Università degli Studi di Milano

Facoltà di Scienze e Tecnologie

Corso di Laurea in Comunicazione Digitale

PROGETTAZIONE E REALIZZAZIONE DI UN SISTEMA DI CAPTURE DI IMMAGINI DIGITALI E STREAMING VERSO UN PLAYER DISTRIBUITO

Relatore: Prof. Alessandro Rizzi

Correlatore: Rosario Crispo

Tesi di:
Riccardo Macoratti
Matricola: 801749

Anno Accademico 2014 - 2015

Prefazione

L'intento di questo scritto è di spiegare le modalità di realizzazione del progetto riguardante la progettazione e lo sviluppo di un sistema di acquisizione di immagini digitali e successivo streaming verso un riproduttore multimediale distribuito (anche definito videowall). Il progetto è stato svolto integralmente nell'ambito del tirocinio formativo curriculare di 14 settimane, svolto presso l'azienda OffiCINE Srl, in collaborazione con DataSoft Srl, tra il 27 maggio 2015 e 30 settembre 2015.

Organizzazione della tesi

L'elaborato è organizzato come segue:

- nel capitolo 1 vengono introdotti i sistemi di digital signage e live streaming e sono descritte le motivazioni e gli scopi che hanno portato alla realizzazione di questo progetto, ovvero la condivisione sotto una licenza aperta e il requisito della portabilità;
- nel capitolo 2 viene descritto qual è lo stato dell'arte dei principali sistemi di realizzazione di videowall presenti sul mercato e in che aspetti il progetto trattato si differenzia da tali sistemi;
- al capitolo 3 viene illustrato il modello impiegato per lo sviluppo dell'applicazione; inoltre viene fornita una spiegazione all'organizzazione semantica del codice, ai dettagli riguardanti l'implementazione e vengono descritti gli strumenti di riferimento che sono stati utilizzati in corso di realizzazione;

- nel capitolo 4 vengono spiegati i punti critici e le difficoltà riscontrati durante lo sviluppo e le rispettive soluzioni adottate, con particolare enfasi sull'analisi delle performance in fase di streaming del software realizzato;
- nel capitolo 5 viene introdotta la correzione del colore e illustrata la modalità con la quale si è proceduto ad effettuare l'adattamento cromatico delle immagini visualizzate sul *player* distribuito;
- infine, nel capitolo 6, sono riportati i risultati raggiunti e le ipotetiche migliorie effettuabili avendo a disposizione tempi di sviluppo più ampi, insieme alle possibili future evoluzioni del progetto.

Ringraziamenti

Indice

Capitolo 1

Introduzione

1.1 Digital signage e Live streaming

A metà degli anni novanta, proseguendo fino ai giorni nostri, nell'ambito dell'informatica orientata all'immagine si è affermata l'esigenza di costruire dei sistemi multimediali che supportassero la presentazione di contenuti visivi di vario genere. Sul flusso creato da questo bisogno, sono nate diverse branchie nel settore che si occupano di svolgere compiti anche diametralmente opposti tra di loro.

Tra gli ambiti nei quali l'applicazione dell'informatica multimediale video ha avuto più successo, troviamo quello del *digital signage*, reso in italiano come "segnaletica digitale" e quello della presentazione di contenuti multimediali sia in tempo reale (perciò riguardanti un evento che si sta svolgendo contemporaneamente alla visualizzazione), che *on demand* dell'utente (quindi con possibilità che quest'ultimo richieda la visualizzazione in un ambiente spazialmente o temporalmente diverso da quello in cui si è svolto l'evento).

Altri settori dove la presentazione di contenuti multimediali risulta decisiva sono quelli dove è prevista una gestione condivisa all'interno di un gruppo ampio di persone o un monitoramento continuo e distribuito di qualche risorsa. Esempi sono i centri di controllo del traffico aereo e ferroviario oppure i centri di gestione di operazioni militari, le sedi degli organi istituzionali o delle unità di crisi.

Il digital signage è una nuova forma di pubblicità incentrata totalmente sull'invio

Figura 1: Times Square a New York (da Wikipedia).

di nuovi contenuti che possono essere di tipo statico, al pari dei cartelloni pubblicitari, o dinamico, aggiungendo una sfumatura che nella divulgazione pubblicitaria tradizionale non può evidentemente avvenire. La cartellonistica digitale che viene creata all'interno di quest'ambito trova una posizione, da un punto di vista commerciale, all'interno di un punto vendita, in luoghi che vedono transitare un gran numero di persone, oppure all'interno della sede centrale di una grande azienda multinazionale o di un importante organo istituzionale.

Oggi il digital signage è pressochè indispensabile in qualsiasi azienda che mira a costruire un'immagine di sé, in quanto le caratteristiche proprie di questa forma pubblicitaria, quali la malleabilità, la trasformabilità e l'alto rapporto qualità/prezzo, contribuiscono ad aumentare il valore di tale azienda, pur rimanendo in costi contenuti e facilmente sostenibili.

È importante distinguere nella maniera corretta le applicazioni che cadono nel dominio della segnaletica digitale da quelle che vanno a far parte dei centri di controllo: le prime hanno la caratteristica di essere per lo più statiche e di pura fruizione visiva, mentre le seconde vengono utilizzate in un ambito lavorativo, per questo motivo

necessitano di requisiti di affidabilità e durabilità adatti all'uso prolungato.

I "cartelloni digitali" oggetto del digital signage nella maggior parte dei casi vengono esposti al pubblico e hanno la necessità di essere mostrati in maniera tale da risultare vistosi ed appariscenti. Tale necessità spesso si traduce nelle grandi dimensioni fisiche dell'apparato che si occuperà di effettuare la *delivery* dei contenuti ai potenziali utenti (un esempio è riportare nella figura 1 a pagina 2.

I contenuti digitali che spesso si mostrano in questi "cartelloni multimediali" possono spaziare dalle semplici righe di testo (come nel caso della rete dedicata a fornire informazioni dei voli nell'aeroporto mostrato in figura 2 a pagina 4), alle immagini statiche raffiguranti delle *réclame*, fino ad arrivare, come accennato precedentemente, a veri e propri contenuti dinamici di natura televisiva, a cui può eventualmente essere abbinato anche dell'audio. Nei casi più estesi, dei quali un esempio può essere il sistema creato per la grande distribuzione organizzata, la rete di digital signage assume le vesti di un canale televisivo di ridotte dimensioni, con una vera e propria programmazione a scansione oraria e contenuti informativi e d'intrattenimento[1].

Per visualizzare il materiale multimediale, in maniera tale che fosse fruibile da un pubblico, nel corso degli anni si sono alternate soluzioni basate su schermi LCD dalla grande superficie oppure, per diagonali molto estese, su schermi al plasma[2]; negli ultimi tempi invece si è diffuso l'impiego di videoproiettori digitali, la cui qualità è gradualmente salita fino ad arrivare ad una paragonabile a quella televisiva. Tuttavia queste soluzioni, pur adattandosi e fornendo una buona qualità, sono a volte eccessivamente voluminosi (nel caso di LCD e plasma) oppure altre molto costosi (nel caso di proiettori digitali di buona qualità).

Per far fronte a simili problemi, l'informatica multimediale ha dovuto studiare dei prodotti che coniugassero la dimensione e i costi degli apparati di riproduzione con la

[1] Basti pensare ai monitor situati nelle stazioni delle metropolitane delle grandi città che mostrano delle vere e proprie trasmissioni, quali telegiornali, eventi e pubblicità.

[2] Gli schermi al plasma sono meno duraturi di quelli LCD (*L*iquid *C*rystal *D*isplay), ma leggermente meno costosi, per questa ragione vengono impiegati quando l'apparato di visualizzazione è molto esteso.

Figura 2: Esempio di digital signage in un aeroporto (da Wikipedia).

flessibilità che caratterizza i prodotti di cartellonistica digitale. Piuttosto che produrre degli schermi dalle cospicue dimensioni, che corrispondono quasi sempre a un peso consistente, oppure dei proiettori dalla risoluzione elevata, si è pensato di usare degli esemplari standard e di disporli in una particolare configurazione a matrice, in modo tale da formare un *videowall*.

Un videowall può essere descritto come una configurazione multimonitor, dove gli schermi vengono disposti assieme in maniera continua, o nel caso dei proiettori leggermente sovrapposti, con il fine di formare un apparato di visualizzazione più grande. In termini più rigorosi, un videowall è una matrice $m \times n$ di monitor di forma solitamente quadrata o rettangolare. dove ogni schermo (e conseguentemente ogni apparato che lo gestisce) prende in carico uno *slice* del video e lo visualizza (l'immagine 3 a pagina 6 mostra un esempio di videowall composto da quattro monitor).

Attraverso un videowall si possono realizzare diverse configurazioni:

- una possibilità è quella di sfruttare l'intera superficie visibile come un grande monitor, perciò ogni monitor ritaglia la parte del segnale video corrispondente

alla sua posizione nella matrice e la visualizza;

- un'ulteriore opzione è quella di inviare a tutti i monitor lo stesso segnale, oppure comporre dei gruppi di monitor che riceveranno segnali dello stesso tipo;
- l'ultima possibilità è un sistema ibrido tra le prime due, cioè una matrice contenuta in quella del videowall viene impiegata come un monitor unico, mentre ai restanti monitor si invia un segnale diverso o, come prima, si formano dei gruppi.

Con il diffondersi di queste tecnologie sono stati creati dei monitor appositi per la composizione di videowall; questi monitor montano cornici di ridotto spessore per cercare di ridurre il *mullion*[3], cioè lo spazio tra due zone attive del display. In aggiunta, sono già predisposti per il collegamento in serie e vengono costruiti con un'elettronica robusta dal punto di vista elettrico, dato che si vuole ottenere un prodotto che duri nel tempo.

Tra i fattori in cui possiamo ricercare i motivi per i quali i videowall si sono diffusi così tanto, oltre quelli già citati, figurano soprattutto i motivi economici: una configurazione videowall ha un costo più basso per metro quadrato e soprattutto una densità di pixel più alta, sempre per metro quadro, di un singolo monitor della stessa dimensione.

La segnaletica digitale presenta, inoltre, alcune peculiarità che la rendono sostanzialmente diversa, e per molti versi migliore, dei sistemi di *signage* tradizionali; è possibile notare tali peculiarità nella maggior parte dei sistemi costruiti: i contenuti che vengono mostrati al pubblico devono poter essere facilmente gestiti e modificati in maniera dinamica e deve essere possibile, escludendo la presenza fisica in loco, inviare un dato messaggio in un luogo specifico ad un tempo definito. È quindi naturale che per questo tipo di applicazioni siano state studiate e implementate delle reti di calcolatori; il *delivery* del materiale multimediale viene infatti fatto per mezzo di Internet

[3]Il *mullion*, traducibile letteralmente in italiano come "inglesina"; si tratta di un termine preso in prestito dall'architettura ed indica il montante divisorio tra due vetri di una finestra.

oppure viene direttamente creata una intranet di tipo LAN, la quale fungerà anche da interfaccia di controllo delle varie periferiche di output.

In un tale contesto, è naturale che si sia approfondito lo studio verso l'invio di contenuti digitali attraverso una rete (o via Internet), definito *streaming*. La capacità di una rete non è però illimitata e, per questa ragione, di pari passo allo sviluppo di soluzioni di streaming c'è stato quello delle tecniche di compressione di immagini digitali, sia statiche che in movimento.

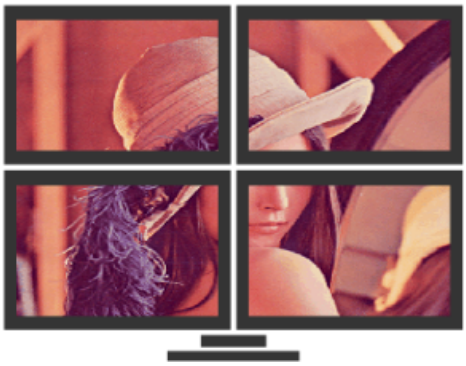

Figura 3: Esempio di monitor in configurazione videowall.

L'altro campo dove si è impiegata l'informatica video negli ultimi anni è, come precedentemente accennato, la presentazione di contenuti in tempo reale, soprattutto in occasione di fiere, manifestazioni e convegni; tale presentazione è incentrata sul successivo consumo da parte degli utenti. Un fattore che non si può tralasciare è che molto spesso tali utenti non sono fisicamente presenti all'evento, ma vi assistono da remoto.

In questo ambito, valgono ancora le considerazioni fatte finora per il digital signage, anche se i fattori di accentramento dell'attenzione sono diversi. Nella fruizione in tempo reale, c'è bisogno che il contenuto trasmesso sia immediatamente disponibile al pubblico sia che esso sia fisicamente presente, sia che si lo visualizzi remotamente: particolare importanza viene data allo studio della latenza[4] tra l'immissione del

[4]Il tempo di latenza è una misura del tempo di risposta di un sistema; è l'intervallo di tempo che

contenuto nella rete e l'effettiva reazione dell'apparato di visualizzazione, cioè la visualizzazione stessa.

Un ruolo centrale è ricoperto dallo streaming, che viene usato per trasportare il materiale multimediale agli utenti collegati dalla rete. È molto importante che l'utente remoto non percepisca un ritardo distinguibile tra l'evento effettivo e la conseguente ricezione. A questo scopo infatti, la ricerca si è curata potenziare le tecniche di compressione di un flusso video, che possono essere usate allo scopo ridurre il peso della sequenza di immagini mantenendo però uno standard di qualità. Nel tempo si sono evoluti particolari versioni di codificatori (come H.264, VP8 ed i più recenti H.265 e VP9) e formati contenitori (come MPEGTS, FLV, WebM o ASF) dedicati solo allo streaming.

La ricerca ha fornito anche diversi metodi per produrre una sessione di streaming riducendo al minimo il carico sull'elaboratore e sulla rete; la maggior parte di essi si basa su protocolli di rete già pienamente e universalmente accettati come TCP o UDP[5]. Alcune società molto attive nell'ambiente, come Adobe e Apple, hanno studiato delle soluzioni per lo più basate su UDP e orientate esclusivamente allo streaming in tempo reale: RTSP (*R*eal-*t*ime *S*treaming *P*rotocol) per la Adobe e HLS (*H*TTP *L*ive *S*treaming) per la Apple.

Recentemente hanno visto la luce perfino una serie di tecnologie che si basano sull'adattamento dinamico del *bitrate* del flusso video inviato in streaming, con lo scopo di risparmiare quantità di banda oppure ridurre automaticamente la quantità di dati inviati in caso di congestione della rete. Una soluzione su tutte è quella proposta da MPEG-DASH, ideata dal Moving Experts Picture Group; essa si prefissa lo scopo di riunire le già citate tecnologie esistenti e uniformarle sotto uno stesso standard.

Anche i protocolli di rete hanno subito un'evoluzione cooperante col miglioramento delle prestazioni dello streaming, soprattutto in tempo reale. Se prima si doveva

---

intercorre fra il momento in cui viene immesso un input al sistema e il momento in cui è disponibile il suo output.

[5]Per lo streaming in *realtime* di solito si preferisce usare UDP piuttosto che TCP, in quanto, al contrario di quest'ultimo, evita tutto il meccanismo di ritrasmissione dei dati in casi di errore di consegna.

obbligatoriamente instanziare uno streaming di tipo unicast, adesso è possibile sfruttare il miglioramento portato dal multicast. Quindi se nel passato ogni client che si fosse connesso ad un provider di contenuti avrebbe dovuto creare, anche a parità di contenuto inviato, il suo personale canale di connessione, ora è possibile che i client sfruttino un canale condiviso col server, minimizzando l'*overhead* creato.

Un altro problema, riguardante sia il digital signage, sia lo streaming in tempo reale, che si è venuto a creare con l'uso di più schermi per la creazione di una superficie più ampia di visualizzazione è quello dell'uniformazione delle caratteristiche delle immagini visualizzate sui monitor accostati. Ciò è fondamentale sotto il punto di vista della qualità visiva se lo scopo del videowall è quello di diventare un "cartellone digitale", ma non è di minore importanza neppure nel mondo delle applicazioni di controllo, in quanto esistono applicazioni che si basano esclusivamente sull'uniformità tra sorgente in input e flusso in output. A questo scopo è stata introdotta la correzione cromatica del flusso generato (i dettagli su questo processo si possono trovare al capitolo 5).

1.2 Il progetto

Il progetto, di cui si tratterà in questo elaborato, si colloca nell'ambito del *live streaming*, mutuando però alcuni concetti di base del digital signage. Esso mira a realizzare un sistema di cattura di immagini digitali e conseguente streaming in modalità *multicast*, per mezzo del protocollo UDP, verso un *player* a finestra singola oppure multifinestra in configurazione videowall.

La soluzione creata è un'applicazione innovativa scritta completamente ex novo e fortemente incentrata sulla parte software. È divisa in due parti principali. Una prima componente fungerà da server e si occuperà di catturare le immagini, comprimerle ed inviarle sulla rete in streaming; la seconda componente sarà invece il client distribuito, nel quale ci saranno un insieme di elaboratori, collegati ad altrettanti monitor, che si faranno carico di ricevere lo streaming, ritagliare la parte del flusso di video di propria competenza e visualizzarla.

Si fa largo uso della rete come veicolo di comunicazione tra server e client, anzichè usare nella maniera più tradizionale dei collegamenti orientati al video e un centro di controllo situato nel server. Per questa ragione la tipologia del videowall che si andrà a realizzare ricadrà nella definizione di *network videowall*.

La cattura dei contenuti verrà effettuata tramite un'apposita scheda di *capture*, predisposta solo per immagini digitali. Si è scelto di evitare le sorgenti analogiche, poiché al giorno d'oggi nella pratica sono ormai poco usate e introducono una serie di complicazioni in più nella loro manipolazione.

Come si potrà notare si è optato per uno streaming in modalità IP multicast. Gli indirizzi IP di tipo multicast sono una classe di indirizzi IP che referenziano solo un gruppo degli occupanti la rete che è un sottoinsieme del numero totale. È differente dalla classe di indirizzi broadcast[6] poiché permette ai dispositivi connessi alla rete di scegliere di "iscriversi" come destinatari dello stream.

Si è deciso di non inizializzare lo streaming nella maniera tradizionale sfruttando protocolli unicast, ma di usarne uno multicast per un motivo fondamentale: risparmiare al massimo l'utilizzo della rete. Con unicast ogni ramo del client distribuito per richiedere il segnale avrebbe avuto bisogno di dialogare col server e crearsi il proprio canale di comunicazione con quest'ultimo, lasciando il server con tanti canali aperti quanto il numero di client. Con multicast invece è il server ad aprire un canale verso un indirizzo specifico: quando un client si collega utilizza il canale già creato condividendolo con gli altri client e in questo modo il server mantiene aperto un solo canale di comunicazione.

Un'altra scelta concettuale che riguarda da vicino quella del multicast è stata quella dell'utilizzo di pacchetti incapsulati in datagrammi riconducibili al protocollo UDP per lo streaming (com'è noto, a differenza di TCP, nel protocollo UDP non è garantita la consegna del pacchetto di dati alla destinazione). Questo sì elimina l'*overhead* imposto dall'*handshaking* e dall'eventuale ritrasmissione dei dati, ma non

[6]Gli indirizzi IP broadcast permettono di inviare a tutti gli utilizzatori della rete un messaggio.

garantisce la consegna del contenuto al destinatario (ma trattandosi di un'applicazione video, la perdita di alcuni frame non inficia le performance generali).

UDP è stato inoltre scelto per la semplicità d'implementazione del protocollo e per la compatibilità. Ormai, un buon numero di riproduttori multimediali *general purpose* riescono a ricevere e decodificare uno stream incapsulato in datagrammi di questo tipo.

Infine, l'applicazione, come già detto, contiene due parti: un server, che sarà realizzato completamente in maniera software il più possibile indipendente dalla piattaforma di esecuzione, ed un client, che avrà dei requisiti hardware più stringenti da rispettare per il suo corretto funzionamento. La componente server, che funziona anche da centro di controllo, sarà disponibile inizialmente per il sistema operativo libero GNU/Linux, ma verrà predisposta per un facile adattamento ai sistemi operativi chiusi più in uso nel settore, nella fattispecie i sistemi Microsoft Windows.

Tutto il progetto rispetterà i requisiti di conformità al software libero (come descritto alla sezione 1.3) e di portabilità (descritta nella sezione 1.4. Il codice dell'intero progetto è completamente disponibile sotto una licenza per software libero agli indirizzi web `https://github.com/datasoftsrl/v4lcapture` e `https://github.com/datasoftsrl/pimcplayer`.

1.3 Software libero

Come già accennato precedentemente, uno dei capisaldi nella realizzazione di questo ecosistema multimediale è stato la condivisione del codice scritto sotto una licenza cosiddetta libera e a sorgente aperta, come la GPL [7]. Tale regime legale impone ai licenziatari il permesso di visionare e modificare il codice sorgente, di redistribuirlo sia nella forma originale, che con modifiche, ma lascia la libertà di sfruttarlo per scopi commerciali, senza limitazione alcuna.

[7] La licenza GPL, o *G*eneral *P*ublic *L*icense, è il prodotto della Free Software Foundation che ha coniato la definizione di *free software*. Ne esistono tre versioni: la prima è caduta in disuso, la seconda e la terza sono le licenze libere più usate al mondo.

Questa particolare forma di licenza non solo si occupa di definire le azioni che il licenziatario può o non può intraprendere nei confronti del software, ma regola anche le condizioni alle quali quest'ultimo è tenuto a ridistribuirlo. Dopo un'attenta analisi, è stata scelta la seconda versione della licenza, poiché impone che, nel caso di restrizioni di qualsivoglia tipo sulla redistribuzione del software nei termini previsti dalla licenza, esso non possa essere ridistribuito in alcun modo.

Tutte le soluzioni simili al progetto posto in argomento di questo scritto apparse finora sul mercato sono sistemi chiusi e con licenza proprietaria, e molto spesso basate esclusivamente su sistemi operativi allo stesso modo a sorgente chiusa (l'esempio più comune è Microsoft Windows). Un licenza aperta invece garantirebbe l'accesso al codice ad altri sviluppatori, anche nel caso essi fossero indipendenti dal progetto originale. Questo fatto promuove la scrittura di un'applicazione *peer reviewed*, vantaggio che visto da un punto di vista progettistico, può garantire la creazione di un software meno soggetto a generare errori, mentre visto da quello ingegneristico, fornisce una struttura meglio organizzata del codice, in quanto risultato di un brainstorming collettivo.

Inserendosi nella posizione del distributore dei prodotti di digital signage o dell'utente finale che andrà ad utilizzarli, ci si accorge che fin troppo spesso le soluzioni a sorgente chiusa mancano di una determinata funzionalità o rendono difficile, in alcuni casi impossibile, l'aggiunta di particolari moduli, la personalizzazione dell'applicazione stessa secondo speciali esigenze oppure la correzione di piccoli errori in autonomia.

Il progetto si insinua proprio in questa crepa e cerca di coprire tali esigenze sfruttando il paradigma dell'*open source*, quindi basandosi inizialmente su sistemi operativi notoriamente aperti (come quelli costruiti sul kernel Linux). La disponibilità del codice sorgente si rivela perciò fondamentale nel momento in cui si localizza la necessità di correggere o aggiornare il funzionamento del sistema creato.

Allo stesso modo, le condizioni della licenza riguardanti la redistribuzione sono un'arma potente nell'ambito aziendale: ogni modifica al codice, effettuata da una qualsiasi delle parti incluse nel processo di produzione e consumazione del software, dev'essere rilasciata, facendo così in modo che possa essere, alla fine dell'intero processo di convalida, fusa nella base di codice originaria, costituendo di fatto un vantaggio

industriale da non sottovalutare.

Il progetto non solo segue lui stesso i paradigmi dell'*open source*, ma negli ambiti dove ve n'era necessità si è cercato di includere e operare con librerie che fossero disponibili con una licenza libera e a codice aperto. Inoltre tutti i software di cui ci si è serviti durante il corso della realizzazione dell'applicazione, come ad esempio il programma usato per effettuare la correzione del colore pilotando il colorimetro, sono di natura aperta e libera.

1.4 Portabilità e indipendenza dall'hardware

Le applicazioni comparse sul mercato finora, oltre che ad essere a sorgente chiusa e strettamente integrate ad un sistema operativo chiuso e inestensibile, hanno anche il limite di essere fortemente sviluppate per sfruttare le caratteristiche di un tipo o un modello di hardware specifico (o nella maggior parte dei casi sono completamente basate su un'hardware specifico).

Un obiettivo principale del progetto è stato quello di creare un sistema per cui la disponibilità di un particolare hardware non avrebbe dovuto influire sulla possibilità di replicare il sistema multimediale su un altro setup hardware → sistema operativo → periferiche. Per questa ragione, si è voluto applicare il concetto di portabilità, sviluppando un'applicazione fortemente incentrata sull'operato della parte software, ma debolmente legata all'architettura dell'elaboratore che la esegue e alle varie componenti che svolgono i compiti più specifici, quali schede di acquisizione o periferiche di visualizzazione. Per raggiungere questo scopo, all'interno del software sono presenti diversi livelli di astrazione che abilitano la conformazione, ad esempio, all'infrastruttura di cattura di immagini standard del sistema operativo in uso.

L'applicazione è stata scientemente divisa in sezioni e stratificata in maniera tale che l'insieme delle diverse parti formi un sistema complesso. Così facendo ogni parte è rimpiazzabile da una nuova a patto che quest'ultima compia le stesse azioni di quella che va a sostituire. Questo l'intero progetto ad essere portato ad altre piattaforme; inizialmente esso sarà reso disponibile esclusivamente per sistemi *open* (tra i quali quello più utilizzato in ambito aziendale è GNU/Linux) e successivamente potrà

essere sviluppato anche per sistemi chiusi ma assai più diffusi nell'ambiente, come Microsoft Windows, al fine di raggiungere pari funzionalità con le soluzioni concorrenti.

Si è deciso di utilizzare componenti hardware il più possibile a basso costo e soprattutto che rientrassero, ove possibile, nella descrizione di *open hardware*. Pur non esistendo una definizione ufficiale di *open hardware*, un progetto che si qualifica come tale, solitamente rilascia, sotto un regime legale che permette la libera consultazione e il libero ampliamento senza vincoli di licenza e di utilizzo, gli schemi elettrici, la lista dei materiali costruttivi e gli schemi per la realizzazione di eventuali circuiti aggiuntivi.

La possibilità di disporre di simile materiale, permette l'utilizzo di componenti generiche, facilmente acquistabili sul mercato, anche da società produttrici terze, senza rivolgersi ad aziende selezionate. Diventa evidente l'agevolazione economica e costruttiva derivante dal risparmio in fase di acquisto dei materiali, che, in ultima analisi, si trasforma in un vantaggio industriale non trascurabile.

Inoltre, rilasciando succitati documenti, si è voluta lasciare la libertà a chiunque, interno o esterno al progetto, di contribuire, migliorando il progetto stesso o la sua documentazione. Non si è voluta comunque eliminare la prospettiva futura di aggiornamenti e miglioramenti non solo dal lato software, ma anche da quello hardware, compresa la costruzione ad hoc di componenti specifiche.

Capitolo 2

Stato dell'arte

L'universo del digital signage e del live streaming, descritto al capitolo 1, negli ultimi anni è considerevolmente progredito. Il motivo di questa affermazione è sicuramente da ricercarsi nella miniaturizzazione dei sistemi di calcolo e più in generale di tutta la componentistica del settore (ad esempio i chip per la codifica e decodifica video a livello hardware). A questo avvento di transistor di dimensioni minime e circuiti integrati non si è però corrisposta una riduzione della potenza di calcolo, bensì un aumento esponenziale. Anche lo studio di nuovi tipi di connessioni di rete sempre più veloci e di formati con possibilità di codifica e decodifica sempre più efficienti ha avuto il suo ruolo fondamentale.

Intorno a questa nuova opportunità di business, si sono sviluppate, nel corso del tempo, diverse realtà che ormai operano a livello internazionale, in quanto sono considerate le massime esponenti della produzione della tecnologia usata in questo ambiente. Aziende quali la Matrox Graphics, la Datapath Limited, la Extron Electronics e la Jupiter Systems producono la maggior parte delle componenti usate a livello industriale per la produzione di sistemi multischermo, di cartellonistica digitale, di streaming in tempo reale e, più in generale, di *video processing*.

Queste aziende si spartiscono un volume di affari che sta crescendo sempre di più e ogni anno aumenta di qualche miliardo di dollari; secondo una relazione stilata nel 2014 [6], questo business arriverà al 2020 a valere circa 20 miliardi di dollari (come si può osservare in figura 4). Grazie a questa movimentazione di denaro, la ricerca ha potuto progredire nello studio di nuove tecnologie per rendere ancora più efficienti

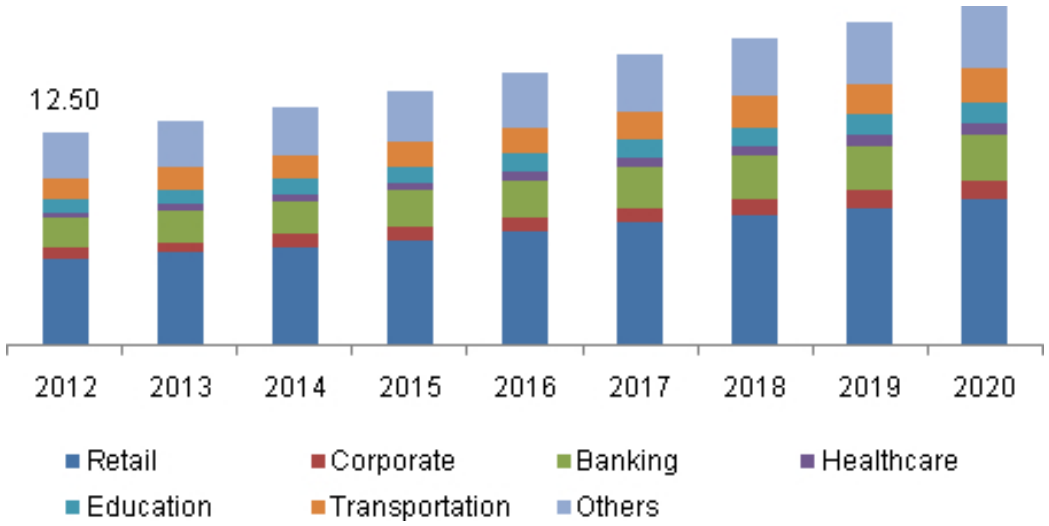

Figura 4: Evoluzione prevista del volume di affari del digital signage (da [6]).

questi sistemi.

I prodotti che si possono trovare spaziano lungo diversi fronti: esistono soluzioni integralmente dedicate ai videowall e altre che invece possono essere usate in maniera *standalone*, cioè con tutte le funzionalità autocontenute nello stesso modulo.

A questo scopo è utile compiere un excursus sulla strumentazione che viene normalmente usata durante la costruzione di un sistema con un videowall, comprensiva di eventuali estensioni accessorie.

Per prima cosa, occorre definire cos'è una sorgente: una sorgente (in questo caso video) è un flusso di dati generato da una qualsiasi periferica (che è in grado di farlo) e diventa tale nel momento in cui viene inviata su un canale di comunicazione. Nel caso pratico, una sorgente può essere un computer, una videocamera, un satellite o persino una televisione. Il canale di comunicazione può essere di diverso tipo, esempi sono un cavo per trasporto di video (come cavi VGA, DVI, HDMI, DisplayPort, ecc.) oppure cavi per trasporto di dati (come cavi ethernet, che si usano per creare reti LAN o Internet).

Dall'altro capo del canale di comunicazione appena descritto, deve esserci uno strumento che sia in grado di interpretare il flusso di dati che gli viene inviato. In questo caso, il tipo di hardware che realmente viene impiegato nella pratica dipende dal produttore che si sta considerando: per esempio aziende come la Datapath

propongono sia soluzioni totalmente *standalone* basate su hardware specializzato, sia soluzioni che fanno uso, invece, di architettura PC abbinata ad una scheda di acquisizione appositamente studiata; altre aziende, la Jupiter ne è un esempio, offrono solo soluzioni integrate sotto forma di moduli di input che possono ospitare in ingresso anche più di un dispositivo.

Giunti a questa meta, serve qualcosa che faccia da collettore per tutti gli input e fornisca un'unica immagine divisa atomicamente da trasmettere a ogni monitor che compone il videowall. La realizzazione pratica si divide in due possibilità, diverse a seconda del produttore che viene considerato.

Se si tiene a modello la Datapath o la Matrox, queste offrono dei prodotti basati su architettura PC. Tali prodotti oltre che montare delle schede di *capture* possono occuparsi anche del *processing* dell'immagine e ospitare delle schede di uscita apposite per pilotare ogni schermo del videowall. Questo approccio ha due punti critici: innanzi tutto serve un software che si occupi di decidere, per mezzo dell'input dell'utente, qual è la configurazione dei monitor e proceda a commutare i pacchetti del flusso video dalle schede di cattura alle schede di uscita; in secondo luogo, è necessario avere delle schede di cattura che comunichino direttamente con le schede di uscita, poiché se il flusso di dati transita all'interno della CPU, il *throughput* del sistema viene drasticamente ridotto. Se si sceglie l'approccio appena descritto, a questo punto il videowall è già funzionante, perciò non è necessario collegare altri elaboratori al sistema. Addirittura grazie all'uso di un elaboratore provvisto di sistema operativo Windows, con questo sistema è possibile recuperare i dati in ingresso non solo dalla scheda di acquisizione, ma anche da sorgenti remote (ad esempio uno streaming che transita su Internet) utilizzando software standard (ad esempio un *player* come VLC).

Se, invece, vengono considerati i prodotti della Jupiter o della Extron, si trova un'ampia gamma di soluzioni basate su hardware specializzato. Per poter raggiungere un sistema funzionante, bisogna collegare diversi moduli hardware in cascata, ognuno dei quali assolve il proprio compito. Per esempio, una volta catturati i diversi flussi per mezzo dei moduli di input, il collettore in questo caso è un particolare switch di rete, che si occupa di ritagliare alla giusta sezione e commutare i dati video verso i monitor del videowall. In questo modello, i monitor non sono passivi, bensì attivi, cioè necessitano di un elaboratore che acquisisca e riproduca il flusso video in arrivo dalla rete interna; siffatti elaboratori sono costituiti da moduli di output, che

hanno esattamente questo scopo. Per controllare il sistema c'è comunque bisogno di un calcolatore provvisto di sistema operativo con annesso un software apposito per il controllo dei flussi (che è compatibile esclusivamente con Microsoft Windows). Se per esempio si volesse catturare dei flussi provenienti da sorgenti sulla rete Internet, bisognerebbe aggiungere un altro modulo hardware la cui mansione sarebbe esclusivamente quella appena descritta.

Un altro caso che è molto frequente è quello della realizzazione di un videowall con una superficie di visualizzazione molto estesa. Col primo modello descritto, quello che riguarda i prodotti Datapath e Matrox, si utilizzano schede di visualizzazione che hanno al massimo quattro porte di uscita. Questo limita il rendimento di ogni singola scheda; se volessimo aggiungere più monitor sarebbe necessario procurarsi un'altra scheda identica. Col secondo tipo, invece, serve un modulo di output per ogni monitor: per aggiungere n monitor alla configurazione sarebbe sufficiente aggiungere n altri moduli di output.

Com'è possibile dedurre, l'insieme di componenti utili a realizzare un sistema funzionante con tutte le funzionalità desiderate è molto vasto. Le componenti hardware indicate durante questa panoramica sono altamente specializzate: si tratta di elaboratori dalla potenza di calcolo molto grande. La potenza di calcolo e la specializzazione sono direttamente proporzionali al prezzo, infatti un sistema provvisto di tutti i moduli descritti può arrivare a costare diverse decine di migliaia di dollari.

Nonostante il costo di produzione così grande, c'è reticenza da parte dei produttori a trasferire il parco applicativo a livello software. Questo ha una spiegazione chiara; l'architettura computer più usata a livello mondiale è quella PC, anche detta x86. Tale architettura è stata pensata per un elaboratore *general purpose* o addirittura un *personal computer*. Anche se la potenza di calcolo negli ultimi anni sta aumentando velocemente, non sarà mai al pari di un circuito hardware specializzato. Inoltre, l'architettura x86 solo negli ultimi tempi sta iniziando ad esporre delle interfacce di input/output abbastanza perfomanti da supportare un flusso video (ne è un esempio la PCI Express, su cui si basa la quasi totalità dei sistemi di *video processing* costruiti su modello PC).

Un altro problema che viene riscontrato spesso è quello della compatibilità software di gestione con diversi sistemi operativi. Attualmente i software di controllo

forniti dai produttori di hardware sono esclusivi per Microsoft Windows, che è un sistema a licenza proprietaria, poiché è il più diffuso universalmente. Qualora venissero prodotti dei software che imitano le funzioni delle controparti hardware, anche questi verrebbero scritti esclusivamente per Windows. Ciò limiterebbe la disponibilità di soluzioni *cross-platform* (ad esempio compatibili con GNU/Linux, che ormai sta diventando una realtà nel mondo aziendiale), nonchè la possibilità di estensione, modifica e correzione del software stesso, poiché quest'ultimo sarebbe, nella maggioranza dei casi, concesso tramite una licenza altrettanto limitante e proprietaria.

Ha senso, arrivati a questo punto, procedere ad elencare e descrivere una serie di soluzioni che possono, per funzionalità, essere accostate al progetto trattato in questo scritto.

I prodotti che maggiormente corrispondono, considerando esclusivamente i top di gamma disponibili al momento, ad una descrizione delle funzionalità richieste sono tre. Due di essi sono fabbricati dalla Matrox Graphics, il MX02 LE MAX e il Monarch HDX, mentre l'ultimo è commercializzato dalla Extron Electronics, il Pure 3 VNE 250. Nella tabella 1 è possibile trovare un confronto di quelle che sono le funzionalità principali. Le voci della tabella rappresentano un insieme di caratteristiche fondamentali che ci si aspetta di trovare in una generica soluzione di questo genere.

La prima cosa che è possibile evidenziare è che tutte quante sono implementazioni hardware; come già sostenuto, le aziende preferiscono creare hardware specifico che si integri alla perfezione piuttosto che utilizzare un'interfaccia comune. Questo comporta tutta la serie di problemi già descritti e limita di molto l'interoperabilità tra i sistemi (anche di diverse aziende), circoscrivendo allo stesso tempo anche molte possibilità di estensione (ad esempio aggiungere un nuovo formato supportato).

A livello di interfaccia di collegamento in input, tutti e tre i prodotti, come la maggioranza delle soluzioni di *video processing* presenti sul mercato dagli ultimi anni, presentano una porta HDMI di tipo A (e tutti e tre i device hanno il supporto fino alla specifica 1.4b[1]). Ciò in cui i prodotti si differenziano sono risoluzione, *framerate*

[1]Le versioni di HDMI indicano le diverse revisioni della specifica: ogni revisione contiene miglioramenti quali maggiore larghezza di banca o altre *capabilities*. La versione è l'ultima per quanto

Tipo		Hardware	Hardware	Hardware
Input				
	Tipo	1 × HDMI tipo A	2 × HDMI tipo A	1 × HDMI/DVI
	HDCP	✗	✗	✓
	Risoluzione	1920 × 1080 p	1920 × 1080 p	3840 × 2160 p
	FpS	fino a 60 fps	fino a 60 fps	fino a 60 fps
Encoding				
	Codec	H.264	H.264	Pure 3
	Risoluzione	Da 64 × 64 a 1920 × 1080 p	Da 128 × 128 a 1920 × 1080 p	VESA e SMPTE fino a 1080p
	Bitrate	Fino a 50 Mb/s	Fino a 30 Mb/s	Fino a 270 Mb/s
	FpS	Fino a 30 fps	Fino a 60 fps	Fino a 60 fps
	Profili	Baseline, Main, High	Baseline, Main, High	
	Controlli	Livello, GOP, CRF, CBR, filtro antiblocchi	Livello, GOP, CRF, CBR, filtro antiblocchi	Bitrate
Streaming		RTP, RTMP	RTP, RTMP	RTP, RTCP, TCP, UDP multicast
Audio		✓	✓	✓
Preview		✗	✓	✓
Compatibilità		Utility solo per Windows	Web UI, utility solo per Windows e Mac	Web UI
		Matrox MXO2 LE MAX	**Matrox Monarch HDX**	**Extron Pure 3 VNE 250**

Tabella 1: Caratteristiche dei prodotti per streaming in commercio.

in entrata e supporto all'HDCP[2].

MXO2 LE MAX e Monarch HDX supportano entrambi una risoluzione massima di 1920 × 1080 pixel con codifica progressiva, ma il primo riesce a catturare solo fino a 30 fps, mentre il secondo fino a 60 fps. Ciò ha una rilevanza particolare poiché molti dei flussi video che vengono tramessi sono proprio a 60 fps. Differente è il

riguarda l'alta definizione (esiste la 2.0, che però è dedicata all'Ultra HD).

[2]HDCP (*H*igh *B*andwidth *C*ontent *P*rotection) è il sistema di protezione dei diritti digitali multimediali sviluppato da Intel per tutelare le trasmissioni via interfaccia HDMI di materiale protetto da copyright.

supporto di VNE 250 che arriva fino a 3840×2160 progressivi a 60 fps. L'aumento della risoluzione in input è la risposta verso un futuro in cui i flussi video diventano sempre più ampi. Per quanto riguarda HDCP, essendo un'implementazione di natura proprietaria, il supporto non è garantito. Infatti le prime due soluzioni non sono in grado di decodificare i flussi che usano questo protocollo, mentre il terzo sì.

Se consideriamo invece le possibilità di encoding i parametri principali da controllare sono il codec utilizzato, la risoluzione e il *framerate* in uscita, il *bitrate* massimo raggiungibile, il tipo di profilo video selezionabile e le regolazioni di qualità supportate. I due prodotti della Matrox hanno una dotazione piuttosto standard: il codec utilizzato è H.264, con risoluzione massima di 1920×1080 (viene dinamicamente assunta la dimensione di input) e un *framerate* di 30 fps per MXO2 LE MAX e di 60 fps per Monarch HDX. Il *bitrate* varia da range ampi dai 100 Kb/s ai 50 Mb/s o 30 Mb/s. I tipi di codifiche prodotte possono spaziare dai profili a qualità più bassa come Baseline a quelli a qualità maggiore come High (per una panoramica sulle impostazioni di H.264 riferirsi alla sezione 3.3). Per quanto concerne i settaggi di qualità, sono i medesimi per entrambe le soluzioni: GOP[3], CRF (Constant Rate Factor), CBR (Constant Bit Rate) e un *deblocking filter*.

VNE 250 invece non implementa H.264 ma un codec proprietario chiamato Pure 3. L'implementazione di questo particolare formato proprietario limita ancora di più l'interoperabilità tra le diverse soluzioni, spingendo l'utente finale a dover acquistare tutti i prodotti dallo stesso venditore. L'unica giustificazione per l'adozione di una codifica inusuale è la sua efficienza. In questa sede non si sono potute verificare le effettive performance del codec, ma il *bitrate* raggiungibile è molto alto, dell'ordine di 270 Mb/s con una risoluzione standard di 1920×1080 e un *framerate* di 60 fps. Per quanto riguarda le impostazioni di qualità, Pure 3 ha solo un profilo dedicato allo streaming, che è quello standard e l'unica regolazione possibile è quella del *bitrate* in uscita.

Tutte e tre le opzioni prevedono la possibilità di catturare e portare in streaming un flusso audio, abbinato a quello video, comprimendolo (con codifica AAC, la più usata in abbinamento ad H.264). Le tipologie di streaming supportate sono direzionate

[3]Il GOP (*G*roup *o*f *P*ictures) nell'ambito delle specifiche H.264 è la distanza tra due frame di tipo I, quindi non predittivi.

verso protocolli Adobe: c'è la possibilità di avviare streaming in RTP e RTMP. Solo il VNE 250 permette la produzione di flussi in streaming incapsulati in datagrammi TCP e UDP, con l'opzione del multicast.

Tipo		Software
Input		
	Tipo	Dipendente dalla scheda di acquisizione in uso
	HDCP	Dipendente dalla scheda di acquisizione in uso
	Risoluzione	Dipendente dalla scheda di acquisizione in uso
	FpS	Dipendente dalla scheda di acquisizione in uso
Encoding		
	Codec	H.264
	Risoluzione	Pari a quella in input
	Bitrate	Fino a 51 (scala CRF)
	FpS	Fino a 30 *fps*
	Profili	Baseline, Main, High, Extended
	Controlli	Risoluzione, Multiplexer, Preset, Tune, CRF
Streaming		Tutti quelli supportati dalla libreria usata
Audio		✗
Preview		✓ a 480p massimo
Compatibilità		GNU/Linux
		V4LCapture

Tabella 2: Caratteristiche provvisorie del progetto realizzato.

Per pilotare la ricezione dei flussi, le regolazioni di qualità e la gestione dell'hardware stesso (come l'aggiornamento del firmware interno) è necessario un software a parte che ogni produttore realizza. Come è d'uso nei prodotti più recenti, tali programmi abbinati sono multipiattaforma, utilizzabili tramite un'interfaccia costruita sulle tecnologie Web (e questo è valido anche per gli apparati descritti). L'unica eccezione sono i software adibiti ad esempio alla gestione del firmware, che spesso vengono

forniti solo per Windows (come nel caso della MXO2 LE MAX) e per Mac OS X (come nel caso della Monarch HDX).

I programmi di configurazione via Web, definiti WebUI, hanno diverse funzionalità a seconda dai produttori; una funzione di grande utilità è quella di preview del contenuto che successivamente si immetterà nella rete in streaming ed è difficile fornirla tramite tali programmi di configurazione. Infatti, solo Monarch HDX e VNE 250 supportano questo tipo di vista, mentre per MXO2 LE MAX non è disponibile.

È necessario anche considerare che le componenti presentate, come accennato nei paragrafi precedenti, non formano un sistema a sé stante. Per costruire un videowall completo servono ancora almeno i nodi di output verso gli schermi oppure un elaboratore x86 che faccia da tramite con i monitor per mezzo delle schede di uscita. Inoltre, per captare e decodificare dei flussi video provenienti dalla rete (LAN o Internet) serve aggiungere un nodo di input specializzato (oppure aggiungere del software all'elaboratore aggiunto in precedenza).

Bisogna tenere conto, però, che questi modelli di componentistica non sono comuni apparati di calcolo, ma vengono costruiti per assolvere un solo compito specifico. Come tali, il loro prezzo è molto alto: ogni modulo aggiunto al sistema vale migliaia di dollari da solo. Come già anticipato, per un sistema videowall di alta qualità completo in ogni sua parte se ne può arrivare a sborsare svariate decine di migliaia.

Il sistema prodotto come oggetto di questo scritto mira ad essere l'esatto opposto di quanto esposto finora.

Nella tabella 2 vengono elencate le funzionalità caratterizzanti di tutto questo progetto. È essenziale tenere presente che il software creato è ancora in via di perfezionamento; nel tempo di sviluppo a disposizione sono state introdotte le funzionalità principali per il corretto funzionamento dello stesso, lasciando comunque lo spazio a futuri cambiamenti ed estensioni.

Al contrario di tutte le soluzioni viste finora, il progetto è principalmente costituito da elementi software. Come verrà spiegato meglio più avanti (al capitolo **??**), è stato ideato tenendo come base l'architettura PC, ma ciò non influisce sulla possibilità di funzionamento anche su altre architetture, grazie alle scelte implementative compiute.

L'architettura PC garantisce, come già spiegato più volte, una libertà di estensione e di controllo impossibile nell'ambito hardware, soprattutto se abbinata alla licenza totalmente aperta e alla disponibilità del completo codice sorgente dell'applicazione prodotta.

L'acquisizione dell'input è basata su una scheda collegata all'elaboratore su cui viene eseguito il software. Non è però vincolante la scelta di tale scheda, poiché l'insieme di astrazioni di cui è stato fatto uso ne permette l'utilizzo di svariate. In questa situazione, le caratteristiche che lo stream video in input deve avere sono decise da quelle della scheda scelta, lasciando completa libertà all'utente finale di valutarne un modello congruo alle sue esigenze. A titolo di esempio, una delle schede a disposizione durante lo sviluppo prevedeva un input su interfaccia DVI, con risoluzione non superiore a 1920×1080 e un *framerate* al massimo di 60 fps. Inoltre supportava la crittografia HDCP.

Sulla codifica, invece, si è rimasti più fedeli alle soluzioni hardware già viste, infatti allo stesso modo è stato scelto di usare H.264 (sempre lasciando aperta l'implementazione di altri formati), poiché è lo stato dell'arte dei formati sotto il punto di vista del rendimento e dell'affidabilità. Il flusso uscente dal *decoder* è comunque conforme a quello in input: la risoluzione rimane invariata, ma il *framerate* è al massimo di 30 fps. Per quanto concerne la qualità, si è deciso di non includere l'opzione per mantenere un *bitrate* costante, ma è stata lasciata solo la funzione CRF, permettendo quindi di scegliere una soglia di qualità costante. Tra i profili H.264 supportati figurano tutti quelli visti nei prodotti hardware più un altro chiamato Extended, simile ad High, ma ottimizzato per lo streaming. Come regolazioni, allo stato attuale sono disponibili solamente la scelta del *container*, del *preset* e del *tune*. Sia la qualità (sotto forma di valori CRF) che la risoluzione possono essere variate a piacere: per quanto riguarda quest'ultima, se viene superato il valore in input, l'output sarà necessariamente ingrandito per coprire l'intera superficie scelta.

Come opzioni di streaming ne sono disponibili molteplici, ma il progetto si concentra sul semplice UDP con l'uso di indirizzi multicast. La libreria sottostante prevede, però, il supporto a molti altri protocolli: RTP, RTMP, RTSP, TCP/IP e anche diversi tipi di comunicazione, quali unicast (punto a punto) e broadcast.

Lo streaming di flussi audio non è ancora supportato in questa versione, ma è stato lasciata la possibilità di inserire questa funzionalità senza alterare lo stato generale

del programma. È disponibile, invece, una preview di dimensioni notevoli rispetto agli standard di altri produttori (di solito è massimo di 320×240), per avere un'anteprima significativa del flusso che sarà trasmesso in rete.

I sistemi operativi supportati sono tutti quelli compatibili con il kernel Linux e il sottosistema GNU. Questa limitazione non è molto flessibile, in quanto la disponibilità del codice sorgente e le scelte implementative compiute in via di sviluppo rendono lineare l'operazione di *porting* ad altre piattaforme. Questo rende l'applicazione una dei rari esempi di applicazioni per *network videowall* per il sistema GNU/Linux.

Diversamente dalle soluzioni hardware viste, il progetto mira non solo a fornire una parte server capace di comprimere i contenuti in input e trasmetterli in rete; viene fornita anche una parte client, che è distribuita (maggiori dettagli si possono trovare nella sezione **??**) al fine della creazione di un videowall. Tale client distribuito è costituito da microcomputer che assolvono lo stesso compito dei nodi di output visti. Le componenti di calcolo usate, al contrario di questi ultimi, sono assai poco costose e molto comuni, perciò è possibile trovarle facilmente sul mercato.

La vera forza di questo sistema, oltre alla flessibilità, alle possibilità di personalizzazione e all'apertura ai cambiamenti, è la scalabilità del costo. Il costo è direttamente proporzionale alla qualità degli elementi costruttivi scelti: essendo in grado di scegliere le varie componenti del sistema, quali schede di acquisizione, elaboratori, switch di rete e microcomputer, è possibile stabilire un *target* economico a cui attenersi, non per forza molto basso, nè a maggior ragione eccessivamente alto. Inoltre, la componentistica usata è comune e per questo molto diffusa, al punto che la disponibilità sul mercato è ampia, anche con diversi modelli e fasce di prezzo.

Capitolo 3

Progettazione e sviluppo

3.1 Contesto di realizzazione

Lo sviluppo dell'applicazione descritta in questo elaborato è partito come studio di fattibilità di un progetto più ampio, all'interno dell'azienda OffiCINE Srl, con la collaborazione della DataSoft Srl. Il progetto verte sulla realizzazione di un sistema di *capture* di immagini digitali con possibilità di *live streaming* verso un *player* distribuito di tipo videowall con l'aggiunta di un editor di video non lineare che permetta l'introduzione di effetti di distorsione geometrica e cromatica *on-the-fly*, cioè agisca in tempo reale sul flusso video creato, senza bisogno di utilizzare la tecnica della post-produzione.

OffiCINE nasce nel 2003 e si occupa principalmente di commercio di sistemi di videoproiezione e relativo supporto tecnico; questi sitemi comprendono ma non si limitano ai videoproiettori per diversi usi, alle sale cinema e ai prodotti di digital signage. I casi d'uso variano da applicazioni di tipo *rental*[1] ad installazioni fisse. Altri prodotti proposti sono i sistemi di videoproiezioni 3D in modalità stereoscopica oppure in modalità immersiva, con ambiti di impiego che spaziano da quello automobilistico a quello aeronautico.

[1]Le installazioni di tipo *rental* sono quelle mobili che vengono noleggiate per l'organizzazione di eventi *live*.

DataSoft nasce anch'essa come piccola realtà nel 2003 producendo soluzioni hardware e software per sale di controllo di varia natura. Con gli anni si è aperta alla creazione e al supporto di prodotti per l'acquisizione e il trattamento di segnali video (prima analogici, poi con la loro evoluzione anche digitali). Altri servizi proposti sono la creazione di software accessori per schede di cattura e la messa a punto dei *driver* appositi per tali schede. Tutta l'offerta è basata su hardware prodotto da terze parti.

Il mercato di riferimento di entrambe le aziende è sopratutto quello italiano, ma la qualità dei prodotti le ha portate a farsi conoscere anche all'estero: Cina, Giappone, Medio Oriente, Sudafrica ed Europa in generale. Il fatturato generato da OffiCINE si aggira intorno al milione di euro, mentre quello di DataSoft spazia a seconda delle necessità del mercato dal mezzo milione ai settecentomila euro.

Secondo un'analisi della stessa azienda, il mercato mondiale del digital signage si sta fortemente ampliando (come spiegato anche nel capitolo 2), ma in Italia, per le aziende che si inseriscono tra produttore e consumatore, utilizzando hardware prodotto da terze parti e personalizzandone il software, come OffiCINE e DataSoft, non c'è più possibilità di ulteriore espansione e pianificazione di nuovi investimenti su progetti innovativi, a causa della forte presenza burocratica che spesso limita le possibilità di azione. Ciò non coopera con l'andamento generale del mercato riscontrato, che presenta dei picchi di concentrazione della domanda a distanza variabile l'uno dall'altro, difficilmente prevedibili con una certezza significativa.

Questo è aggravato dalla sempre più crescente forza dei principali *competitor* delle due aziende descritte, che sono i rivenditori di società multinazionali molto sviluppate, come ad esempio la Barco, la Christie Digital, la Eyevis e Delta India Electronics. Queste società sono tutte basate all'estero e spesso provenienti da mercati in via di sviluppo, come quelli della Cina e dell'India.

3.2 Metodologia di sviluppo

Come trattato nel capitolo 2, il progetto nasce a causa dell'esigua, o molto probabilmente nulla, presenza di software di questo genere sul mercato ed in seguito alla necessità di avere un prodotto con determinate specifiche che fosse, come detto, *open source* e seguisse tutti i criteri stabiliti al capitolo 1.

I tempi di sviluppo disponibili sono stati piuttosto brevi, abbastanza solo per il completamento di un applicativo funzionante; è stato necessario, perciò, stabilire sin da subito una serie di passi per uno sviluppo rapido e mirato alla produttività. A inizio attività, non è stato formalmente stabilito un chiaro modello da seguire; alla fine dello sviluppo è possibile affermare che il modello utilizzato è risultato essere quello "a cascata". Alcune fasi del modello a cascata tradizionale sono state però riviste per adattarsi meglio alla realtà nella quale il progetto ha visto la luce: il modello finale è risultato rientrante, cioè alcune fasi prevedevano anche il ritorno ai passi precedenti, ed è stato leggermente modificato e semplificato in corso d'opera.

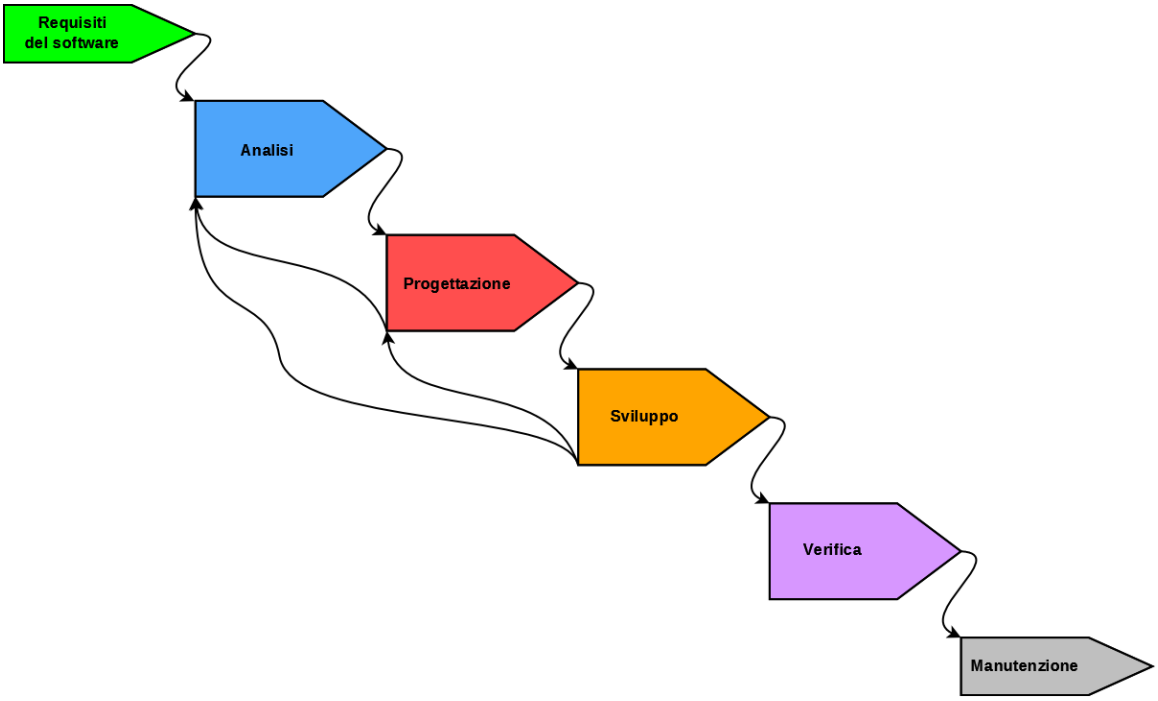

Figura 5: Modello di sviluppo pensato per l'applicazione realizzata.

Come è possibile notare in figura 5 (a pagina 27), alcuni passi del tradizionale modello a cascata sono stati eliminati per velocizzare il processo di sviluppo. Per quanto riguarda i passi mancanti, si è potuto far ricorso alla già descritta esperienza nel campo maturata dall'azienda presso la quale è stata sviluppata l'applicazione.

I requisiti minimi del livello hardware e di quello software, infatti, sono stati definiti in maniera quasi automatica, facendo capo alle funzionalità che l'applicazione

avrebbe dovuto implementare.

L'insieme dell'hardware è basato su tre elementi:

- la periferica che effettua il *capture* delle immagini digitali;
- un calcolatore adibito a server, che si preoccupa di dirigere le operazioni di cattura e streaming;
- un secondo calcolatore, o un *pool* di calcolatori se si desidera un ambiente distribuito, viene utilizzato come client.

Le interfacce software e di comunicazione sono state ben delineate poiché sono servite alla realizzazione di pratica di alcuni criteri che l'applicazione avrebbe dovuto rispettare. Per quanto riguarda quelle hardware, tenendo conto delle funzionalità allo stato attuale, non sono risultate utili. Durante lo sviluppo sono stati applicati degli assunti e delle semplificazioni specificati più avanti alla sezione 3.3.

Sempre facendo capo all'esperienza nel campo già citata, un'analisi di mercato per raccogliere e fissare un insieme di standard di qualità a cui attenersi non è stata necessaria.

Continuando nella fase di analisi, ci si è, però, concentrati sul produrre una scaletta delle azioni che avrebbero potuto essere intraprese all'interno dell'applicazione e le conseguenti reazioni della controparte applicativa. In questo modo si sono fissati anche i vari punti critici del progetto (analizzati più avanti nel capitolo 4).

Le fasi da svolgere sono risultate le seguenti:

- comunicazione in input con la periferica di acquisizione frame;
- conversione di formato pixel, compressione e *multiplexing* dello flusso video risultante;
- inizializzazione e gestione dello streaming, con conseguente preparazione della rete;

- passaggio dei dati tra il *core* di calcolo e l'interfaccia grafica;
- configurazione degli elaboratori del client distribuito alla ricezione dello stream da datagrammi UDP provenienti da un indirizzo IP multicast.

Una volta giunti alla fase di analisi dell'interfaccia grafica, si sono potuti definire i *trigger* sui quali un eventuale utente ha la possibilità di agire. Sul lato server essi si sono configurati in:

- scelta della periferica di acquisizione, tra quelle valide e connesse all'elaboratore;
- visualizzazione di una preview del flusso video che successivamente sarà immesso nella rete;
- abilitazione e disabilitazione dell'invio dei dati in streaming;
- creazione di un percorso statico di *routing* per indirizzi IP multicast su un'interfaccia di rete scelta;
- accesso ai settaggi della cattura e dello streaming, con possibilità di modifica;
- accesso a una wiki con la guida d'uso, quindi di aiuto in linea.

Dalla parte client, invece, i *trigger* progettati sono stati:

- creazione della configurazione riguardante videowall, quindi scelta della dimensione della matrice, della dimensione di ogni monitor e dei bordi;
- caricamento della configurazione e del servizio di visualizzazione tramite un protocollo di scambio dati in rete;
- avvio del servizio, che si metterà in attesa;
- ricezione dei pacchetti in streaming, ritaglio della zona interessata e conseguente riproduzione di ogni frame.

Successivamente si è passati alla fase di progettazione. Durante questo step è stato prodotto un *mock up* dell'interfaccia grafica, insieme ad un primo schema di come le varie azioni andassero ad integrarsi fra di loro e da quale parte logica dell'applicazione fossero svolte.

In questo modo, si è ottenuto anche un modello utile come punto di riferimento per orientarsi tra i vari layer e capire a quali funzionalità fossero associati (un'approfondita analisi di questo passo si trova più avanti, alla sezione 3.3).

Dopo la fase di codifica dell'applicazione, nella quale si sono concretizzate le idee e gli spunti visti nei passi precedenti, si è giunti alla fase di verifica. Questa è stato fondamentale per il funzionamento dell'intero progetto poiché sono stati rivisti e corretti alcuni meccanismi tecnici alla base delle funzionalità principali del software, quali la cattura e memorizzazione dei frame catturati sotto forma di immagini digitali, la loro successiva conversione e il *multiplexing*, il passaggio dei dati dal back-end al front-end applicativi e la realizzazione dell'interfaccia grafica, con aggiornamento istantaneo della vista di preview e delle impostazioni. Queste ultime rappresentano le principali criticità incontrate lungo tutta la realizzazione del progetto (saranno trattate in dettaglio nel capitolo 4).

La verifica del software è stata fatta per mezzo di test studiati appositamente per l'applicazione in fase di sviluppo. Grazie alla presenza di altri esperti nel campo video non coinvolti nello sviluppo del progetto, è stato possibile organizzare una valutazione dell'usabilità dell'interfaccia grafica e della corretta applicazione dei principi definiti precedentemente, e verificare se l'astrazione delle operazioni eseguite ai layer più bassi del software fosse efficace. Ad esempio, procedure in realtà complesse, come l'avvio della fase di streaming, sono state semplificate al massimo per non richiedere all'utente finale uno sforzo eccessivo nella comprensione della GUI.

L'ultimo passo del modello a cascata usato è stato quello riguardante manutenzione del software. Questo step per sua natura non è destinato a una vera e propria conclusione, infatti si svolge continuativamente per tutta la vita dell'applicazione e, spesso, solo in caso di necessità (per esempio nel caso in cui venga riscontrato un problema o ci si accorga di qualche errore commesso in fase di sviluppo, sfuggito nella

fase di verifica). Miglioramenti e aggiornamenti delle funzionalità, con possibilità di rilascio di nuove versioni del software, sono alcune tra le opzioni studiate in questo ultimo passo del ciclo di vita descritto.

3.3 Organizzazione e implementazione

Lo sviluppo dell'applicazione è avvenuto seguendo i principi fissati precedentemente nella fase di progettazione, uno su tutti è quello della portabilità. Ogni componente tiene conto di questa logica ed è appositamente scritta per essere fortemente integrata, ma allo stesso tempo indipendente dalle altre; sarà, seguendo la stessa logica, successivamente possibile sostituire parte degli ingranaggi per adattare il sistema a funzionare su piattaforme non previste nella fase iniziale e, allo stesso modo, saranno resi semplici e incoraggiate l'estensione e il miglioramento di alcuni elementi, insieme allo sviluppo di altri completamente nuovi.

Uno degli obiettivi principali che si è voluto raggiungere durante la progettazione dell'applicazione è quello di avere due parti totalmente indipendenti l'una dall'altra. Una prima parte è costituita dal software che funge da server ed una seconda è formata dall'insieme dei sistemi che formano il client distribuito.

Per la prima parte, il server, si è quindi deciso di creare la divisione fisica delle due componenti sottosistema-interfaccia grafica, giungendo quindi alla creazione di un *back-end* di calcolo e di un *front-end* grafico. Il back-end, quindi, assume le funzionalità di una libreria a più ampio spettro d'uso.

L'idea alla base di questa suddivisione del codice è stata quella che non solo il back-end fosse una libreria completamente indipendente dalle altre parti scritte e che potesse funzionare autonomamente anche in altre applicazioni totalmente differenti o di perfino di diversa natura (secondo il principio del riutilizzo del codice), ma che anche il front-end fosse sviluppato in modo tale da essere slegato dal sottosistema appena descritto.

Le due parti sfruttano per l'appunto il concetto di interfaccia pubblica. Tale interfaccia è servita sia per definire le azioni da compiere, sia per fissare dei capisaldi nello sviluppo. Viene sfruttata dai possibili utilizzatori (di cui fa parte il front-end), nel

senso che questi usano liberamente le funzioni in essa definite, senza però essere connessi direttamente all'implementazione sottostante. L'implementazione che permette all'interfaccia di svolgere il suo compito è invece assolta dal back-end, che la sfrutta come punto fermo a cui attenersi.

Per la seconda parte invece, si è ideato un sistema completamente autocontenuto che funzioni indipendentemente dal software utilizzato come server. Si può considerare come un progetto derivato che completa il pacchetto di funzionalità, senza però essere legato strettamente al resto delle applicazioni implementate. Il client è incentrato sulla ricezione da parte di ogni elaboratore incluso di un flusso in streaming che verrà poi riprodotto in parallelo sui monitor collegati a questi ultimi.

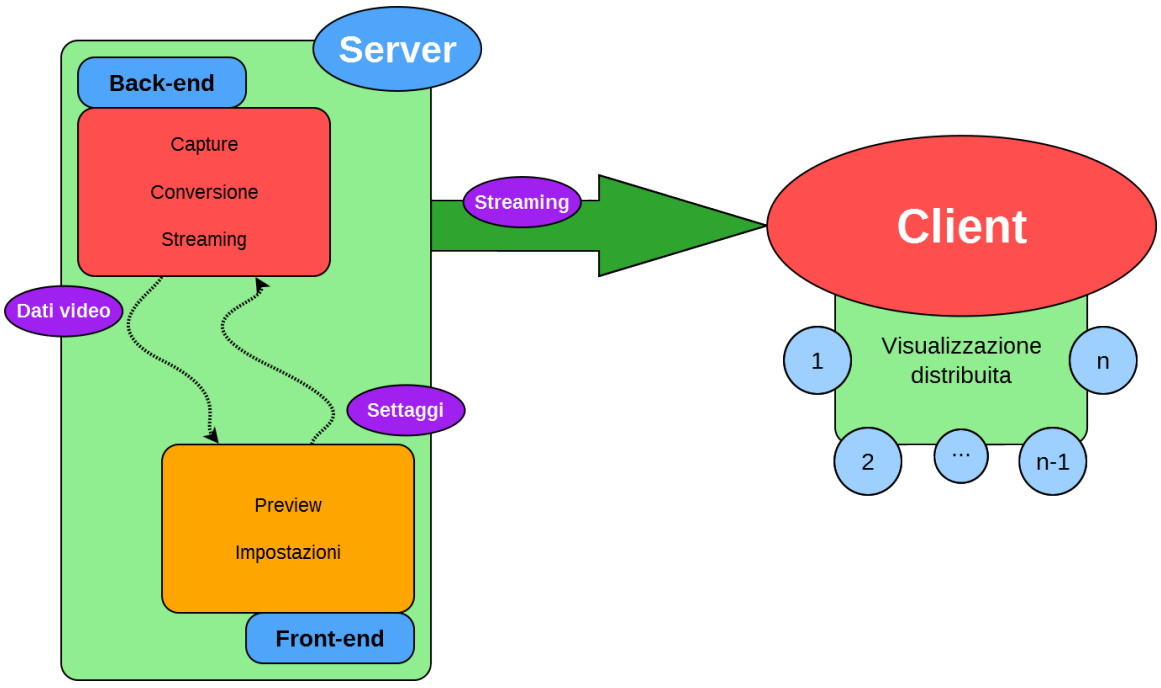

Figura 6: Schema di funzionamento dei diversi moduli nell'applicativo.

La figura 6 (a pagina 32) mostra un diagramma che raffigura la struttura del progetto dal punto di vista dello scambio di dati. I dati in primo luogo vengono acquisiti da una scheda di *capture* di immagini digitali, collegata alla parte server.

Tali dati vengono memorizzati in un buffer che poi sarà reso disponibile a tutta la libreria.

La prima componente che manipolerà questo buffer è il back-end. Esso si occuperà in primis di convertire i dati affinchè siano comprimibili dal compressore, nello specifico convertirà il formato del pixel in input (che può essere di tipo eterogeneo) in uno appartenente al modello Y'UV; dovendo anche fornire i dati per un'eventuale preview si preoccuperà di convertire una seconda volta il formato pixel di input in un altro, questa volta facente riferimento al mondo RGB. Successivamente, il front-end si occupa di inizializzare la rete e aprire un formato *container* che verrà alla fine copiato sulla rete stessa. All'interno di questo *container* verranno copiati i dati che saranno il risultato della compressione messa in atto sempre dal *core*, con i settaggi provenienti invece dal front-end.

I dati poi vengono, tramite la già descritta interfaccia pubblica, passati al front-end, che li usa per mostrare una preview all'utente. Nell'interfaccia grafica sono presenti dei menu che permettono sia di selezionare i settaggi desiderati da inviare al *core*, sia gli *hook* per poter dare inizio a una sessione di streaming. Quando viene fornito il comando per aprire lo streaming, il front-end sottostante inizia a copiare i dati sulla rete.

Per mezzo dello streaming in rete, i dati vengono finalmente passati al client. Il client non è esclusivamente locale, ma c'è la possibilità che sia distribuito. Se si tratta di un client autocontenuto, i dati vengono semplicemente portati in riproduzione, se invece si tratta di un client distribuito ci sarà una configurazione videowall. Questa configurazione, prevede un elaboratore per ogni monitor che si occuperà di ricevere il flusso in arrivo e ritagliare la parte di interesse, secondo la posizione, e infine anch'esso porterà la sezione ritagliata in riproduzione.

Come formato di codifica del flusso di dati trasmesso in streaming è stato scelto H.264. H.264 è conosciuto anche come MPEG-4 parte 10 AVC (*A*dvanced *V*ideo *C*oding) poiché fa parte della specifica MPEG ed è stato sviluppato dal Joint Video Team, uno sforzo condiviso di Video Coding Experts Group e Moving Picture Experts Group, con le migliori innovazioni dell'epoca. Nel corso degli anni è diventato il codec di riferimento del mondo streaming, potenziandosi e raggiungendo una stabilità difficilmente visibile in altri progetti.

La scelta è ricaduta su questo codec, poiché, oltre alcune tra le più recenti innovazioni nel campo, presenta delle opzioni dedicate esclusivamente allo streaming: un esempio è `fastart`, che permette di inserire il `moov atom`[2] all'inizio del file. Ciò dimostra anche la flessibilità dello standard, rendendolo usabile in diversi ambienti; non è un caso che sia diventato l'unico codec usato nei campi di digital signage e live streaming.

Esso è un codec orientato alla compressione di blocchi di immagine tramite la *motion compensation*, cioè la compensazione del moto, con la possibilità di comprimere sia le immagini dei frame stessi, e questo metodo è detto *spatial intra-picture prediction*, che un gruppo, quest'altro invece è chiamato *inter-picture prediction*.

La prima tecnica si basa invece sulla predizione spaziale. Vengono, cioè, considerati dei fino a 16×16 pixel, detti macroblocchi e su ognuno di essi viene calcolata la differenza spaziale con il suo vicinato. Inoltre su ogni frame viene effettuata della compressione sfruttando le trasformate discrete del coseno, sempre su macroblocchi con le stesse limitazioni di quelli della predizione spaziale. Considerando dei macroblocchi, si potrebbe avere il problema che nella resa finale questi siano troppo visibili. H.264 include anche un filtro di *deblocking*, cioè di rimozione dell'aspetto squadrato all'interno dell'immagine.

Invece, la seconda tecnica sfrutta l'assunto, vero nella maggior parte dei casi, che il frame $n-1$ sia molto simile al frame n, ad eccezione di alcuni oggetti nella scena che si sono mossi. La predizione inter-frame agisce in questo modo: esamina un range di frame, ad esempio dal frame 0 al frame n (questi due frame verranno chiamati frame di tipo I). Successivamente considera il frame $\frac{n}{2}$, che viene calcolato come la differenza pixel per pixel tra il frame $\frac{n}{2}$ e la media fra il frame 0 e n; ciò che ne risulta è un frame di tipo P, poiché è *p*redetto dai frame di tipo I. Se invece la predizione ha luogo tra un frame di tipo I e un altro di tipo P, il frame risultante è doppiamente predetto, perciò assume la nomenclatura di B (*b*ipredicted).

Lo streaming non si può avvantaggiare in ampia maniera di questa tecnica poiché un ipotetico client che riceve il flusso di dati, dovrà aspettare un frame di tipo I, prima

[2]Il `moov atom`, meglio conosciuto come *movie atom*, definisce la scala temporale, la durata e le caratteristiche di visualizzazione dello stream H.264, oltre a contenere *subatoms* riguardanti ogni eventuale traccia video contenuta in quest'ultimo.

di poter decodificare correttamente gli altri. Se il range di frame considerato è molto ampio questo potrebbe comportare una latenza molto grande. Da ciò ne consegue che un flusso H.264 orientato allo streaming avrà la maggior parte di frame I.

Lo spazio colore di riferimento di questo codec è Y'UV[3]. Di solito, almeno per quanto riguarda l'universo dello streaming, il colore, ovvero le componenti UV, che accompagna la componente Y è sottocampionato. Il campionamento tradizionalmente usato, soprattutto se si desidera ottenere una compressione efficace, è 4:2:0, cioè i flussi UV sono un quarto di quello Y. Quest'ultimo sarà quello usato anche nell'ambito del progetto.

H.264 presenta anche altre funzionalità: profili, *preset* e *tune*.

I profili sono un insieme di opzioni preconfezionate che variano a seconda dell'applicazione per cui lo stream è prodotto. Il vantaggio è che un ipotetico *decoder* può riconoscere immediatamente un set di impostazioni preconfezionate appartenenti a un certo profilo. Ce ne sono di vario tipo, uno dedicato alle videoconferenze o alle applicazioni a basso costo, un altro dedicato alla compressione di flussi video in definizione standard e un ultimo specifico per contenuti in alta definizione. È possibile notare che ne esiste uno molto simile a quello per i contenuti in HD (*H*igh *D*efinition), ma dedicato esclusivamente allo streaming e include un'altra compressione e alcune opzioni per contrastare la perdita di dati durante la trasmissione.

I *preset* riguardano più il server che prepara lo streaming. Si occupano di definire un guadagno di qualita a scapito del tempo impiegato. È possibile scegliere tra sei opzioni: `ultrafast`, `superfast`, `veryfast`, `faster`, `fast`, `medium`, `slow`, `slower`, `veryslow` e `placebo`. Com'è facile intuire con `ultrafast` si ottengono i tempi di compressione migliore, mentre con `placebo` la qualità migliore. Per il progetto, sebbene si sia mantenuta la possibilità di scelta, l'opzione di default e quella consigliata è `ultrafast`.

Infine, i *tune* impostano alcune opzioni a seconda dello scopo per cui è prodotta la compressione. Anche qui il range di opzioni è limitato: `film`, `animation`, `grain`, `stillimage`, `psnr`, `ssim`, `fastdecode` e `zerolatency`. Tra queste l'opzione dedicata

[3]Lo spazio colore Y'UV a cui ci si riferisce all'interno di questo scritto è in realtà lo spazio YCbCr. Per ragioni storiche quest'ultimo è spesso usato se si parla di segnali analogici, mentre il primo se vengono trattati segnali digitali.

allo streaming è `zerolatency`, che rende la compressione priva di frame di tipo P e di tipo B. Ovviamente è anche quella usata nell'ambito del progetto.

Per quanto riguarda i container usati nel progetto, questi sono principalmente due (anche se la scelta rimane comunque possibile): MPEGTS e AVI.

MPEGTS (MPEG *T*ransport *S*tream) è il formato di contenimento apposito per trasmissioni lungo canali non affidabili che il gruppo MPEG ha ideato intorno al 1995. È usato principalmente per trasmissioni televisive terrestri o satellitari. Si basa su pacchetti elementari di flussi video e audio e contiene dei meccanismi per la correzione degli errori introdotti dalla invio e per la sincronizzazione del flusso, per mantenere l'integrità di quest'ultimo anche quando il segnale di trasmissione viene degradato. MPEGTS è compatibile con le codifiche di tipo MPEG, specialmente MPEG-2 e MPEG-4, tra le quali figura anche H.264, il formato usato nel progetto.

AVI (*A*udio *V*ideo *I*nterleave), invece, è un formato ideato da Microsoft nel 1992. Attualmente viene usata una particolare versione della specifica con estensioni elaborate dalla Matrox Video. Questo *container* è molto semplice e viene introdotto per gestire la compatibilità con altri sistemi. Un AVI è diviso in varie sezioni definite *chunk*; il *chunk* denominato `movi` contiene i dati del flusso audiovisivo. Per mezzo di questo sistema un *container* di questo tipo può virtualmente contenere qualsiasi tipo di compressione al suo interno, H.264 compreso[4].

3.3.1 Back-end

Il substrato dell'applicazione è rappresentato appunto da una libreria che è stata denominata `libv4lcapture`, studiata per prima in maniera sostanzialmente distaccata dal resto del progetto. Come già specificato tale libreria può essere utilizzata per altri scopi (come ad esempio la creazione di un'altra applicazione di *capture* → *encoding* → *streaming*), oppure un'altra completamente di natura differente; si può inoltre impiegare in maniera diretta, senza un'interfaccia grafica, ma sfruttando l'applicativo

[4]Un *container* di tipo AVI che incapsula uno stream in formato H.264 sarebbe tecnicamente non realizzabile. Tramite alcuni stratagemmi, ovvero rinunciando ad alcune funzionalità proprie del formato di compressione si può però renderlo possibile.

`test` (un semplice software creato per per verificare il funzionamento complessivo del sistema).

Come si nota in figura 6 (a pagina 32), il cuore del progetto scambia dati con la parte anteriore dell'applicazione, l'interfaccia grafica. In questo frangente è evidenziato il ruolo chiave dell'interfaccia pubblica: infatti, per riuscire a trasmettere dati e contemporaneamente rimanere indipendente dal sistema con il quale questi vengono scambiati, l'unico modo che è messo a disposizione è la creazione di una API pubblica. Questa API definisce in aggiunta la direzione del flusso dei dati all'interno della libreria e l'interazione tra i vari sottomoduli della stessa.

Com'è noto le operazioni di cattura dei dati, ricodifica e compressione e streaming, sono operazioni soprattutto *CPU bound*, cioè richiedono un uso intensivo del processore per effettuare molti calcoli, anche complessi. Per questo motivo si è deciso di non interporre un ulteriore strato, come ad esempio un interprete o una macchina virtuale, nello svolgimento di questi compiti, ma piuttosto d'implementare la libreria nel linguaggio C, trovandosi così ad avere la possibilità di utilizzare le *feature* specifiche del processore in uso.

Sempre allo stesso scopo, è stata presa la decisione di usare l'ottima libreria, licenziata sotto regime GPL, "FFmpeg"; questa libreria è una delle più usate in ambito di conversione e compressione audio e video, poiché al suo interno comprende già il supporto a molteplici spazi colore e tipi di pixel, numerosi *encoder* e formati *container*, insieme alla possibilità nativa di scrivere e leggere i file prodotti, indipendentemente, sia su una memoria ad accesso diretto, sia su rete locale o globale.

All'interno della medesima libreria, queste caratteristiche sono state implementate tenendo preventivamente conto delle istruzioni specifiche che l'architettura in uso di volta in volta mette a disposizione del programmatore; per questo motivo, alcune routine sono state scritte direttamente utilizzando istruzioni del linguaggio assembly specifico per quel processore (che ovviamente mutano da un'architettura all'altra).

In quanto sarà nel futuro necessario mantenere il codice scritto in questa fase, è necessario che quest'ultimo sia organizzato sotto buona struttura; a questo scopo la libreria è stata suddivisa semanticamente in sottomoduli. Per ogni sottomodulo presente è risevato un file specifico. Le sezioni, e quindi i file presenti, presenti sono i

seguenti:

`capture.c`

Questo sottomodulo contiene tutte le procedure atte ad inizializzare la periferica di acquisizione e trasportare i frame, servendosi del *bus* di collegamento, da quest'ultima ad un buffer in memoria. Inoltre in questa parte viene creato, inizializzato e messo a disposizione della libreria il contesto riguardante i dati di cattura. Un estratto del codice prodotto che mostra principalmente la sequenza di azioni intraprese è disponibile nell'appendice A.1.

Per essere compatibile col maggior numero di periferiche e, allo stesso tempo, rimanere nello standard si è scelto di utilizzare il framework Video4Linux[5]. Tra le varie possibilità questo sistema gestisce anche l'acquisizione delle immagini dalle periferiche, rendendo completamente trasparente l'interazione col kernel Linux sottostante e focalizzandosi sulle direttive da dare agli strumenti in uso.

Nell'ambito di V4L, le periferiche di acquisizione vengono controllate tramite chiamate alla funzione `ioctl`, con parametri che di volta in volta specificano l'azione da eseguire; tutti i parametri riguardanti la cattura di immagini sono denominati dal prefisso `VIDIOC_`. Un esempio di chiamata a `ioctl` è possibile osservarlo alla funzione `start_capture`.

V4L, per effettuare l'acquisizione dei frame video, mette a disposizione due funzioni: `read`, la più semplice da implementare ma che troppo spesso risulta inefficiente, e `mmap` (*m*emory *map*ping), che è di più largo uso in quanto alloca uno spazio di memoria riservato esclusivamente alla memorizzazione seriale dei frame e lo mappa nella memoria virtuale del processo.

Per poter arrivare ad avere i frame catturati via, via a disposizione all'interno di un buffer condiviso in memoria bisogna seguire un percorso che comprende alcune azioni da intraprendere. Per primo bisogna creare un `context` di cattura tramite la funzione `init_ctx`, la quale assegna dei valori di default alle sue variabili richiesti direttamente alla periferica tramite la routine `get_default` solo

[5]Il framework Video4Linux (anche abbreviato con V4L) è un'insieme di interfacce pubbliche compreso nel kernel Linux riguardanti la cattura video e le periferiche di output video; supporta diverse schede di *capture*, webcam, sintonizzatori TV e altre periferiche.

dopo aver eseguito l'apertura come flusso di byte di quest'ultima, utilizzando la funzione `open_device`.

In fase di cattura, naturalmente, è possibile specificare diversi parametri, che cambiano il tipo di dati restituito, come la dimensione dell'immagine catturata, il *framerate*, il tipo di pixel desiderato e il ritaglio dell'immagine voluto. Quest'operazione si effettua modificando i corrispondenti valori di `width`, `height`, `framerate`, `pixelformat` e `crop` nel contesto creato precedentemente. A questo punto il sistema ha piena coscienza dell'entità dei dati che devono essere catturati e può inizializzare le strutture atte a contenerli in memoria. Viene perciò chiamata la funzione `read_frame` che inizializza la sequenza di inizializzazione periferica, caricando i dati settati prima e le strutture associate alla funzione scelta per la cattura, e successivamente richiede un frame e lo memorizza in una coda all'interno del contesto, alla variabile `buffers`, contenente un array di buffer. Un puntatore al frame corrente all'interno del buffer (utile per l'elaborazione da parte di altri moduli) viene offerto nella variabile `work`.

Una volta terminata la cattura, questo modulo permette anche di fermarla in maniera corretta, chiamando la procedura `stop_capture`. Il passo successivo è deallocare la zona di memoria riservata al contenimento delle immagini catturate e ciò si effettua tramite la funzione `uninit_device`. L'ultima azione da compiere è chiudere correttamente la periferica in uso per renderla disponibile ad altri processi e deallocare il contesto creato, usando `close_device`.

`encoding.c`

In questo modulo vengono trattati la conversione, la compressione, il *multiplexing* e lo streaming dei frame salvati precedentemente nel buffer condiviso e l'inizializzazione e uso del contesto di *encoding*. Come guida si può tenere il codice proposto nell'appendice A.2.

Per prima cosa, viene creato un `context` di compressione e inizializzato con valori di default oppure provenienti dal modulo di cattura. I dati catturati, come già descritto, rimangono disponibili nel buffer `work` e le operazioni agiranno su tale buffer.

Questa parte di libreria deve assolvere vari compiti; tra questi, due parti fondamentali sono rappresentate dalla conversione del formato pixel in entrata (che in

una situazione standard è di tipo Y'UY'V[6] con sottocampionamento della chroma a 4:2:2[7]) una prima volta nel modello Y'UV con *chroma subsampling* a 4:2:0 per la successiva compressione e una seconda nel modello RGB a 24 *bit*, 8 per canale, per fornire i dati per la preview. Questa operazione è fatta in due passi per ogni modello; nel primo, tramite le funzioni `scaler_init` e `rgb24_init`, si inizializzano le strutture per la conversione e nel secondo, per mezzo delle funzioni `scale` e `rgb24` viene effettivamente effettuata la conversione.

Successivamente il risultato in RGB 24 viene consegnato direttamente al *font-end*, mentre quello in YUV 4:2:0 planare continua il processo di compressione. Tale processo si sviluppa eseguendo l'*encoding* e il *multiplexing* di suddetti dati. Anche questa procedura si ramifica in due routine diverse. Una, `mux_encoder_init`, si occupa di:

- inizializzare la rete scegliendo uno dei protocolli a disposizione, in questo caso incapsulando in semplici pacchetti UDP, ma sarebbero valide soluzioni più complesse, quali RTSP, RTMP o MPEG-DASH;
- di impostare alcuni parametri per l'encoder H.264, tra i quali, oltre al *framerate* e alla qualità desiderata, compaiono i già citati profili *preset* e *tune*, insieme alla fondamentale opzione `faststart`;
- di aprire l'*encoder*, impostandolo per comprimere in H.264, includendo le impostazioni specificate;
- di aprire il *multiplexer* scrivendo l'apposito *header* all'inizio dello stream (verso un indirizzo IP multicast).

L'altra parte, avviata dalla funzione `mux_encode` si fa carico della reale compressione in H.264 dei frame. Alla fine di questo ciclo di calcolo, si svolge un'altra

[6] Y'UY'V è un particolare modello Y'UV che a differenza di quest'ultimo non è planare, ma pacchettizzato. Nel modello Y'UV sono presenti tre piani contenenti ognuno un canale, mentre in quello Y'UY'V i dati sono salvati sequenzialmente sullo stesso piano.

[7] Il sottocampionamento del segnale della chroma, o *chroma subsampling* è la riduzione dell'informazione colorimetrica se confrontata con quella di luminanza. È espressa in triplette di valori del tipo 4:2:0, in cui il primo valore indica i campioni Y', il secondo quelli U e il terzo quelli V. Se appare uno zero all'ultimo posto significa che il valore di V è identico a quello di U.

routine, `write_cached`, che invia in rete gli ultimi frame compressi rimasti nel buffer e scrive il *trailer*, cioè il corrispettivo dell'*header*, ma alla fine del flusso creato.

Alla fine del modulo, sono presenti anche le funzioni dette di *uninit*, cioè `scaler_uninit`, `mux_encoder_uninit` e `conv_ctx_uninit`, che rispettivamente liberano la memoria associata ai contesti della conversione di formato pixel, dell'*encoder* e quello generale della compressione.

`error.c`

In questa sezione trova spazio la gestione delle situazioni di errore e l'informazione dell'utente di eventuali problemi. Un riferimento alle funzioni citate può essere trovato nell'appendice A.3.

Per gestire efficientemente l'informazione dell'utente di eventuali errori nel corso dell'operato dell'applicativo, si è deciso di sfruttare il meccanismo di *logging* standard e già compreso in qualsiasi sistema operativo GNU/Linux, chiamato "syslog".

Al fine di una gestione efficiente, gli errori sono stati classificati prima secondo tipologia, poi secondo secondo gravità. Vengono individuati tre tipi di errori: quelli riguardanti il funzionamento o la comunicazione con periferiche, quelli riguardanti la capacità di una periferica di eseguire un dato compito ed infine gli errori generici, cioè quelli che non ricadono nei due casi precedentemente descritti. Le funzioni associate a questi tre tipi di errori sono rispettivamente contrassegnate col prefisso `dev` (cioè *device*), con il prefisso `cap` (che sta per *capability*) e col prefisso `gen`.

A loro volta ogni tipologia citata si può declinare in tre priorità diverse, da cui nasce il suffisso delle precedenti funzioni: `log` (o `warning`), `error` oppure `critical`. Ad ogni priorità corrisponde una gestione differente. `log` è solo un avviso di servizio all'utente che notifica il completamento di una routine spesso prona a generare errori. `error` e `critical` delineano quegli errori che precludono il completamento di qualche funzione e vengono gestiti allo stesso modo, ad eccezione del fatto che nel caso di un errore `critical` è prevista la cessazione immediata dell'esecuzione di tutto il processo.

`util.c`

Questo sottomodulo contiene tutte le procedure di utilità.

In particolare, è stata inclusa una routine che è un *wrapper* di `ioctl`, denominata `xioctl` (e visibile nell'appendice A.4. Come si è visto precedentemente, questa funzione è importante poiché è il ponte di comunicazione tra l'applicativo e la periferica.

Questo *wrapper* controlla il valore di ritorno di `ioctl` e nel caso in cui questo non sia positivo ritenta l'esecuzione della suddetta funzione. In caso di malfunzionamento o di valore di uscita positivo questa si comporta trasparentemente come farebbe la semplice `ioctl`.

`libv4lcapture.c`

In questo file, troviamo la dichiarazione dell'interfaccia pubblica della libreria e quindi tutte le funzioni accessibili al programmatore sotto forma di API. Un estratto contenente solo le funzioni che l'API pubblica offre è mostrato all'appendice A.6.

L'importanza dell'interfaccia pubblica definita all'interno di questo modulo è già stata più volte sottolineata. Per creare le funzioni appartenenti all'interfaccia è stato usato uno stratagemma: vengono inizializzati i due contesti di cattura e di compressione in maniera globale e statica, affinchè siano visibili all'interno di tutte le funzioni dichiarate. In questa maniera i contesti diventano invisibili all'utente finale, infatti verranno utilizzati esclusivamente in maniera privata all'interno delle funzioni esposte.

Ci sono procedure atte ad impostare all'interno della libreria i parametri impostati, e queste sono contraddistinte dal prefisso `set`, altre sono adibite alla consultazione dei valori già presenti, contrassegnate dal prefisso `get`. Le routine rimanenti sono invece funzioni di controllo, cioè permettono di specificare l'azione che la libreria dovrà compiere (ad esempio catturare un frame, comprimerlo o deallocare un contesto).

Le funzioni dichiarate in questo file hanno tutte una particolarità: non hanno argomenti oppure li hanno solo di natura primitiva (il motivo sarà esplicitato chiaramente nel capitolo 4). Una routine degna di nota è quella di richiesta dei dati di preview provenienti dalla periferica di cattura, poiché questi ultimi

saranno gli unici dati che non siano valori primitivi come interi o stringhe di caratteri, passati al layer superiore dell'interfaccia grafica.

Sono a tutti gli effetti considerabili come facciate dell'infrastruttura della libreria, con lo scopo di effettuare una semplificazione orientata all'uso. Inoltre, un altro fine primario è quello di permettere la modifica o la richiesta di diversi parametri e l'avviamento delle fasi viste nei diversi sottomoduli senza incorrere in qualche rischio dato, ad esempio, dalla errata specificazione dei parametri.

3.3.2 Front-end

Il front-end, denominato `V4LCapture`, dà il nome a tutto il progetto e si concretizza in un'interfaccia grafica per la libreria appena descritta.

È stato pensato per essere facilmente estensibile e completamente slegato dall'implementazione delle procedure sottostanti. Il compito principale di questo modulo è quello di fornire un'accesso alle funzionalità semplice e comprensibile all'utente finale, semplificando, o direttamente nascondendo, la maggior parte dei dettagli strettamente correlati all'implementazione, dettagli che, altrimenti, renderebbero poco fruibile tutta l'applicazione.

Al contrario del back-end, il front-end dev'essere dinamico, inoltre non necessita di una particolare ottimizzazione in termini di affinità con il processore, ma deve privilegiare la facilità e la rapidità di sviluppo ed estensione. Per questo motivo (ed altri spiegati meglio nel capitolo 4) si è deciso di adottare la versione 3 del linguaggio Python abbinata al *framework* grafico Qt5, grazie al progetto PyQt5. Python fornisce un paradigma di sviluppo dinamico, che permette di creare con facilità strutture, anche complesse; allo stesso modo, Qt5 offre una pletora di *widget* grafici preconfezionati, minimizzando allo stretto necessario il bisogno di crearne ex novo. In ultima analisi, questo insieme di tecnologie ha reso possibile, oltre ai vantaggi già elencati, restare fedeli al principale obiettivo di restare nell'ambito dell'*open source*.

I punti focali dell'interfaccia grafica sono tre e possono essere riassunti nei seguenti elementi:

Finestra principale

La finestra principale è quella che viene mostrata all'apertura dell'applicazione ed è simile a quella osservabile in figura 7 (a pagina 44). Al centro è visibile una preview dell'immagine acquisita indipendente dallo streaming, una barra dei menu dove sono accessibili le varie funzioni ed una barra di stato dalla quale è possibile apprendere se è in corso una sessione di streaming e, nel caso, le informazioni ad esso associate (come IP e porta, codifica utilizzata e settaggi di qualità).

Figura 7: Finestra principale dell'applicazione.

In questa vista, il *widget* di preview disposto centralmente, mostra i frame catturati ma non ancora compressi e fornisce un'anteprima del flusso di dati da cui avrà origine lo streaming. I dati per questa preview provengono direttamente dalla libreria sottostante. È uno dei due flussi di dati, quello che non è stato compresso, ma convertito di formato pixel in RGB a 24 *bit*. Ogni frame con cui viene aggiornata la preview è fornito dalla funzione dell'interfaccia pubblica

`getRGBData`, che è l'unica che non ritorna valori primitivi, ma, appunto, un array con valori per R, G e B.

Continuando nell'esame della GUI (*G*raphic *U*ser *I*nterface), si arriva alla barra dei menu. La prima voce di tale barra è il fulcro per controllare l'intera applicazione e impostare dei parametri nella libreria sottostante. Da questo punto di accesso è infatti possibile scegliere la periferica, tra quelle collegate al sistema, che fungerà da sorgente di immagini, aprire un cammino di rete statico verso indirizzi IP multicast per lo streaming e accedere alla vista delle impostazioni, da cui è possibile modificare la resa o la qualità del flusso video.

Scelta della periferica e impostazioni

La scelta della periferica di *capture* tra quelle collegate al sistema avviene seguendo un algoritmo espresso in due funzioni (riportate nell'appendice B.2). Per primi vengono esaminati tutti i *device* fisici e virtuali che il kernel Linux mette a disposizione al punto di *mount* `/dev`. Successivamente questi vengono filtrati per considerare solo quelli di tipo `/dev/videoN` (dove `N` indica un numero compreso tra 1 e 63). Le periferiche di questo tipo sono quelle riconosciute e gestite da Video4Linux e saranno quelle mostrate all'utente nell'apposito menu dell'interfaccia grafica.

Altra voce fondamentale nel menu è quella delle impostazioni. Questa vista a sua volta è divisa in due sottosezioni: *capture* e streaming (visibili affiancate in figura 8 a pagina 46).

La prima vista raffigurata a sinistra nell'immagine 8 permette di specificare le impostazioni di acquisizione, quindi quelle trattate dal modulo `capture.c`. È possibile specificare il metodo di acquisizione in input delle immagini, tradotto nelle due funzioni disponibili già descritte `read` oppure `mmap`. Successivamente si trovano dei menu a tendina dove è possibile scegliere il numero di frame al secondo catturati dalla periferica, i valori di *crop*, cioè di ritaglio delle zone periferiche dell'immagine (specificate in distanza dall'alto e da sinistra e altezza e larghezza), e la dimensione di output dei frame catturati (in termini di altezza e larghezza).

La seconda schemata rende disponibili altre impostazioni riguardanti lo streaming, perciò inviate al modulo `encoding.c`. I settaggi accessibili sono l'indirizzo

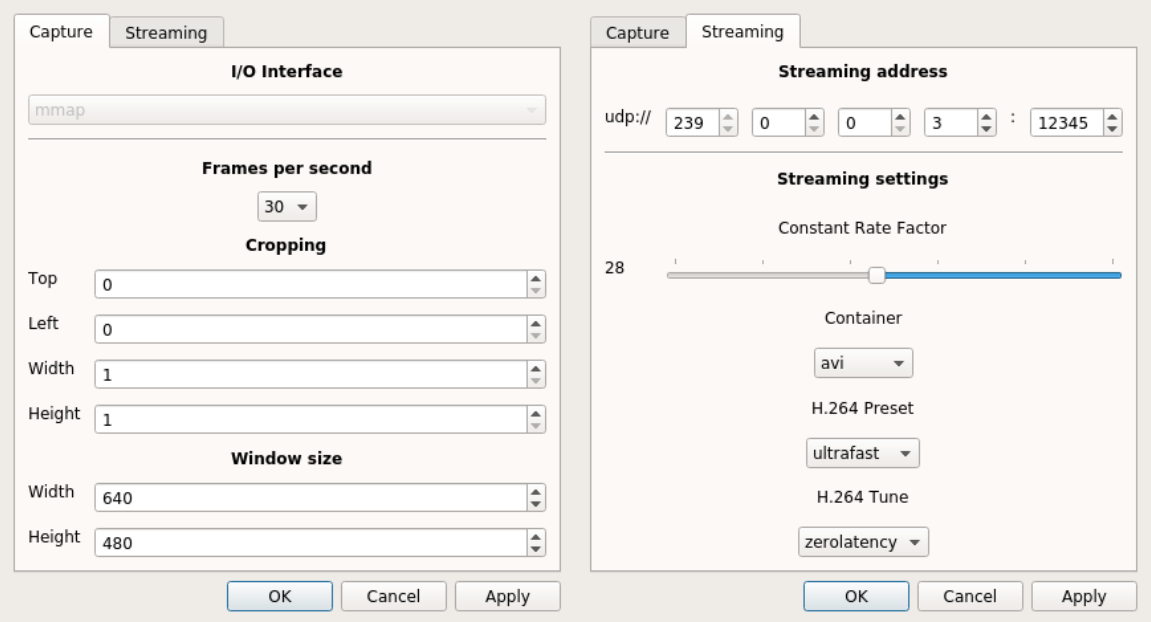

Figura 8: Pannello dedicato alle impostazioni (cattura a sinistra e streaming a destra).

IP multicast e la porta sui quali il flusso video compresso sarà aperto, la qualità (espressa in valori di CRF[8]) e le opzioni relative al codec e *container* usati, quali il *demuxer* che si desidera usare, il *preset* e il *tune*, già descritti in precedenza.

Thread di cattura immagini

La gestione della comunicazione con la libreria, e la conseguente acquisizione di informazioni dalla stessa, viene effettuata separatamente rispetto a quella dell'interfaccia grafica (a ognuna delle due viene assegnato un thread differente).

Il thread che controlla l'interfaccia grafica è il thread principale dell'applicazione. Il thread che controlla la cattura delle immagini, invece, è dedicato solo a questo compito. Come è possibile notare nel codice proposto all'appendice B.1, nella classe `CaptureThread`, questo si occupa della comunicazione con la libreria

[8]La qualità può essere espressa in *bitrate*, che ha come unità di misura il bit al secondo, oppure in valori CRF; il CRF, Constant Rate Factor, è selezionabile in un range di valori che spazia da 1 a 51, dove 1 è la qualità migliore e 51 qualità pessima, ed è un valore che piuttosto che quantificare il peso della codifica prodotta, quantifica la qualità ottenuta.

sottostante, tramite la API pubblica. Il suo compito è anche quello di compiere tutti i passaggi necessari (descritti alla sezione **??**) per catturare un frame e portarlo in streaming in sicurezza, abbassando le probabilità di generare errori. Prevede anche una variabile, `self.work`, che permette di avviare e fermare lo streaming, continuando comunque a catturare frame da mostrare nella preview.

È presente un'altra classe chiamata `CaptureParams` che si occupa del salvataggio e del ripristino delle impostazioni scelte. Al suo interno è presente un insieme di chiavi valide, nella tupla `VALID`, che, appunto, stabilisce quali sono le opzioni memorizzabili. Successivamente appaiono dei metodi per impostare e richiedere il valore di un dato settaggio, cancellare completamente i settaggi inseriti, ripristinare tutto all'ultimo salvataggio e salvare in una memoria temporanea la configurazione corrente. I dati salvati nella memoria temporanea vengono scritti in accordo alla sintassi JSON[9]. Alla chiusura dell'applicazione, questi ultimi verranno salvati in un punto preciso del file system, dove l'applicazione sarà di in grado di trovarli e caricarli alla successiva apertura.

3.3.3 Client

Il modello di client adatto per questo tipo di applicazione può essere di due tipi: a finestra singola o a multifinestra; quest'ultimo tipo è anche definito distribuito.

Restando nell'ambito dell'utilizzo di strumenti standard, un qualsiasi *player*, anche già esistente, che supporti la decodifica di frame H.264 e la ricezione di uno stream incapsulato in datagrammi UDP da un indirizzo IP multicast può adattarsi al progetto. In fase di sviluppo e di verifica sono stati utilizzati software ben noti, quali VLC, `mplayer` o `ffplay` (compreso nel progetto FFmpeg).

Per quanto concerne invece la versione distribuita, è stato studiato un modello che al contempo fosse accessibile e che centrasse gli scopi iniziali di essere *open source* e *open hardware*. Come sarà descritto meglio più avanti nella sezione 3.4, come

[9]JSON (*J*avascript *O*bject *N*otation) è un formato di salvataggio dati serializzati; è un'alternativa a XML (e*X*tensible *M*arkup *L*anguage) ed è compatibile con numerosi linguaggi di programmazione.

hardware si è scelto di utilizzare diversi Raspberry Pi, con maggiore precisione uno per schermo, corredati dal sistema operativo basato su GNU/Linux. Come software invece ci si è affidati ad un progetto esterno chiamato PiWall, con *feature* specifiche per la creazione di configurazioni multischermo o videowall. Questo sistema permette ai diversi Raspberry Pi di comunicare con il server, ma allo stesso tempo coordinarsi tra di loro e riprodurre una porzione di frame di flusso video a seconda della loro posizione nella matrice della configurazione videowall. PiWall, per ricevere il flusso in input, si basa sulla ricezione di stream video incapsulati in pacchetti UDP e provenienti da indirizzi multicast ed è nato per decodificare principalmente il formato H.264 incapsulato in un container di tipo AVI.

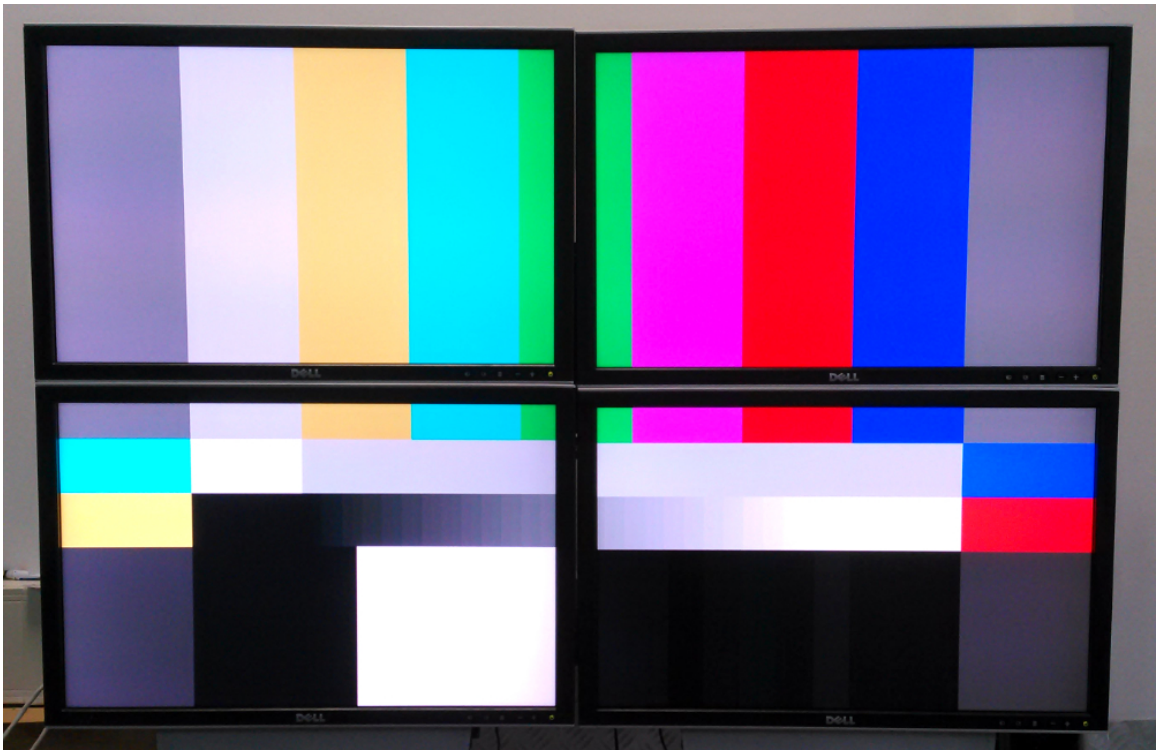

Figura 9: Il client distribuito nella configurazione utilizzata durante i test.

Il sistema PiWall legge due file di configurazione. Il primo file, `.piwall` (l'esempio generato durante il corso del progetto è disponibile nell'appendice C.2), specifica la dimensione della matrice che forma il videowall, insieme alla dimensione e alla posizione relativi ad ogni monitor singolo; inoltre, presenta un elenco di monitor

coinvolti, definiti *tile*. Comprese in queste misure viene anche effettuata quella che viene chiamata *bezel compensation* o *gap filling*, cioè la compensazione dei bordi all'interno delle dimensioni del flusso video.

Il secondo file, `.pitile` (anche di questo una copia relativa ad un solo monitor è presente all'appendice C.3), assegna invece un'etichetta, quindi un nome, alla *tile* specificata.

Una copia del file `.piwall` e del file `.pitile`, corrispondenti al Raspberry Pi in considerazione, andrà poi inviata nella cartella `home` dell'utente che eseguirà il processo relativo al sistema PiWall.

Siccome ad un utente finale sarebbero potuti risultare difficili la creazione di questi file di configurazione (comprensivi del calcolo delle dimensioni collettive e della compensazione dei bordi) e la copia di essi nei giusti percorsi, si è scelto di creare una semplice *utility* che lo guidasse durante questi passi. Le parti di codice salienti sono disponibili all'appendice C.1. Questo software, propone, gestendo in maniera consistente gli errori di input una serie di domande all'utente:

- la dimensione in numero di monitor del videowall;
- la larghezza e l'altezza in pixel di ogni monitor;
- la larghezza interna di ogni monitor in millimetri;
- la larghezza della cornice in millimetri.

Queste informazioni sono sufficienti per calcolare in automatico le dimensioni delle cornici in pixel e quindi affrontare il calcolo per la *bezel compensation*. La formula usata è:

$$Bezel_{pixel} = \frac{Monitor\_Width_{pixel}}{Monitor\_Width_{millimeters}} \times Bezel_{millimeters}$$

Gli stessi dati gli permettono di generare correttamente tutti i file di configurazione. Prevede anche le opzioni per caricare in automatico, tramite il protocollo SSH[10],

[10]SSH (*S*ecure *Sh*ell) è un protocollo di invio comandi remoti; permette anche il trasferimento remoto di file, tramite il protocollo SCP (*S*ecure *Co*py) che è quello utilizzato in questo progetto.

sia i file che riguardano il servizio di PiWall, sia i file di configurazione (`.piwall` e `.pitile`).

3.4 Strumenti utilizzati

In corso di realizzazione del progetto, sono stati utilizzati diversi strumenti, che possono essere considerati al pari di un hardware di riferimento; inoltre sono sempre state tenute in considerazione le idee esposte precedentemente, ovvero d'impiegare, ove fattibile, dell'hardware aperto, anche nella sue declinazioni più specifiche.

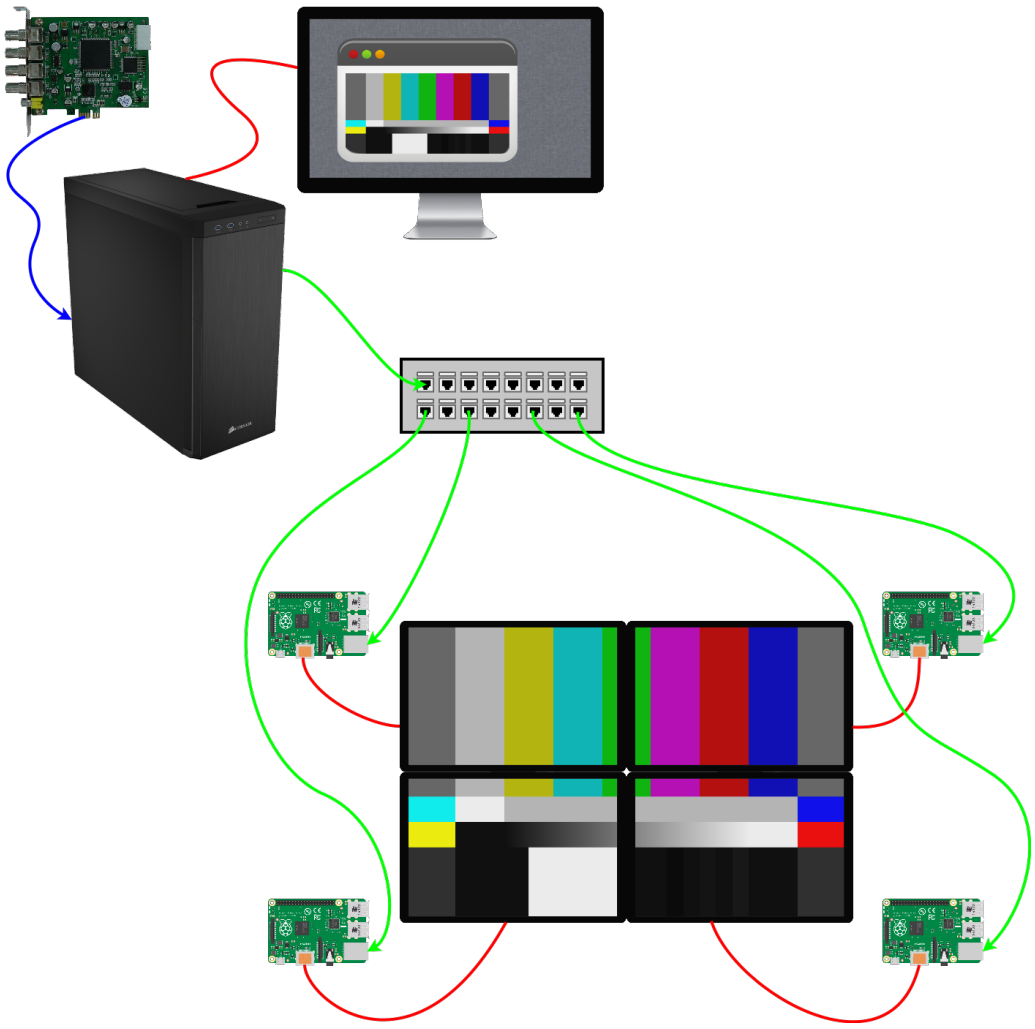

Figura 10: Schema del flusso di dati nel progetto.

In figura 10 (a pagina 50), si possono osservare i principali componenti del sistema utilizzati nel modello di riferimento. Nello specifico, essi sono:

- scheda di acquisizione di immagini digitali;
- elaboratore adibito a server, comprensivo di monitor;
- switch di rete;
- matrice di $m \times n$ monitor, adibita a videowall;
- un elaboratore per ogni monitor, facente parte del client distribuito.

La scheda di *capture* di immagini digitali utilizzata durante lo sviluppo è la Vision RGB-E1S, prodotta dalla Datapath Limited. Purtroppo in questo frangente non è stato possibile far ricadere la scelta su un prodotto che rispondesse ai criteri di *open hardware*, poiché attualmente non ne sono presenti sul mercato, a causa dell'alta specificità del compito che vanno a svolgere. Per sopperire a tale mancanza, ci si è limitati a sceglierne una che avesse il supporto per il kernel Linux. Questa limitazione può comunque essere aggirata tenendo conto che il *framework* Video4Linux gestisce molte periferiche, tra cui le webcam allo stesso modo di una scheda di acquisizione.

Questa scheda viene montata su porta PCI Express x4, con un *data rate*di 650 Mb/s e una memoria video di 32 MB. Supporta la cattura su un singolo canale con interfaccia DVI-I digitale ed una risoluzione massima in entrata di 1920×1080 con profondità di colore di 24 *bit*, quindi 8 *bit* per canale. Inoltre, è presente la possibilità, largamente utilizzata in questo progetto, di comprimere a livello hardware la componente cromatica, ottenendo così un'uscita nello spazio colore Y'UV con sottocampionamento della croma a 4:2:2.

Per quanto concerne il server, l'applicazione è stata sviluppata per essere compatibile con architetture di diverso tipo, ma tenendo presente come base quella Intel x86, nello specifico x86_64 (propria dei più performanti processori a 64 bit). La macchina utilizzata è stata dotata di Ubuntu, uno dei più comuni sistemi operativi basato sul kernel Linux. In fase di test, il progetto è stato installato e verificato con successo anche su macchine con processore CISC di tipo ARM. Ovviamente, com'è facile prevedere, l'applicazione sviluppata ha bisogno di un minimo di potenza di calcolo per

svolgere correttamente la sua funzione, perciò è auspicabile che tale potenza sia offerta in caso di esecuzione su microcomputer o elaboratori basati su un SoC (*S*ystem *o*n a *C*hip).

Come si potrà poi osservare nella sezione 4.1 dedicata all'analisi delle performance, non è trascurabile, per poter effettuare uno streaming di buona qualità, l'infrastruttura di rete utilizzata per il passaggio di dati tra il server e i diversi client. Durante la fase di sviluppo è stata utilizzata una rete locale di tipo LAN su cavo ethernet di categoria minima 5 che arrivasse al trasporto teorico di 100 Mb/s di dati. In questo caso un router non è stato necessario, infatti si è preferito usare un solo switch compatibile con la velocità precedentemente specificata.

La parte client è, al contrario delle componenti descritte sinora, distribuita: ad ogni monitor è associato un calcolatore. Non è un requisito stretto che i monitor siano identici oppure finemente comparabili, ma il software sviluppato per creare la configurazione del videowall mantiene comunque questa assunzione. Suddetta ipotesi è considerevole poiché il calcolo riguardante la *bezel compensation* diventerebbe, in caso contrario, assai più complesso. È alla stessa maniera un assunto importante, poiché nella misura opposta la correzione del colore effettuata in fase di visualizzazione (trattata nel capitolo 5) risulterebbe molto meno utile ed efficace.

In fase di sviluppo i monitor disponibili sono stati quattro DELL UltraSharp 2408WFP. Questi monitor montano schermi LCD a matrice attiva, dotati di supporto al 110% del gamut colore di riferimento NTSC (visibile nella figura 15 a pagina 66), con risoluzione massima 1920×1200 a circa 90 *ppi* fisici (si può fare riferimento alla figura 9 a pagina 48 che raffigura i quattro esemplari usati durante lo sviluppo).

Come già accennato nei capitoli precedenti, per quanto riguarda i calcolatori della parte client sono stati scelti dei Raspberry Pi, sia in quanto rispondono, nella loro quasi interezza, ai criteri di *open hardware*, che per la facile reperibilità sul mercato e il basso prezzo di vendita. Inoltre, la facoltà di avere disponibile un sistema GNU/Linux completo, garantisce la possibilità di impostare un profilo colore per l'adeguamento cromatico.

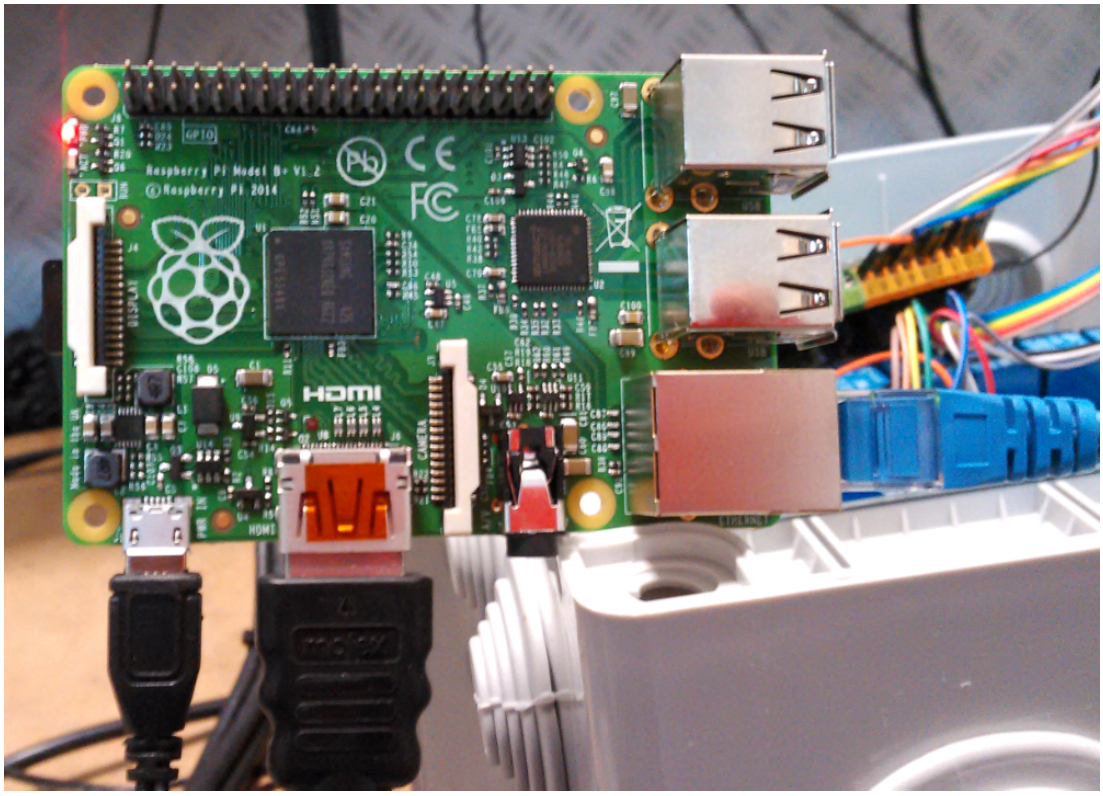

Figura 11: Un Raspberry Pi collegato al videowall utilizzato durante lo sviluppo.

Nonostante la complessivamente bassa capacità di calcolo e grazie al progetto PiWall, questo microcomputer si è comunque dimostrato all'altezza del compito assegnatogli. Per lo sviluppo è stato installato sui Raspberry Pi usati, impiegando una microSD card di classe abbastanza elevata da garantire uno scambio dati adeguato all'utilizzo nel progetto, la distribuzione GNU/Linux Raspbian[11] con la configurazione di default.

L'unico svantaggio, riscontrato nella fase di verifica del progetto, nell'uso di Raspberry Pi come elaboratore per il client è l'assenza di una periferica di rete che

[11]Raspbian è una distribuzione GNU/Linux basata su Debian, un'altra distribuzione molto famosa, ottimizzata per l'architettura ARMv6 presente sul Raspberry Pi; nello specifico sfrutta il coprocessore per il calcolo in virgola mobile, strumento chiave quando si ha la necessità di maneggiare dei flusso video.

sia in grado di supportare velocità superiori ai 100 $Mbit/s$, limitando di fatto il numero di flussi di streaming contemporanei possibili sulla rete. L'inconveniente però è giustificato dal basso costo di acquisto per unità di calcolo e sarebbe risolvibile semplicemente utilizzando dei microcomputer simili, ma provvisti di un modulo che raggiunge tale velocità (ad esempio Banana Pi) oppure utilizzando un modulo gigabit ethernet esterno con interfaccia USB.

Capitolo 4

Risoluzione delle criticità

Durante lo svolgimento del progetto, sono stati incontrati diversi punti critici, che sono stati via, via risolti effettuando delle scelte cooperanti con lo stile di sviluppo pensato e le idee descritte in fase introduttiva. Tali decisioni hanno dovuto tenere di volta in volta conto degli eventuali *trade-off*, in quanto nella maggior parte la criticità era nella scelta di due casi differenti, per i quali la scelta di uno andava a inficiare l'efficienza dell'altro e viceversa. Nella maggioranza delle occasioni è stata preferita la soluzione che avrebbe trovato un successivo riscontro nel momento dell'uso pratico del software.

Un primo problema è stato incontrato in seguito alla definizione dell'ordine delle azioni intraprese all'interno della libreria `libv4lcapture` (descritte nell'immagine 12 (a pagina 56).

Come già sostenuto in precedenza, le diverse procedure descritte descritte nell'immagine risultano piuttosto onerose da eseguire per il processore; tenendo conto che un punto chiave del progetto è lo streaming in tempo reale, la latenza tra la cattura di un frame ed il suo conseguente invio in rete è un fattore decisivo. Sin da subito è nata l'esigenza di sincronizzare temporalmente nel miglior modo possibile l'istante di tempo della cattura di un frame con quello in cui il frame subisce la compressione.

Era necessario rimuovere della complessità computazionale al fine di coadiuvare il lavoro dell'elaboratore e ridurre il ritardo causato dal calcolo. Dopo diverse verifiche empiriche, è stato stabilito che, utilizzando l'elaboratore corrente, un output compresso di massimo 30 fps sarebbe bastato a garantire una latenza minima. È stato deciso

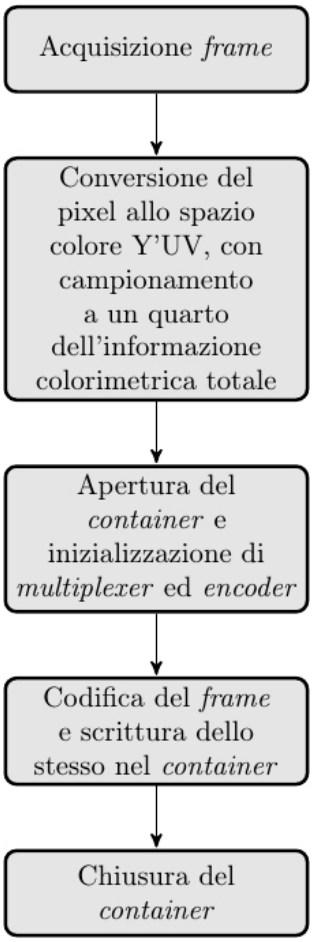

Figura 12: Schema di flusso delle operazioni compiute in `libv4lcapture`.

di limitare il *framerate* di uscita del flusso di dati in rete a tale velocità, scartando, quando e solo se necessario, i frame superflui direttamente all'atto della cattura, per non oberare comunque il processore con il calcolo della conversione di formato pixel.

Una necessità, riguardante sempre siffatta libreria `libv4lcapture` e nota già all'inizio della scrittura dell'applicazione, era quella di poter avviare più sessioni del processo server, e poter accedere in più utenti alla stessa scheda o a schede diverse, in maniera tale da generare più flussi di streaming contemporanei e renderli disponibili su coppie indirizzo IP-porta diverse. Questa funzionalità può essere utile nella pratica, ad esempio quando esistono due o più sorgenti differenti in input sullo stesso elaboratore e si desidera pilotarle tutte dalla stessa postazione di controllo.

In questo caso il punto cruciale è quello della quantità di banda di trasferimento disponibile in fase di cattura. È necessario considerare che a causa dell'astrazione operata dal *framework* Video4Linux, qualsiasi *device* capace di fornire in input immagini digitali può essere impiegato come periferica di cattura. Questo fatto introduce la possibilità che tali periferiche utilizzino *bus* di collegamento non particolarmente performanti o adatti alla situazione. Ad esempio una comune webcam utilizza l'interfaccia di comunicazione seriale USB 2.0 mentre una comune scheda di acquisizione si server dell'interfaccia PCI Express. L'USB 2.0 raggiunge velocità teoriche di 480 Mb/s, che non sono nemmeno lontanamente comparabili ad esempio a quelle della scheda PCI Express usata in fase di sviluppo, che totalizza circa 650 MB/s, quindi circa 5.2 Gb/s.

Per ovviare alla scarsità di banda nel caso si debba lavorare con periferiche cosiddette "a trasferimento lento[1]", si è quindi pensato di mantenere il *framerate* in output pari a quello in input. Ad esempio una webcam di scarsa qualità permette la cattura di frame a 15 fps; in questo caso il flusso in streaming risultante avrebbe comunque mantenuto lo stesso *bitrate* di 15 fps. Ovviamente le limitazioni introdotte precedente di 30 fps rimangono comunque applicate.

Se viene considerato lo sviluppo del front-end `V4LCapture` è stato necessario trovare una soluzione alla fondamentale criticità da cui sarebbe completamente dipesa la riuscita di tutto il progetto: il passaggio di dati tra il *core* dell'applicazione e la relativa interfaccia grafica.

Ricapitolando quanto già descritto, le due parti, *core* di calcolo e interfaccia grafica, sono state progettate per essere indipendenti l'una dall'altra, ma allo stesso tempo necessitano di scambiarsi molto spesso informazioni, infatti una richiede all'altra le impostazioni per inizializzare la cattura e lo streaming e l'altra restituisce alla prima le immagini catturate. Come già spiegato, dal punto di vista programmatico, questo si è tradotto in una API pubblica, definita nel *core* e resa disponibile ai *layer* successivi del progetto, siano essi delle interfacce grafiche (come in questo caso) o completamente altri tipi di applicazioni.

[1]Un device viene considerato "lento", se non riesce a inviare più di 30 fps.

È stato necessario studiare un metodo che avrebbe reso fruibile soprattutto la libreria sottostante, indipendentemente dal linguaggio di programmazione in cui l'utilizzatore fosse stato implementato o dall'architettura usata. La risposta a questo problema è stata trovata in questo modo: i messaggi scambiati tra back-end e front-end sono sempre incapsulati in variabili di tipo primitivo, quali interi, stringhe oppure valori booleani; questi tipi di variabili sono comuni a tutti i linguaggi di programmazione e a tutte le architetture.

L'unica procedura che fa eccezione è quella adibita al passaggio dei dati al fine di visualizzare la schermata di preview, inclusa nella GUI, del flusso di dati che sarà la sorgente dello streaming; per sua natura, quest'ultima procedura, deve restituire un array, o qualche sorta di buffer che possa contenere valori interi con precisione pari a 8 bit[2]. La struttura restituita dalla routine in questo caso è una sequenza variabile che mantiene l'ordine `(R, G, B)` indipendentemente dall'architettura in uso.

Nel caso specifico del progetto, il *core* è collegato ad un front-end di tipo grafico scritto nel linguaggio Python, perciò un linguaggio sostanzialmente diverso dal C. Per collegarli si è scelto di utilizzare un modulo già contenuto all'interno della distribuzione standard di Python, chiamato `ctypes`. Suddetto modulo rende fattibile la chiamata diretta di funzioni scritte nel linguaggio C, importandole da una *shared library*, una libreria condivisa, e scrivendo i nomi di tali funzioni direttamente nel *namespace* dall'ambiente di Python; le procedure della libreria condivisa risultano perciò chiamabili al pari di funzioni native scritte in Python.

Per l'interfaccia grafica presentata nel progetto è stato scelta la versione 3 del linguaggio Python in abbinamento alle librerie grafiche Qt5, per mezzo del progetto PyQt5; tale decisione ha una motivazione pratica: sia Python che il *framework* Qt sono studiati e sviluppati per essere multipiattaforma[3] e condividono l'idea di uno sviluppo rapido, non inficiato da particolari poco rilevanti. In aggiunta sia Python

[2]Le quantità intere a 8 bit a cui si fa riferimento sono triplette di valori che rappresentano i canali R, G e B, che, come noto, spaziano da 0 a 255.

[3]Python si avvale di una *virtual machine* disponibile per numerose piattaforme; in aggiunta una copia dell'interprete è spesso già inclusa nelle distribuzioni GNU/Linux più diffuse. Qt è scritto nel linguaggio C++, ma si integra perfettamente con i *toolkit* grafici nativi di molti sistemi operativi.

che Qt5 sono due progetti il cui scopo principale è sì la facilità d'uso, ma soprattutto quella di estensione; questa è una motivazione valida per tutti i principi discussi all'inizio della trattazione. Il *framework* Qt5 presenta inoltre una funzionalità molto utile: si integra indipendentemente e trasparentemente con le librerie grafiche nativa di sistema, facilitando nel complesso l'esperienza utente e migliorando il design dell'interfaccia stessa.

Procedendo in sequenza, dal cuore del progetto verso la periferia, un altro punto critico da prendere in considerazione è stato, per l'appunto, quello dell'usabilità dell'interfaccia grafica sviluppata per l'utente finale. Il criterio stabilito all'inizio della fase di progettazione non prevede che l'utente *target* abbia una particolare conoscenza tecnica dei meccanismi di *capture* e streaming; per questo motivo è stato fondamentale assicurarsi che le azioni selezionabili nei vari menu che riguardanti le impostazioni avvessero una reazione istantanea alla pressione dell'apposito pulsante di conferma delle modifiche scelte. Quanto appena detto è applicabile sia per la vista delle impostazioni che per quella dell'anteprima, la quale necessita di essere aggiornata ogni volta che il *core* fornisce un nuovo frame catturato.

Questo risultato è stato ottenuto sfruttando un espediente tecnico: la creazione di un thread dedicato solo alla cattura immagini (descritto alla sezione 3.3.2). La creazione di un'unità di calcolo separata che gestisce esclusivamente la comunicazione con il back-end permette non solo di far rimanere performante l'interfaccia grafica liberandola dall'*overhead* del compito appena descritto, ma anche di dividere completamente dal resto del codice la sequenza di passi da eseguire per inizializzare correttamente lo streaming. È stato quindi creato un modulo che, cooperando con un gestore delle impostazioni incluso, anch'esso descritto estensivamente, permette di avviare, riavviare e fermare la fase di streaming, caricando le nuove impostazioni e restituendo così all'utente un *feedback* dei cambiamenti.

Un ulteriore punto di interesse si colloca alla periferia del progetto, più precisamente riguarda il client: la correzione del colore in fase di visualizzazione. La definizione data di "correzione del colore" è corretta, ma imprecisa; in realtà l'operazione che bisogna in ultima analisi effettuare è una calibrazione o, ancora meglio, un adeguamento cromatico tra i diversi dispositivi di uscita facenti parte del *player* distribuito.

L'operazione sarebbe molto meno utile nel caso in cui il client non fosse distribuito ma la visualizzazione avvenisse su un unico monitor. Infatti le disuguglianze cromatiche e di contrasto percepite tra il flusso in input e quello in output diventerebbero rilevanti solo nel situazione in cui la visualizzazione venisse operata da due monitor del client distribuito accostati.

Inizialmente, si era pensato di introdurre una correzione del colore a monte, subito dopo la cattura dei frame. Una volta superati gli ostacoli tecnici (riguardanti soprattutto il tempo impiegato per svolgere gli algoritmi di aggiustamento cromatico), ci si è ritrovati con una soluzione poco funzionale. Il gamut[4] in uscita, leggermente diverso tra un monitor e l'altro, ha reso necessaria l'implementazione di una correzione a valle, quindi effettuata poco prima dell'effettiva visualizzazione da parte dei riproduttori.

Sebbene qui venga accennato solo per sommi capi, l'intero processo non è automatico e merita una trattazione più approfondita e particolareggiata. L'intero capitolo 5 descrive la procedura effettuata per ottenere una adeguamento cromatico accettabile e diminuire, per quanto possibile, le differenze tra un *player* e l'altro.

4.1 Analisi prestazionale

Sia in fase di verifica che nel contesto aziendale in cui l'applicazione va a inserirsi è importante avere un'attestazione dell'efficienza che la contraddistingue.

L'efficienza di applicazioni di tipo *network videowall* si valuta durante la fase di streaming, sotto forma di occupazione della banda disponibile in rete. Tale indicatore di prestazione serve ad uno scopo ben preciso: spesso è necessario trasmettere sulla rete più di un flusso video. Questi stream vengono catturati da diverse schede che possono sia essere montate sullo stesso elaboratore sia essere componenti di calcolatori differenti. In entrambi i casi l'occupazione di banda scala linearmente con l'aumento dei flussi presenti in rete.

[4]Il gamut di un dispositivo di visualizzazione è l'insieme di colori che tale dispositivo è in grado di riprodurre; questo insieme generalmente è un sottoinsieme molto limitato dello spettro della luce visibile. Solitamente il gamut cerca di conformarsi ad un modello di colore, ma in quanto parte di un'approssimazione, questo non è garantito.

Prendendo in esame la rete costruita durante lo sviluppo e utilizzata nel corso della verifica, si ha una velocità teorica massima di 100 Mb/s. Le reti più recenti nate allo scopo di scambiare contenuti di tipo video raggiungono velocità teoriche anche dieci volte superiori, perciò di 1 Gb/s. Ciò significa che se il flusso prodotto da un'applicazione che scambia dati in rete è di, ad esempio, 5 Mb/s, con una rete 100 megabit si raggiunge un valore di capienza a pieno regime di 20 flussi contemporanei; con una rete gigabit invece la situazione migliora notevolmente, in quanto i flussi contemporanei trasportabili sono 200.

A questo scopo è stato ideato un test per valutare le performance, tradotte nell'occupazione di banda, dell'applicazione creata, lasciando i parametri di default di cattura e compressione, che sono:

- dimensione: 1920×1080;
- *framerate*: 60 fps;
- CRF: 28;
- *container*: avi;
- *preset*: ultrafast;
- *tune*: zerolatency.

Il sistema costruito per effettuare quest'analisi delle performance era così formato: l'applicativo server è stato installato su una macchina che e è stata poi collegata ad uno switch con velocità massima di 100 Mb/s. Allo stesso switch sono stati collegati quattro Raspberry Pi come client, corrispondenti ai quattro monitor usati. Come monitor sono stati usati i quattro già descritti alla sezione 3.4. Lo switch era un particolare modello provvisto di una porta utile per l'analisi dei pacchetti in rete in modalità promiscua[5]. A questa porta è stato collegato un altro elaboratore provvisto

[5]La modalità promiscua è una particolare modalità della scheda di rete usata per l'analisi, nella quale l'elaboratore non analizza solo il traffico che invia o riceve, ma tutto il traffico complessivo all'interno della rete.

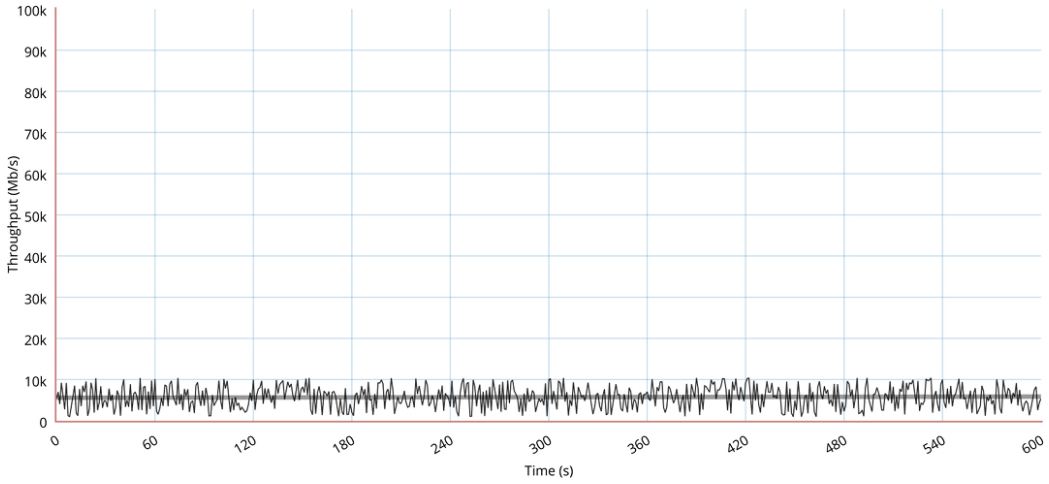

Figura 13: Grafico dei dati ottenuti durante l'analisi delle prestazioni.

di "Wireshark", un applicativo *open source* per analisi del traffico di rete, che ha provveduto a raccogliere informazioni per valutare l'occupazione di banda.

Come sorgente si è deciso di utilizzare un terzo computer (collegato alla scheda di cattura) che mostrasse un video con contenuti e scene abbastanza differenti tra di loro (durante il test è stato scelto un documentario naturale) in modo da inibire al massimo l'azione del sistema di *motion detection*, che altrimenti, a causa della staticità, avrebbe permesso la produzione un flusso molto leggero. Il video è stato visualizzato a schermo intero, in modo da avere il solo flusso video inalterato da eventuali componenti grafici del sistema operativo. La simulazione di streaming ha avuto una durata di 10 minuti e Wireshark ha fornito dei dati che hanno risoluzione di 1 s tra di loro. Ne consegue che sono stati raccolti 600 campioni (mostrati in un grafico nella figura 13 a pagina 62).

Si può notare che l'occupazione massima di banda si è attestata intorno a $1,3$ MB/s, corrispondenti a 10.4 Mb/s, mentre quella minima è stata intorno ai 120 KB/s, pari a 960 Kb/s. L'occupazione media quindi si è stabilizzata intorno ai 710 KB/s, cioè circa 5.7 Mb/s. In una situazione di questo tipo, la rete 100 megabit utilizzata potrebbe supportare fino a 20 flussi in streaming contemporanei, che nella media degli strumenti che si occupano di questo ambito è un buon valore (soprattutto se si pensa che una rete gigabit i flussi contemporanei possibili sarebbero 200).

Sono stati effettuati altri due test in casi limite per capire quanto fosse ampia

l'effettiva possibilità di configurazione dell'occupazione di banda. Il primo test è stato eseguito con un valore CRF pari a 47, cioè avendo un'immagine risultante dalla compressione appena accettabile (sufficientemente da poterne distinguere il soggetto). Per il secondo, invece, ci si è attestati su un valore di 16, cioè la soglia massima, sopra la quale il risultato prodotto sarebbe stato il medesimo. Per quanto riguarda la qualità minima, il range di saturazione si è assestato su valori che vanno da 50 KB/s (400 Kb/s) a 220 KB/s (1.7 Mb/s), mentre nell'altro caso il range è partito da 1.5 MB/s (12 Mb/s) e è arrivato fino a 2.1 MB/s (16.8 Mb/s).

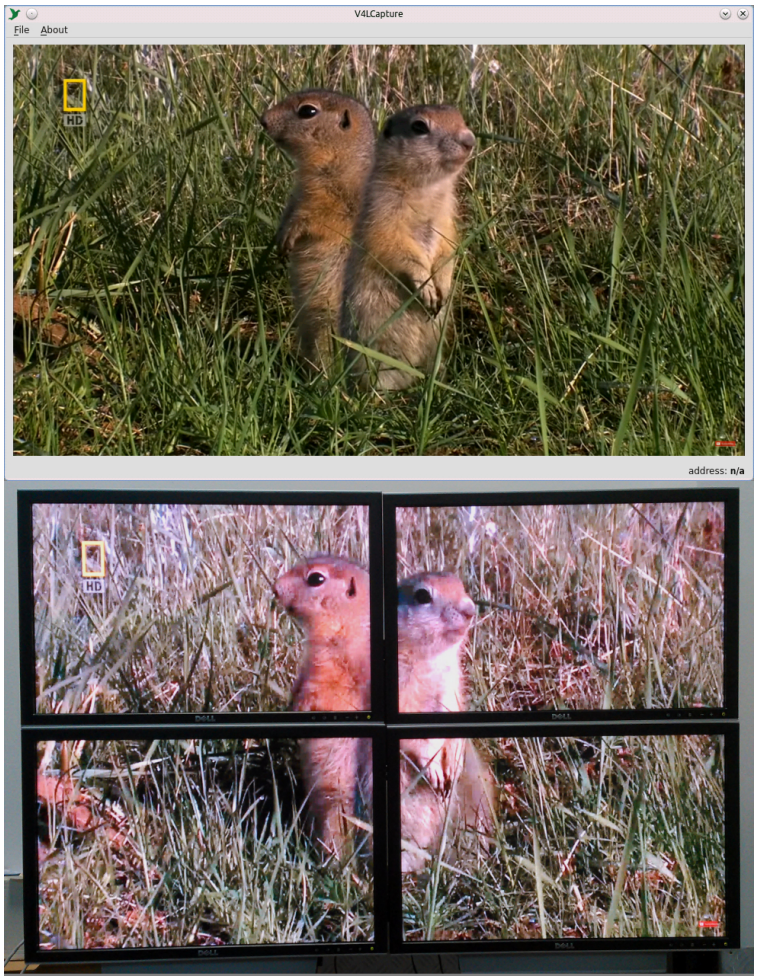

Figura 14: Paragone tra il video in input e quello mostrato sul videowall (CRF 28).

La qualità della compressione, nel complesso, è risultata molto buona, soprattutto se confrontata con il *bitrate* prodotto (specialmente nel caso con un valore CRF minimo di 47) che è risultato tra i più bassi nel panorama delle applicazioni di questo tipo (l'immagine in figura 14 a pagina 63 fornisce un chiaro esempio). Bisogna, però, considerare che la risoluzione di cattura era 1920×1080 e quella di un singolo monitor 1920×1200, perciò la risoluzione totale del videowall era di 3840×2400, e quindi il sistema PiWall per riempire l'intera superficie di visualizzazione è stato costretto a compiere un'operazione di *upscaling* sulle dimensioni del flusso video.

Da tenere presente è che l'applicazione utilizza uno streaming verso indirizzi IP di tipo multicast. Questo fa sì che il flusso venga inviato su uno specifico IP senza che ci sia bisogno di un client che lo richiede. La particolarità, però, è che il flusso non viene realmente inviato (e perciò non occupa la rete), fintantoché non risulta realmente usufruito da un utilizzatore. Questo può portare ad affermare che se durante il test fossero stati scollegati i quattro elaboratori, il server avrebbe continuato a creare il flusso, ma non a inviarlo sulla rete, lasciandola con un'occupazione di banda nulla.

Capitolo 5

Calibrazione del colore

Riprendendo da quanto accennato prima, in questo capitolo viene spiegato nel dettaglio il processo di adeguamento del colore su ogni monitor, associato ad un elaboratore, facente parte del client distribuito.

Se prendiamo in esame il caso in cui il client distribuito sia un videowall in cui ogni singola cella della matrice di cui è composto sia parte di una superficie di visualizzazione più ampia, questa operazione è necessaria. I monitor usati sono, come già descritto, periferiche professionali e tecnicamente di buona fattura, ad esempio forniscono il supporto al 110% del gamut di riferimento NTSC[1]; bisogna tenere sempre presente, però, che l'insieme di colori resi nella fase di visualizzazione rimane comunque un insieme di onde e, in quanto tale, è difficile produrre due dispositivi che emettano la stessa, identica onda (ad esempio, potrebbe cambiare leggermente il coefficiente di riflessione[2] della superficie di visualizzazione del monitor). Per questo motivo, specialmente in una configurazione videowall dove i monitor vengono accostati, l'operazione di correzione del colore è fondamentale.

Nell'idea prevista inizialmente, come già accennato, il processo di correzione era

[1]Lo spazio colore NTSC è stato introdotto nel 1953 dalla FCC (*F*ederal *C*ommunications *C*ommission), come parte dell'omonimo standard televisivo in uso all'epoca.

[2]Il coefficiente di riflessione, o riflettività spettrale, è il rapporto tra l'intensità della radiazione riflessa e quella della radiazione incidente.

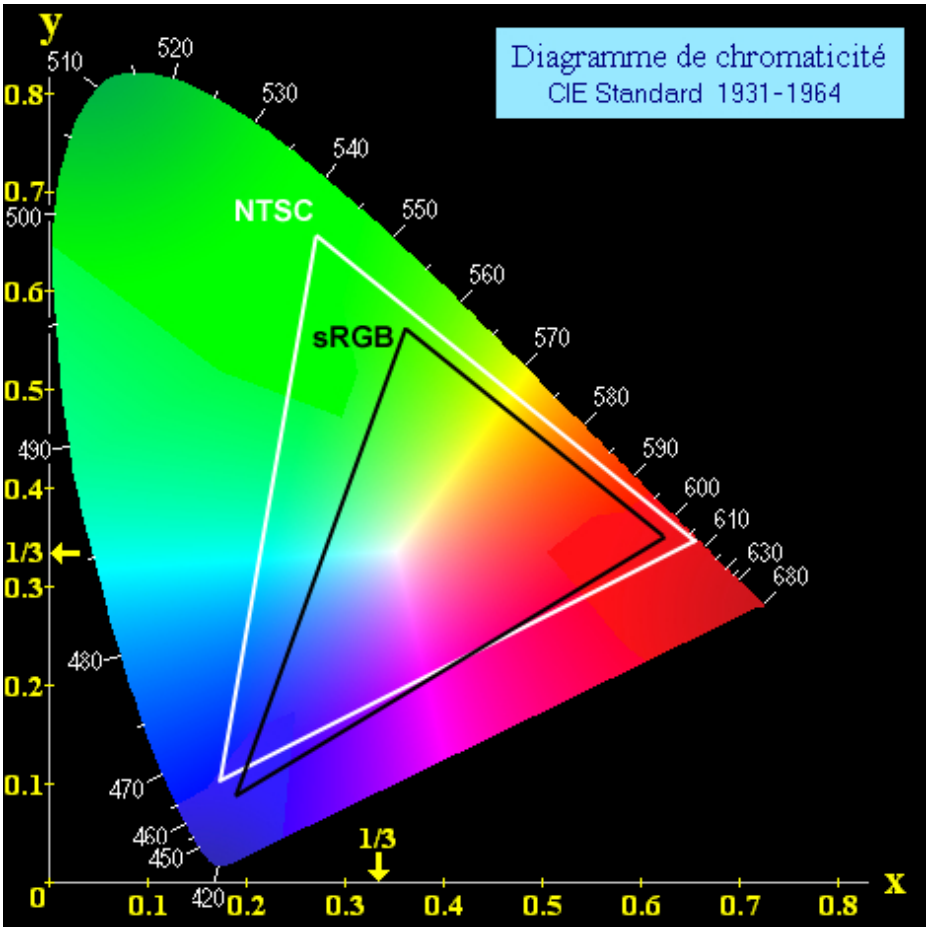

Figura 15: Rappresentazione dello spazio colore NTSC confrontato con sRGB (da Wikipedia).

posto a monte della sequenza di streaming, cioè veniva eseguito in successione alla cattura dell'immagine, ma prima della compressione del flusso. Sono stati riscontrati tre problemi sostanziali, a cui questa configurazione non ha saputo trovare una soluzione valida.

Il primo riguarda intrinsecamente il modo in cui il processo di correzione del colore veniva messo in atto. In una prima versione dell'applicazione, era presente una terza scheda apposita demandata alla correzione cromatica inglobata nella vista delle impostazioni dell'interfaccia grafica. In questa sezione, tramite degli *slider* e delle *checkbox*, veniva chiesto all'utente finale di modificare leggermente luminosità,

contrasto, gamma, bilanciamento del bianco, del nero e dei tre colori primari; in aggiunta era possibile avviare un processo di equalizzazione adattiva del contrasto, tramite la tecnica di CLAHE[3].

Tanto questo poteva sembrare un vantaggio sotto il punto di vista della personalizzazione, quanto, in realtà, si sarebbe trasformato in una situazione critica. Questo poiché la correzione in atto era demandata ad un utente e quindi era una correzione totalmente soggettiva, dipendente fortemente da tale utente. La correzione non sarebbe, perciò, rimasta più tale, ma si sarebbe trasformata in un vero e proprio viraggio del colore. Il processo risultante quindi non si sarebbe più potuto chiamare *color correction*, ma sarebbe passato sotto la definizione di *color grading*.

Il secondo problema era invece situato nella scelta della posizione, nella sequenza di azioni compiute dal server, della correzione del colore. Nella configurazione descritta precedentemente, essa veniva introdotta a seguito della cattura del frame, ma prima sia della compressione del flusso di dati che dell'avvio dello streaming. Dato che il flusso inviato in rete si serve del formato H.264, la sua componente colore, chiamata chroma, è sottocampionata a un quarto dell'informazione originale per risultare conforme agli standard. In più H.264 è è anche un formato compresso e tale compressione ha la possibilità di modificare ancor di più il colore finale.

Tenendo conto di tutti questi fattori, non era perciò garantito che l'adeguamento effettuato in coda alla cattura, ma pur sempre prima della compressione delle immagini, non venisse inficiato tali fattori.

Il terzo punto critico è quello di maggiore considerazione in quanto con questa configurazione non è risultato risolvibile. Se la correzione fosse stata effettuata a monte della fase di streaming, ogni monitor avrebbe ricevuto lo stesso segnale video, senza la possibilità di agirvi ulteriormente, ad esempio operando una correzione personalizzata.

Come già accennato prima, le lievi differenze di gamut tra un monitor e l'altro, pur

[3]CLAHE (*C*ontrast *L*imited *A*daptive *H*istogram *E*qualization) è una tecnica di correzione adattiva del contrasto; trae le sue origini da AHE (Adaptive Histogram Equalization), nella quale ogni pixel diventa il risultato della funzione di trasformazione derivata dal suo vicinato, con una funzione di trasformazione proporzionale alla CDF (*Cumulative Distribution Function*).

usando monitor di buona fattura dello stesso modello e stesso produttore, risultavano sempre sufficientemente ampie da generare una rivalità cromatica, nella situazione in cui i suddetti monitor venissero accostati. Questa tesi è stata poi confermata in maniera più rigorosa e definitiva durante la fase di correzione a valle, grazie all'uso di un colorimetro.

Prendendo atto di questi tre fattori decisivi, si è scelto di introdurre un'adeguamento che operasse a monte del sistema e che fosse capace di generare una configurazione personalizzata per ogni monitor; per questo motivo la correzione effettuata in questa fase ricadrà nella definizione di correzione adattiva.

Per aggiungere al sistema un'adeguamento cromatico a valle di tutto il processo di streaming, la scelta obbligata è stata quella di progettare qualcosa che venisse eseguito in maniera indipendente per ogni ramo del client distribuito. Alla fine, si è deciso di operare una correzione colore dell'intero sistema operativo degli elaboratori impiegati nel client. Sarebbe servito uno strumento specifico che interpretasse i colori mostrati in output sullo schermo, cioè un colorimetro o un spettrometro. Tale strumento ad ogni avvio del sistema si sarebbe dovuto preoccupare di generare e applicare un profilo di calibrazione su ogni elaboratore facente parte del client distribuito.

Tra le varie soluzioni pensate, una è stata quella di costruire, tramite la piattaforma *open hardware* di sviluppo Arduino, uno strumento specifico per il caso. Purtroppo, la realizzazione di un tale strumento in maniera prototipale non è infine risultata attuabile, in quanto la sua costruzione e verifica avrebbe richiesto particolari competenze ingegneristiche non raggiungibili con i tempi e gli strumenti tecnici a disposizione.

Per poter avere comunque a disposizione una tale tecnologia, ci si è perciò dovuti affidare ad uno strumento che fosse già presente sul mercato; era importante che questo fosse di buona qualità e permettesse la facile generazione di un profilo colore.

Restando fedeli ai principi dell'*open hardware* espressi precedentemente, si è scelto

di usare un ColorHug[4], un colorimetro digitale, sviluppato comunque sulla piattaforma Arduino, che, tra i molteplici vantaggi, è risultato di gran lunga più economico rispetto alla maggior parte dei *competitor* non liberi. ColorHug è un progetto completamente libero, compresa la sua documentazione; in un ipotetico futuro, sarebbe possibile, se necessario, migliorarlo e integrarlo completamente negli elaboratori che formano il client.

ColorHug si posiziona sotto la categoria dei colorimetri tristimolo, ovvero capaci di rilevare i colori alla base della teoria tricromatica di Young-Helmholtz, rosso, verde e violetto (blu), RGB. Esso contiene una matrice di 64 fotorivelatori divisi in gruppi: ce ne sono 16 rossi, 16 blu e 16 verdi più altri 16 di colore variabile; restituiscono triplette di valori interpretabili all'interno dello spazio colore CIE XYZ (in figura 16 a pagina 69 è possibile osservare la risposta in frequenza del colorimetro).

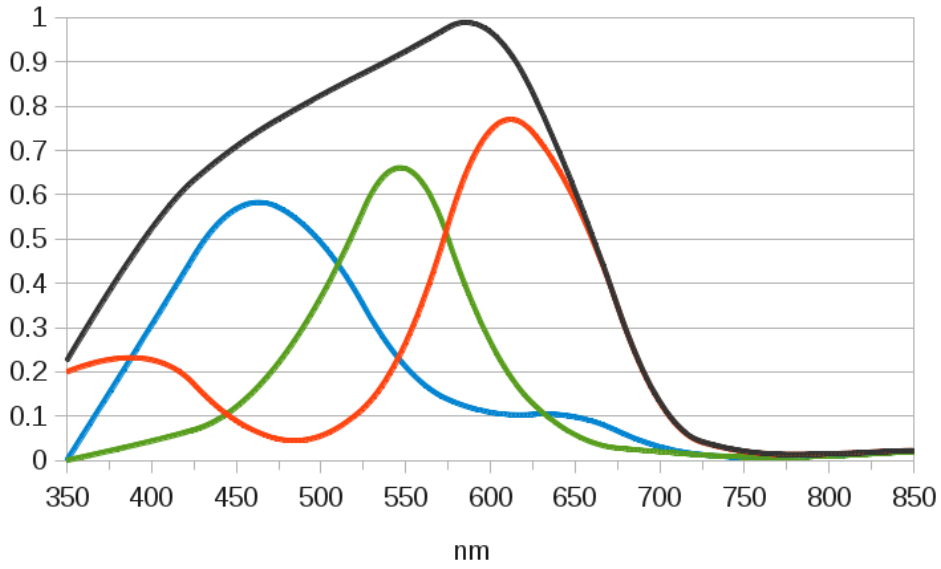

Figura 16: Risposta in frequenza del Color Hug normalizzata a 575 nm (da hughski.com).

[4]Durante lo sviluppo del progetto, è stato impiegato un ColorHug 2, evoluzione più precisa della prima versione.

Tramite ColorHug si è stati in grado di creare un profilo di colore, agnostico rispetto al sistema operativo e all'elaboratore utilizzato (e quindi valido anche per utilizzi futuri) che seguisse lo standard ICC[5] versione 4.3.

Un profilo ICC serve principalmente per la conversione di uno spazio colore in un altro di diversa natura. Contiene informazioni sullo spazio colore a cui si riferisce insieme ad dati per il passaggio ad uno spazio colore di transizione. I dati vengono forniti sotto forma di matrici di trasformazione, che definiscono le regole per la conversione dallo spazio colore descritto ad uno spazio colore di "connessione", detto PCS (*P*rofile *C*onnection *S*pace), che, secondo le specifiche, è di un tipo simile ai modelli CIE L*a*b* oppure CIE XYZ[6]. Succitati dati di conversione possono anche essere forniti sotto forma di LUT (*L*ook *U*p *T*ables) che funzionano in questo modo: per ogni tripletta dello spazio colore di provenienza, forniscono la corrispondente tripletta nel PCS. Le matrici di trasformazione vengono preferiti per i profili di tipo input, mentre le *look up tables* si usano solitamente per quelli di tipo output.

Avendo usato un colorimetro tristimolo, la calibrazione è avvenuta sfruttando i meccanismi alla base della sintesi additiva. Per coadiuvare le operazioni di generazione del profilo ICC, è stata utilizzata la suite di software messa a disposizione dal progetto "Argyll CMS" e la sua interfaccia grafica "dispcalGUI". Questo *tool* guida nell'impostazione dei settaggi del profilo colore che si vuole generare.

Si è scelto di generare un profilo colore che rispondesse alle seguenti caratteristiche: un punto di bianco corrispondente a 6500° K, una luminanza pari a 120 nit[7] e un valore di gamma pari a 2.2. Siccome la periferica oggetto della calibrazione era un monitor si è scelto di procedere con un profilo matriciale, saltando la fase di calcolo delle tabelle chiave-valore.

La procedura effettiva di calibrazione si è svolta subito dopo aver posizionato il colorimetro al centro dello schermo da calibrare. Successivamente dispcalGUI ha

[5]ICC sta per *I*nternational *C*olor *C*onsortium, un'unione stabile di diverse aziende per creare un sistema di gestione del colore aperto e *platform independent*. Lo standard ICC è definito minuziosamente, ma non vengono specificati gli algoritmi o i dettagli di calcolo.

[6]Entrambi gli spazi colore, CIE XYZ e CIE L*a*b*, vengono proposti con illuminante di riferimento D50, perciò con punto di bianco posto alle coordinate XYZ=(0.9642, 1.000, 0.8249).

[7]Il *nit* è un'unità di misura del Sistema Internazionale e 1 *nit* corrisponde ad 1 cd/m^2.

visualizzato delle *patch*, cioè delle zone uniformi di colore, al centro dello schermo che permettevano di illuminare di volta, in volta i subpixel rossi, verdi e blu. In successione, sono poi state mostrate altre serie di *patch* ognuna differente dall'altra. Ciò ha permesso al colorimetro di esaminare la resa del monitor in diverse condizioni e per diversi colori, calcolando la discrepanza di gamma e di luminosità. I valori ottenuti sono poi stati confrontati con quelli di riferimento dello spazio colore CIE XYZ e si sono potute calcolare le matrici di trasformazione.

Il calcolo è risultato piuttosto preciso, soprattutto grazie ai sensori presenti sul Color Hug, che hanno reso possibile eseguire i calcoli su valori nativamente interpretabili direttamente nel modello CIE XYZ. Il risultato di tutta la procedura è stato, alla fine, scritto nel profilo ICC, generando quindi un file `.icc`.

Il profilo sotto forma di file è stato applicato per mezzo di "Xcalib", un programma di utilità appositamente scritto e compatibile con il protocollo "X", implementato dal server grafico usato nel mondo GNU/Linux. È stato anche realizzato un servizio che ha permesso ad Xcalib di essere portato in funzione ad ogni avvio del sistema operativo presente sugli elaboratori

Per mezzo dell'intero procedimento appena descritto, è stato quindi possibile effettuare un adeguamento cromatico che non solo è differenziato per ogni monitor che fa parte del client distribuito, ma che è anche indipendente dalla coppia elaboratore-monitor. Se un nuovo monitor dovesse venire collegato ad un elaboratore del client, sarebbe sufficiente eseguire nuovamente la calibrazione su tale monitor per ottenere un nuovo profilo colore e rendere operativo un altro ramo del *player*.

Capitolo 6

Conclusioni e sviluppi futuri

Il progetto creato, considerando i ridotti tempi di sviluppo utilizzati principalmente per renderlo funzionante, attualmente ha raggiunto una prima versione usabile. Si è arrivati a soddisfare gli obiettivi che erano stati stabiliti inizialmente, cioè creare un'applicazione di cattura di immagini digitali e streaming su un videowall che:

- corrispondesse ad un sistema completamente incentrato sul software, che adottasse un approccio altamente modulare e fosse estensibile con nuove funzionalità;
- risultasse indipendente dl particolare modello di apparato hardware specifico usato, ma che ad esempio permettesse interoperabilità con un vasto di numero di periferiche di cattura, elaboratori e microcomputer grazie all'utilizzo di interfacce hardware tipicamente utilizzate nel campo (come PCI Express);
- rispettasse i criteri del software libero e dell'*open source* con possibilità di partecipazione anche per soggetti terzi non inclusi nello sviluppo originale;
- fosse compatibile con GNU/Linux e facilmente portabile ad altri sistemi operativi e piattaforme;
- facesse riferimento agli standard odierni in fatto di rendimento e di qualità sia per quanto riguarda la cattura delle immagini che per il flusso video prodotto in streaming, adottando tecnologie di uso comune in questo ambiente (ad esempio H.264);

- permettesse di scegliere l'esborso economico previsto in base all'ordine di qualità delle componenti utilizzate, sfruttando l'ampia gamma di prodotti impiegabili, e non a causa della poca interoperabilità con soluzioni di altre aziende.

Nonostante il raggiungimento di questo traguardi, alcune caratteristiche di contorno non sono ancora state completamente ultimate, lasciando spazio ad una considerevole evoluzione del progetto. Tale possibilità di un futuro ampliamento delle funzionalità è stata tuttavia tenuta in considerazione durante lo sviluppo e si è cercato di agevolarla attraverso la struttura modulare descritta nei capitoli precedenti.

Se consideriamo l'applicazione server, un requisito fondamentale per entrare nel mercato è la compatibilità con i sistemi più usati, come Microsoft Windows. Attualmente il software è compatibile esclusivamente con l'ambiente GNU/Linux. Le tecnologie usate durante la programmazione, quali la divisione in due atomi, uno di calcolo e l'altro di comunicazione con l'utente, l'uso di una API pubblica tra questi due elementi e l'adozione dei linguaggi C e Python, insieme alla libera condivisione del codice grazie alle licenze *free software*, rendono il processo di *porting* ad altre piattaforme particolarmente lineare.

Per poter effettivamente compiere quest'operazione, basterebbe riscrivere le funzioni del *back-end* di calcolo in modo che, al posto di usare delle funzionalità proprie del sistema GNU/Linux, usino quelle messe a disposizione dal sistema di arrivo. Nello specifico caso di Windows, suddette funzioni andrebbero a sfruttare i meccanismi di manipolazione di periferiche per il *video processing* proposti da VfW (*V*ideo *f*or *W*indows), al posto che quelli di V4L. L'interfaccia grafica, essendo scritta con linguaggi e *framework* già per loro natura multipiattaforma, non necessità di alcuna modifica.

Sempre nell'ambito del server, una *feature* interessante, per la quale a livello hardware sono disponibili dei moduli dedicati, è quella della possibilità di ricavare un input da un flusso proveniente in streaming dalla rete interna o da Internet. Per aggiungere questa funzionalità bisognerebbe predisporre un modulo all'interno del *core* di calcolo che si occupi di effettuare il *demultiplexing* e la decodifica del suddetto stream, per arrivare ad avere un flusso sotto forma di frame decompressi. Successivamente questo

flusso può seguire la procedura di ricompressione e tramissione in rete connessa col videowall (contenuta nel modulo già presente `encoding.c`).

Nel caso in cui il flusso proveniente dalla rete fosse conforme a quello prodotto in output dall'applicazione, una soluzione alternativa all'implementazione descritta prima sarebbe quella di commutare direttamente tale flusso nella rete connessa col videowall.

Altra serie di funzionalità facilmente realizzabili, ma che aumenterebbero la possibilità di personalizzazione del flusso video generato, sarebbe quella legata ad H.264. Attualmente gli aspetti personalizzabili del codec riguardano il *preset*, cioè il bilanciamento tra *throughput* dell'*encoder* e qualità del flusso prodotto, il *tune*, ossia la scelta della tipologia di flusso creato (per streaming, per test, per animazione, ecc.) e il CRF, cioè la soglia di qualità mantenere; altri aspetti che si potrebbero aggiungere a questa lista e quindi adattare a discrezione dell'utente sarebbero il GOP, la possibilità di codifica a *bitrate* costante e la dimensione della matrice su cui effettuare *motion prediction*.

Attualmente questi valori sono forniti all'*encoder* sotto forma di costanti. Per poterli rendere modificabili dall'utente finale bisognerebbe aggiungerli al contesto di codifica e creare un'ulteriore voce all'interno dell'interfaccia grafica, nella vista delle impostazioni dello streaming.

Una modifica minore, ma di particolare impatto nel range delle *feature* a disposizione, sarebbe la possibilità di inviare in streaming una sorgente audio abbinata a quella video. Nonostante FFmpeg preveda già questa possibilità, dato lo stato provvisorio dell'applicazione, si è deciso di non esporla né nel *core*, né nell'interfaccia grafica. In un futuro sarà però possibile rendere disponibile questa funzionalità, senza alterare in maniera significativa lo stato generale dell'applicativo.

Un'ulteriore aggiunta alle opzioni personalizzabili è costituita dalle tipologie di streaming che è possibile istanziare. Allo stato attuale, nell'interfaccia grafica sono state limitate le opzioni utilizzabili al solo streaming UDP verso un indirizzo IP multicast, sebbene la libreria supporti altre possibilità quali RTP, RTMP, RTSP, l'incapsulamento in datagrammi TCP e addirittura l'unicast e il broacast. Sarebbe utile,

dal punto di vista della forza del prodotto sul mercato, aggiungere un'opzione di streaming adattivo su protoccollo HTTP usando tecnologie apposite, come MPEG-DASH. FFmpeg, la libreria usata principalmente nello sviluppo del core, nelle versioni più recenti ha iniziato a supportare questa tecnologia. È possibile, con limitate modifiche, renderla funzionante anche per l'applicazione sviluppata in questa sede.

Applicando questa innovazione costituita dallo streaming adattivo verso HTTP, si entra nell'ambito delle tecnologie Web, quindi, per estensione, di Internet. Una funzionalità che alcuni moduli hardware presentano è quella di poter inviare anche all'esterno, nella più grande rete di reti al mondo, lo streaming prodotto. Per una questione di sicurezza (ad esempio, si può pensare, se il flusso video facesse parte di un sistema di sorveglianza) sarebbe auspicabile, precedentemente alla trasmissione in rete, crittografare lo stream di dati, effettuando quella che si chiama crittografia *end-to-end*, cioè dall'apparato trasmittente a quello ricevente. Ciò permetterebbe di evitare degli attacchi informatici di tipo MITM (*m*an *i*n *t*he *m*iddle), che si basano sull'intereccettamento di flussi di dati in viaggio su Internet.

Appendice A

Codice del back-end

A.1 capture.c

```
ctx *init_ctx() {
  // allocazione del contesto
  return context;
}
void open_device(ctx *cntx) {
  // apertura della periferica come un file, sfruttando chiamate non
      blocking
}
void v4l2_crop(ctx *cntx) {
  // impostazione del crop
}
int v4l2_format(ctx *cntx) {
  // impostazione del formato di cattura: spazio colore e dimensioni
}
void v4l2_framerate(ctx *cntx) {
  // impostazione del framerate di cattura
}
static void init_read(ctx *cntx, unsigned int buffer_size) {
  // allocazione delle strutture necessarie per I/O con funzione read
}
static void init_mmap(ctx *cntx) {
  // allocazione delle strutture necessario per I/O con funzione di memory
  // mapping
}
void init_device(ctx *cntx) {
  struct v4l2_capability cap; /* device capabilities */
  switch (cntx->io) {
    case MMAP:
    case USERPTR:
      if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
        dev_log(cntx->dev_name,
            "does not support streaming I/O, falling back to read/write");
      } else {
        break;
      }
    case READ:
      if (!(cap.capabilities & V4L2_CAP_READWRITE)) {
```

```
37          dev_critical(cntx->dev_name, "does not support read/write I/O");
38        }
39        break;
40    }
41    int buffer_size = v4l2_format(cntx);
42    if (cntx->cropcap) {
43      /* crops output to view */
44      v4l2_crop(cntx);
45    }
46    v4l2_list_framerate(cntx);
47    v4l2_framerate(cntx);
48    switch (cntx->io) {
49      case READ:
50        init_read(cntx, buffer_size);
51        break;
52      case MMAP:
53        init_mmap(cntx);
54        break;
55      case USERPTR:
56        init_userptr();
57        break;
58    }
59    gen_log("device %s is ready to capture frames\n", cntx->dev_name);
60  }
61  void start_capture(ctx *cntx) {
62    enum v4l2_buf_type type;
63    switch (cntx->io) {
64      case READ:
65        break;
66      case MMAP: {
67        unsigned int j;
68        for (j=0; j<cntx->buffers_n; ++j) {
69          /* create, set up and ioctl buffer */
70          if (-1 == xioctl(cntx->fd, VIDIOC_QBUF, &buffer)) {
71            cap_critical("VIDIOC_QBUF");
72          }
73        }
74        type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
75        if (-1 == xioctl(cntx->fd, VIDIOC_STREAMON, &type)) {
76          cap_critical("VIDIOC_STREAMON");
77        }
78        break;
79      }
80      case USERPTR:
81        break;
82    }
83  }
84  void capture_timeout(ctx *cntx) {
85    int temp;
86    temp = select(cntx->fd+1, &fds, NULL, NULL, &tv);
87    switch (temp) {
88      case -1:
89        if (EINTR == errno) {
90          gen_log("interrupted select syscall\n");
91        }
92        cap_critical("select");
```

```
93          break;
94        case 0:
95          gen_critical("select timeout\n");
96          break;
97      }
98  }
99  int read_frame(ctx *cntx) {
100   // legge il frame e lo memorizza in un buffer
101 }
102 void stop_capture(ctx *cntx) {
103   // ferma la cattura dalla scheda
104 }
105 void uninit_device(ctx *cntx) {
106   // dealloca tutta la memoria impiegata per la cattura
107 }
108 void close_device(ctx *cntx) {
109   // libera la periferica e dealloca il contesto
110 }
```

A.2 encoding.c

```
int set_framerate(struct fraction framerate) {
  int cap_fps = framerate.den / framerate.num;
  if (cap_fps > 30) {
    half_fps = true;
    return cap_fps / 2;
  } else {
    half_fps = false;
    return cap_fps;
  }
}
conv_ctx *conv_ctx_init(ctx *cntx) {
  // ... inizializzazione del contesto
  // comprende crf, tune, preset e framerate
  conv_context->crf = 28;
  conv_context->framerate = set_framerate(cntx->framerate);
  strcpy(conv_context->tune, "zerolatency");
  strcpy(conv_context->preset, "ultrafast");
  return conv_context;
}
void rgb24_init(ctx *cntx, conv_ctx *c_cntx) {
  // inizializza la struttura per convertire lo spazio colore da catturato
  // a RGB a 8 bit per canale (per la preview)
}
void rgb24(ctx *cntx, conv_ctx *c_cntx) {
  // effettua la conversione a RGB 24 bit
}
void scaler_init(ctx *cntx, conv_ctx *c_cntx) {
  // inizializza la struttura per convertire lo spazio colore dal catturato
  // a Y'UV 4:2:0 planare
}
void scale(ctx *cntx, conv_ctx *c_cntx) {
  // effettua la conversione a Y'UV 4:2:0
}
void mux_encoder_init(ctx *cntx, conv_ctx *c_cntx) {
  // inizializza encoder e muxer
  // imposta crf, tune e preset
  snprintf(crf, 16, "%d", c_cntx->crf);
  av_opt_set(c_cntx->cdc_ctx->priv_data, "crf", crf, 0);
  av_opt_set(c_cntx->cdc_ctx->priv_data, "tune", c_cntx->tune, 0);
  av_opt_set(c_cntx->cdc_ctx->priv_data, "preset", c_cntx->preset, 0);
  // apre lo stream di output dei dati
}
void mux_encode(ctx *cntx, conv_ctx *c_cntx) {
  // effettua prima l'encoding e poi il muxing
  // saltando i frame se necessario
}
void rgb24_uninit(ctx *cntx, conv_ctx *c_cntx) {
  // dealloca la struttura per la conversione a RGB 24 bit
}
void scaler_uninit(ctx *cntx, conv_ctx *c_cntx) {
  // dealloca la struttura per la conversione a Y'UV 4:2:0
}
void mux_encoder_uninit(ctx *cntx, conv_ctx *c_cntx) {
```

```
54      // dealloca l'encoder e il muxer
55  }
56  void conv_ctx_uninit(conv_ctx *c_cntx) {
57      // chiude il contesto di compressione e libera le risorse associate
58  }
```

A.3 error.c

```
1  void gen_critical(const char *s, ...) {
2    openlog("libv4lcapture", LOG_CONS || LOG_PID, LOG_USER);
3    va_list args;
4    va_start(args, s);
5    vsyslog(LOG_USER || LOG_CRIT, s, args);
6    va_end(args);
7    exit(EXIT_FAILURE);
8  }
9  void cap_critical(const char *s) {
10   gen_critical("error: %s (%d): %s\n", s, errno, strerror(errno));
11   exit(errno);
12 }
13 void dev_critical(const char *dev_name, const char *s) {
14   gen_critical("%s %s\n", dev_name, s);
15   exit(errno);
16 }
17 void gen_error(const char *s, ...) {
18   openlog("libv4lcapture", LOG_CONS || LOG_PID, LOG_USER);
19   va_list args;
20   va_start(args, s);
21   vsyslog(LOG_USER || LOG_ERR, s, args);
22   va_end(args);
23 }
24 int cap_error(const char *s) {
25   gen_error("error: %s (%d): %s\n", s, errno, strerror(errno));
26   return errno;
27 }
28 int dev_error(const char *dev_name, const char *s) {
29   gen_error("%s %s\n", dev_name, s);
30   return errno;
31 }
32 void gen_log(const char *s, ...) {
33   openlog("libv4lcapture", LOG_CONS || LOG_PID, LOG_USER);
34   va_list args;
35   va_start(args, s);
36   vsyslog(LOG_USER || LOG_INFO, s, args);
37   va_end(args);
38 }
39 int cap_log(const char *s) {
40   gen_log("error: %s (%d): %s\n", s, errno, strerror(errno));
41   return errno;
42 }
43 int dev_log(const char *dev_name, const char *s) {
44   gen_log("%s %s\n", dev_name, s);
45   return errno;
46 }
```

A.4 util.c

```
int xioctl(int fh, int request, void *arg) {
  // wrapper della funzione ioctl
  int ret;
  do {
    ret = ioctl(fh, request, arg);
  } while (-1 == ret && EINTR == errno);
  return ret;
}
```

A.5 test.c

```
int main(void) {
  ctx *context = init_ctx();
  open_device(context);
  init_device(context);
  context->stop = 0;
  bool infinite;
  if (context->stop == 0) {
    infinite = true;
  }
  conv_ctx *conv_context = conv_ctx_init(context);
  start_capture(context);
  rgb24_init(context, conv_context);
  scaler_init(context, conv_context);
  mux_encoder_init(context, conv_context);
  while (context->stop-- | infinite) {
    for (;;) {
      capture_timeout(context);
      if (read_frame(context)) {
        rgb24(context, conv_context);
        scale(context, conv_context);
        mux_encode(context, conv_context);
        break;
      }
    }
  }
  write_cached(context, conv_context);
  rgb24_uninit(context, conv_context);
  scaler_uninit(context, conv_context);
  mux_encoder_uninit(context, conv_context);
  conv_ctx_uninit(conv_context);
  stop_capture(context);
  uninit_device(context);
  close_device(context);
  close_log();
}
```

A.6 libv4lcapture.c

```
// contesti statici
ctx *cntx;
conv_ctx *c_cntx;
// interfaccia pubblica
// funzioni di impostazione dei parametri
void setDevice(const char *s);
void setIO(int sel);
void setFrames(int64_t no);
void setPixFmt(char a, char b, char c, char d);
void setCrop(int top, int left, int width, int height);
void setFormat(int width, int height);
void setFps(int num, int den);
void setFilename(const char *s);
void setCRF(int crf);
void setMuxer(const char *s);
void setTune(const char *s);
void setPreset(const char *s);

// funzioni di richiesta dei parametri
char *getDevice(void);
int getIO(void);
int64_t getFrames(void);
int getCropcap(void);
int *getDefCrop(void);
int *getCrop(void);
int *getDefFormat(void);
int *getFormat(void);
double getRatio(void);
int *getFpsList(void);
int *getDefFps(void);
int *getFps(void);
int getStrFps(void);
unsigned int getArea(void);
unsigned int getQArea(void);
char *getFilename(void);
int getCRF(void);
char *getMuxer(void);
char *getTune(void);
char *getPreset(void);

// l'unica funzione che restituisce un valore non primitivo, ma un buffer
uint8_t *getRGBData(void) {
  return c_cntx->rgb_dst[0];
}

// funzioni di controllo e di azione
void initCtx(void);
void openDevice(void);
void initDevice(void);
void initConvCtx(void);
void startCapture(void);
void initRGB(void);
void initScaler(void);
```

```
54 void initMuxEnc(void);
55 void capTimeout(void);
56 int readFrame(void);
57 void RGB(void);
58 void _scale(void);
59 void muxEncode(void);
60 void writeCache(void);
61 void uninitRGB(void);
62 void uninitScaler(void);
63 void uninitMuxEnc(void);
64 void uninitConvCtx(void);
65 void stopCapture(void);
66 void closeDevice(void);
67 void closeLog(void);
```

Appendice B

Codice del front-end

B.1 Thread di cattura e gestione impostazioni

```
class CaptureParams:

  VALID = ('device', 'io', 'frames', 'pix_fmt', 'crop',
      'format', 'fps', 'filename', 'crf', 'muxer', 'tune',
    'preset') # attributi validi

  def __init__(self, conf):
    self.conf = conf # struttura di configurazione
    self.enc = False # streaming on/off
    self.params = {} # parametri di cattura
    self._restore()
  def store(self):
    self.conf.set('settings', self.params.copy())
  def set_attr(self, key, value):
    if key in self.VALID:
      self.params[key] = value
      error.log('{} changed to {}'.format(key, value))
  def get_attr(self, key):
    if key in self.params:
      return self.params[key]
    return False
  def toggle_encoding(self):
    self.enc = not self.enc
    error.log('encoding set to {}'.format(self.enc))

class CaptureThread(QThread):

  def __init__(self, app, v4l, params, update):
    super().__init__() # inizializza il thread
    self.app = app # finestra della GUI
    self.v4l = v4l # oggetto libreria
    self.params = params # parametri di configurazione
    self.work = True # test di uscita
  def run(self):
    enc = self.params.encoding()
    self.v4l.initCtx()
    device = self.params.get_attr('device')
```

```
38      if device:
39        self.v4l.setDevice(device)
40
41      # ... per tutti i parametri presenti
42
43      if tune:
44        self.v4l.setTune(tune)
45      preset = self.params.get_attr('preset')
46      if preset:
47        self.v4l.setPreset(preset)
48      self.v4l.startCapture()
49      self.v4l.initRGB()
50      if enc:
51        self.v4l.initScaler()
52        self.v4l.initMuxEnc()
53      infinite = False
54      if frames == False or frames == 0:
55        infinite = True
56      while self.work and (infinite or count > 0): # ciclo di cattura frame
57        while True:
58          self.v4l.capTimeout()
59          if self.v4l.readFrame():
60            self.v4l.RGB()
61            self.update(self.v4l.getRGBData()) # aggiornamento preview
62            if enc:
63              self.v4l.scale()
64              self.v4l.muxEncode()
65            break
66        if not infinite:
67          count -= 1
68      if enc:
69        self.v4l.writeCache()
70      self.v4l.uninitRGB()
71      if enc:
72        self.v4l.uninitScaler()
73        self.v4l.uninitMuxEnc()
74      self.v4l.uninitConvCtx()
75      self.v4l.stopCapture()
76      self.v4l.closeDevice()
```

B.2 Ricerca di schede di cattura

```
def list_devices():
  ret = []
  for dev in os.listdir('/dev'):
    if dev.startswith('video'):
      ret.append('/dev/{}'.format(dev))
  return ret

def check_devices():
  for file in os.listdir('/dev'):
    if file.startswith('video'):
      return
  error.critical('no /dev/video* devices found')
```

B.3 Creazione del *route* statico per indirizzi UDP multicast

```
class Route:

  execs = ('ip', 'route')

  def __init__(self):
    if shutil.which('pkexec') is None:
      self.err.error('cannot find libpolkit')
    self.route = self._search()
  def _search(self):
    for i,x in enumerate(self.execs):
      cmd = shutil.which(x)
      if cmd is not None:
        return i
    error.error('cannot find a suitable command: '
      'install one of this tools {}'.format(self.execs))
  def _build(self, eth):
    skel = ['pkexec']
    if self.route == 0:
      skel.append(self.execs[self.route])
      skel.append('route')
      skel.append('add')
      skel.append('224.0.0.0/4')
      skel.append('dev')
      skel.append(eth)
      return skel
    elif self.route == 1:
      skel.append(self.execs[self.route])
      skel.append('add')
      skel.append('-net')
      skel.append('224.0.0.0')
      skel.append('netmask')
      skel.append('240.0.0.0')
      skel.append('dev')
      skel.append(eth)
      return skel
  def execute(self, eth):
    cmd = self._build(eth)
    try:
      sp.check_call(cmd, stdout=sp.DEVNULL, stderr=sp.DEVNULL)
    except sp.CalledProcessError as e:
      if e.returncode == 2:
        error.log('static route for multicast streaming already set')
      else:
        error.error('cannot set static route for multicast streaming')
```

Appendice C

Codice del client

C.1 *Utility* di creazione della configurazione

```
LOCATION = path.dirname(sys.argv[0])
SAVEFILE = path.join(LOCATION, 'save.json')
USERNAME = 'pi'
PASSWORD = 'raspberry'

def gen_config():
  clean_up()
  info = {}
  wallpath = path.join(LOCATION, 'config', 'piwall')
  with open(wallpath, 'w') as wall:
    info['wall'] = rand_name()
    info['wall_x'] = 0
    info['wall_y'] = 0
    tmp_arr = question(
      'Videowall arrangement? (ex. 2x2)',
      check=lambda e: 'x' in e and len(e) > 2
    )
    tmp_col, tmp_row = tmp_arr.split('x')
    info['arr_col'] = int(tmp_col)
    info['arr_row'] = int(tmp_row)
    info['tot_monitor'] = info['arr_col'] * info['arr_row']
    info['width'] = question(
      'Single monitor width?',
      type_='int',
      check=lambda e: e > 0
    )
    # ... uguale per altezza, larghezza in mm, bezel in mm
    info['bezel'] = round(info['width'] / info['mm_width'] * mm_bezel)
    info['wall_width'] = info['width'] * info['arr_col'] + \
        (2 * info['bezel'])
    info['wall_height'] = info['height'] * info['arr_row'] + \
        (2 * info['bezel'])
    info['monitors'] = []
    count = 1
    for i in range(info['arr_row']):
      x = 0 if i == 0 else (i * info['width']) + 2 * info['bezel']
      for k in range(info['arr_col']):
```

```
38            y = 0 if k == 0 else (k * info['height']) + 2 * info['bezel']
39            monitor = {
40              'id': count,
41              'name': rand_name(),
42              'width': info['width'],
43              'height': info['height'],
44              'x': x,
45              'y': y
46            }
47            info['monitors'].append(monitor)
48            count += 1
49        wall.write('[{}]\n'.format(info['wall']))
50        wall.write('x={}\n'.format(info['wall_x']))
51        wall.write('y={}\n'.format(info['wall_y']))
52        wall.write('width={}\n'.format(info['wall_width']))
53        wall.write('height={}\n'.format(info['wall_height']))
54        wall.write('\n')
55        for mon in info['monitors']:
56          wall.write('[{}]\n'.format(mon['name']))
57          wall.write('wall={}\n'.format(info['wall']))
58          wall.write('x={}\n'.format(mon['x']))
59          wall.write('y={}\n'.format(mon['y']))
60          wall.write('width={}\n'.format(mon['width']))
61          wall.write('height={}\n'.format(mon['height']))
62          wall.write('\n')
63          tilepath = path.join(LOCATION, 'config', 'pitile'+str(mon['id']))
64          with open(tilepath, 'w') as tile:
65            tile.write('[tile]\n')
66            tile.write('id=pi{}\n'.format(mon['id']))
67        wall.write('[pimcplayer]\n')
68        for mon in info['monitors']:
69          wall.write('pi{}={}\n'.format(mon['id'], mon['name']))
70
71    def upload_all():
72      scpclient._scp_read_response = _fixed_scp_read_response
73      piwall = path.join(LOCATION, 'config', 'piwall')
74      pitile = path.join(LOCATION, 'config', 'pitile')
75      pwomxp_serv = path.join(LOCATION, 'scripts', 'pwomxplayer')
76      pwomxp_conf = path.join(LOCATION, 'config', 'pwomxplayer')
77      for dev in sorted(info['monitors'], key=lambda x: x['id']):
78        print('Tile {}:'.format(dev['id']))
79        hostname = question(
80          'Hostname or address?',
81          check=lambda e: len(e) > 0
82        )
83        # ... allo stesso modo per utente e password
84        ssh = ssh_connect(hostname, user, password)
85        ssh_file(piwall, ssh, '.piwall')
86        ssh_file('{}{}'.format(pitile, dev['id']), ssh, out_name='.pitile')
87        ssh_file(pwomxp_conf, ssh, out_name="pwomxplayer.conf", permissions='
              0755')
88        ssh_command('mv pwomxplayer.conf /etc/default/pwomxplayer', ssh,
              password,
89            True, True)
90        ssh_file(pwomxp_serv, ssh, out_name="pwomxplayer.serv", permissions='
              0755')
```

```
91      ssh_command('mv pwomxplayer.serv /etc/init.d/pwomxplayer', ssh,
            password,
92           True, True)
93      ssh_command('update-rc.d pwomxplayer defaults', ssh, password, True,
            True)
94      ssh_command('/etc/init.d/pwomxplayer start', ssh, password, True, True)
95
96  def main():
97    while True:
98      if len(menu) == 0:
99        menu.append('Generate configuration files')
100       menu.append('Install configuration and player service')
101       menu.append('Clean created configuration files')
102     else:
103       print()
104     number = print_menu()
105     if number == 1:
106       gen_config()
107     elif number == 2:
108       upload_all()
109     elif number == 3:
110       clean_up(True)
111
112 main()
```

C.2 Esempio di file `.piwall`

```
[aIqcNWSD]
x=0
y=0
width=3980
height=2540

[kKlPYjFg]
wall=aIqcNWSD
x=0
y=0
width=1920
height=1200

[lZFBWBfP]
wall=aIqcNWSD
x=0
y=1340
width=1920
height=1200

[seCsspQf]
wall=aIqcNWSD
x=2060
y=0
width=1920
height=1200

[ypnCNCqv]
wall=aIqcNWSD
x=2060
y=1340
width=1920
height=1200

[pimcplayer]
pi1=kKlPYjFg
pi2=lZFBWBfP
pi3=seCsspQf
pi4=ypnCNCqv
```

C.3 Esempio di file `.pitile`

```
[tile]
id=pi1
```

Bibliografia

[1] Fischer, Walter. *Digital Video and Audio Broadcasting Technology a Practical Engineering Guide*. 3rd ed. Heidelberg: Springer-Verlag, 2010.

[2] Ozer, Jan. *Producing Streaming Video for Multiple Screen Delivery*. Galax, Va.: Doceo Publishing, 2013.

[3] Marini, Daniele, Maresa Bertolo, and Alessandro Rizzi. *Comunicazione Visiva Digitale. Fondamenti Di Eidomatica*. Milano: Addison Wesley Longman Italia Editoriale, 2002.

[4] Poynton, Charles. *Digital Video and HDT: Algorithms and Interfaces*. Morgan Kaufmann Publishers, 2003.

[5] Hurkman, Alexis. *Color Correction Handbook Professional Techniques for Video and Cinema*. 2nd ed. Peachpit Press, 2014.

[6] *Digital Signage Market Analysis by Technology (LED, LCD, Front Projection), by Application (Transportation, Retail, Corporate, Banking, Healthcare, Education) and Segment Forecasts to 2020*. Grand View Research, 2014.

[7] "LiveStream Technology." Datapath Limited. February 27, 2015. Accessed November 10, 2015. `http://www.datapath.co.uk/solutions/datapath-technology/item/481-LiveStream`.

[8] Wikipedia contributors. "Video wall." Wikipedia, The Free Encyclopedia, October 12, 2015. Accessed November 24, 2015. `https://en.wikipedia.org/w/index.php?title=Video_wall&oldid=685427491`.

[9] Wikipedia contributors. "Chroma subsampling." Wikipedia, The Free Encyclopedia. October 18, 2015. Accessed November 10, 2015. `https://en.wikipedia.org/w/index.php?title=Chroma_subsampling&oldid=686326985`.

[10] "What Is a Video Wall?" Pixell. Accessed November 22, 2015. `http://www.pixell.com/what_is_a_vw.htm`.

[11] Griffin, Daniel. "Rise of the Network Video Wall." DigitalSignageToday. June 2, 2015. Accessed November 22, 2015. `http://www.digitalsignagetoday.com/articles/rise-of-the-network-video-wall/`.

[12] Hughes, Richard. "Frequently Asked Questions." Hughski.com. Accessed November 22, 2015. `http://www.hughski.com/faq.html`.