non sembra particolarmente importante in quanto l'intero sistema funziona all'interno di una rete locale che dovrebbe già fornire di per sé la protezione necessaria. Ma, se si scegliesse di fare

il deploy del sistema all'esterno, in questo caso sarebbe opportuno aggiungere il supporto al protocollo HTTPS e implementare un sistema di gestione di account per quanto riguarda la web app

- *Latenza*: sono già state fatte considerazioni riguardanti questo aspetto nel primo capitolo. Tuttavia, un possibile miglioramento potrebbe essere raggiunto semplicemente ricercando più accuratamente una configurazione ancora più efficiente per quanto riguarda la codifica lato server. In generale, la latenza dovrebbe diminuire rimpicciolendo i segmenti generati, questo tuttavia risulterebbe in un calo di efficienza del codificatore e potrebbe aumentare il costo dei trasferimenti a causa dell'incremento del numero di segmenti, a causa di un maggior numero di richieste HTTP utilizzate per prelevarli

Conclusioni

I dispositivi di wearable computing possono introdurre notevoli miglioramenti, soprattutto grazie al fatto di poter essere utilizzati senza impegnare le mani e senza ostruire la visione dell'utilizzatore. Un caso esemplare è quello del sistema *TraumaTracker*, dove grazie all'utilizzo di questi sistemi, combinato ad una infrastruttura di supporto, si è riuscito ad ottimizzare e migliorare l'efficacia di tutte le persone coinvolte nelle attività di primo soccorso, fornendo ai membri la possibilità di accedere ad informazioni critiche, in modo comodo e veloce.

Per quanto visto in precedenza, i dispositivi introducono anche nuovi scenari di utilizzo, con la presenza di un ecosistema sempre più vasto di sensori e il loro perfezionamento, che permettono di ragionare su nuovi possibili applicazioni, come per esempio il video streaming. Tuttavia, a causa della natura dei "dispositivi indossabili" è comunque necessario rivedere alcuni paradigmi di programmazione odierni. Lo sviluppo deve essere perciò incentrato sull'utente, attorno al quale deve essere costruito il sistema, la cui progettazione richiedere di tenere in considerazione i suoi limiti e le sue esigenze. Infatti questo tipo di dispositivi presentano una serie di limiti hardware che non possono essere ignorati, soprattutto in ambiti applicativi come quello considerato, dove lo sfruttamento delle risorse del sistema è intrinsecamente elevato.

Si vede quindi, sia nel Framework di Video Streaming che in TraumaTracker, la necessità di cooperazione tra dispositivi anche molto diversi, per progettare ed implementare sistemi robusti. È necessario quindi studiare il dominio applicativo, definendo i ruoli dei principali sistemi coinvolti e le proprietà hardware e software di cui devono disporre. In questo caso infatti, poiché lo streaming dai dispositivi mobile sarebbe risultato troppo pesante da gestire, si è spostata la logica di accesso e la gestione delle risorse in un server, dove vi è sufficiente capacità computazionale e non vi sono problemi tipici della categoria di dispositivi mobile, come, per esempio, la durata della batteria.

Il codice sorgente del progetto è disponibile al seguente indirizzo:
`https://bitbucket.org/DavideGia/java-video-streaming-framework`

Ringraziamenti

A conclusione di questo lavoro desidero ringraziare tutti coloro che hanno in qualche modo contribuito, direttamente o indirettamente, a questo progetto. In primis, la mia famiglia, per avermi dato la possibilità di intraprendere questo percorso, supportandomi costantemente e spronandomi a dare il meglio di me.

Un ringraziamento speciale va al Prof. Ricci per avermi offerto la possibilità di lavorare su questo progetto, grazie al quale ho potuto mettere in pratica le conoscenze acquisite durante questo corso di studi, acquisirne di nuove e produrre qualcosa di utile per la comunità. Vorrei inoltre ringraziarlo per la sua completa disponibilità, per i preziosi insegnamenti forniti e per la sua instancabile passione. Successivamente desidero ringraziare il Dott. Croatti per il supporto e i consigli offertomi durante tutto lo sviluppo del progetto.

Infine, un ringraziamento va anche ai miei compagni di università, che mi hanno accompagnato durante il corso degli studi.

Bibliografia

[1] S. Jhajharia, S. K. Pal, S. Verma *Wearable Computing and its Application*, 2014.

[2] M. Chan, D. Est'eve, J.-Y. Fourniols, C. Escriba, E. Campo *Smart wearable systems: Current status and future challenges* Elsevier, 2012.

[3] A. Ricci, A. Croatti, S. Montagna, V.Agnoletti *Tracking and Assisting Activities in Trauma Management: The TraumaTracker case, Submitted to 11th EAI International Conference on Pervasive Computing Technologies for Healthcare*, May 2017, Barcelona, Spain.

[4] P. Lukowicz, T. Kirstein, G.Troster *Wearable Systems for Health Care Applications*, Schattauer GmbH, 2004.

[5] W. Barfield *Fundamentals of Wearable Computers and Augmented Reality*, CRC Press, 2016.

[6] W. Seongmin *Inside Vert.x. Comparison with Node.js*, 2013.

[7] J. Kim *Understanding Vert.x Architecture - Part II*, 2013.

[8] *Vert.x*, http://vertx.io/

[9] H. Schulzrinne *Internet Media-on-Demand: The Real-Time Streaming Protocol*, December 2001, New York, New York.

[10] H. Schulzrinne, A. Rao, R. Lanphier *Real Time Streaming Protocol (RTSP)*, February 1998, New York, New York.

[11] H. Schulzrinne *RTP: A Transport Protocol for Real-Time Applications*, July 2003, New York, New York.

[12] M. Handley *SDP: Session Description Protocol*, July 2006, Glasgow, Scotland.

[13] *JSON*, http://json.org

[14] *FFMpeg*, https://www.ffmpeg.org

[15] *WebM*, https://www.webmproject.org

[16] S. Hashemizadehnaeini *Transcoding H.264 Video via FFMPEG encoder* Politecnico di Milano, 2015.

[17] R. Kollar *Configuration of FFmpeg for High Stability During Encoding* 2014, Brno, Czech Republic.

[18] J. Le Feuvre *Adaptive HTTP Streaming and the MPEG-DASH standard*, January 2016, Paris, France.

[19] T. Stockhammer *MPEG's Dynamic Adaptive Streaming over HTTP (DASH) - An Enabling Standard for Internet TV*, September 2011.

[20] A. Vetro *The MPEG-DASH Standard for Multimedia Streaming Over the Internet*, 2011.

[21] I. Sodagar *MPEG-DASH: The Standard for Multimedia Streaming Over Internet*, September 2011, Redmond, Washington.

[22] *MPEG systems technologies — Part 6: Dynamic adaptive streaming over HTTP (DASH)*, January 2011.

[23] F. Fund, C. Wang, Y. Liu, T. Korakis, M. Zink, S. S. Panwar *Performance of DASH and WebRTC Video Services for Mobile Users*, 2013.

[24] V. Swaminathan, K. Streeter, I. Bouazizi, F. Denoual, F. Mazé *DASH with Server Push and WebSockets*, February 2016, San Diego, California.