UNIVERSITA' DEGLI STUDI DI BOLOGNA
FACOLTA' DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica
Reti di Calcolatori

POLITICHE DI BUFFERIZZAZIONE IN INFRASTRUTTURE PER L'EROGAZIONE DI SERVIZI MULTIMEDIALI A DISPOSITIVI WIRELESS

Candidato:

PIERANGELI DIEGO

Relatore:

Chiar.mo Prof. Ing. ANTONIO CORRADI

Correlatore:
Dott. Ing. LUCA FOSCHINI

ANNO ACCADEMICO: 2003 – 2004

INDICE

PAROLE CHIAVE

Wireless Internet
Streaming video
Client
Bufferizzazione
JMF/RTP

Introduzione

Lo sviluppo dei protocolli di rete e della tecnologia che sta avvenendo da tempo permette di immaginare uno scenario in cui sia possibile trasmettere VoD (Video on Demand) a numerosi utenti in movimento, connessi a diversi punti della rete e con terminali di accesso di tipo diverso.

Questo scenario rende necessario uno strato Applicativo capace di gestire lo streaming video e di gestire i nodi coinvolti dallo stesso.

I problemi posti dallo streaming video sono molti: innanzitutto per un PlayBack senza interruzioni di un video il client deve ricevere e decodificare un frame periodicamente (solitamente ogni 40 ms nel caso di video codificato in Mpeg); se il Client non ha ricevuto completamente un frame entro il suo tempo di PlayBack il Client perde (tutto o in parte) il Frame, soffrendo quella che è chiamata "PlayBack Starvation", [Carestia in PlayBack].

Perchè l'utente percepisca una buona qualità video è necessario che la percentuale di Starvation sia bassa (10^{-5}-10^{-2}).

Se poi si considera un utente mobile si deve tenere in conto la eventuale disconnessione casuale oppure il cambio di "cella" con la conseguente disconnessione dalla precedente e riconnessione alla nuova (handoff) e soprattutto l'alta probabilità di errori tipica di una connessione wireless che, data la restrizione sui tempi imposta dallo streaming lascia poco spazio ad una eventuale ritrasmissione.

Per poter gestire questa “volatilità” della connessione si vuole progettare un sistema di bufferizzazione lato client capace di bufferizzare sul client parte dello stream video prima della presentazione di questo all’utente in modo che questi non si accorga della eventuale disconnessione e riconnessione garantendo quindi la QoS [Quality of Service] nella visualizzazione dello stream.
Questo naturalmente è possibile solo a patto che il tempo di riconnessione sia inferiore alla lunghezza del video bufferizzata sul client, problema che si potrebbe risolvere creando un buffer “dinamico”, cioè la cui lunghezza non sia statica e determinata a priori ma sia dinamicamente configurabile in base a parametri esterni come la qualità della connessione ed il traffico di rete.
Un’altra considerazione che bisogna fare è che la velocità a cui l’Utente effettua il PlayBack (ossia “consuma” il video) è fissata e conosciuta a priori (ed è pari al FrameRate del video); se si avesse un client che effettua la presentazione all’utente prelevando i dati dal buffer il server potrebbe variare la velocità a cui invia i dati senza influire sulla qualità video percepita dall’Utente, se la connessione lo consente (stato del traffico di rete, qualità del segnale).

CAPITOLO 1

Scenario di lavoro e linee guida dello sviluppo

Lo scenario in cui si inserisce questa tesi è quello dello streaming di un contenuto multimediale da un server ad una macchina client mobile, connessa alla rete attraverso un dispositivo wireless.

Un utente può avere due tipi di movimento: nomadico o roaming.

Nel movimento nomadico l'utente si sposta tra più terminali fissi chiedendo di trasferire fra essi la sua sessione di lavoro in modo che arrivato al nuovo terminale possa essere ad esempio in grado di riprendere la fruizione del contenuto multimediale dal punto in cui era stata interrotta, il movimento contrapposto a questo è detto roaming cioè l'utente si muova assieme al suo terminale ed è il tipo di movimento preso in considerazione nel nostro scenario.

Nel nostro caso consideriamo un dispositivo wireless che è connesso con un Wireless Access Point, quando lascia la zona coperta da quello a cui era connesso per passare alla zona coperta da un altro access point si ha una disconnessione dal precedente ed una connessione al nuovo; questa perdita e riconnessione si chiama handoff.

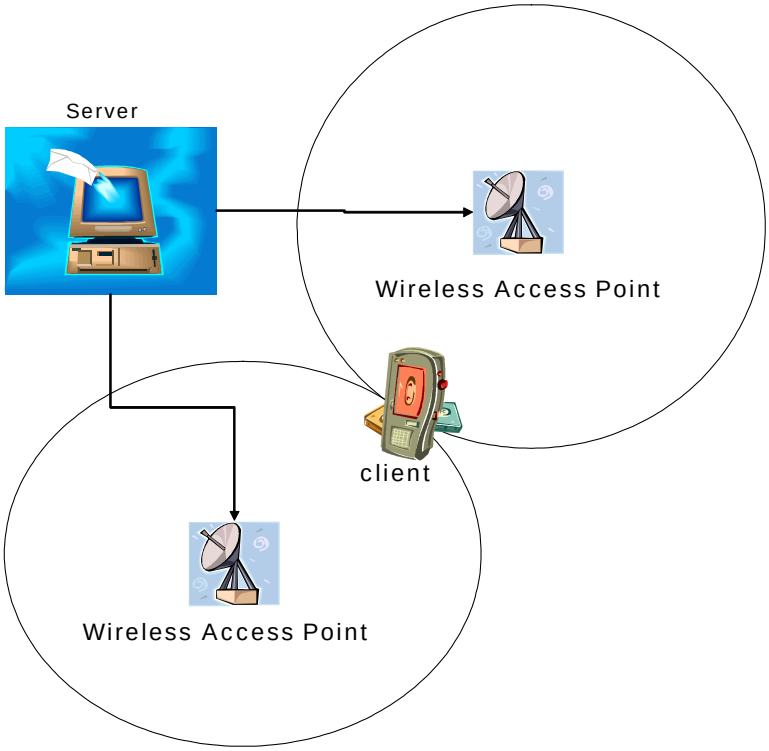


Figura 1: schema dello scenario

Durante l'Handoff il client non riceve più il flusso multimediale trasmesso dal server, bloccando quindi la presentazione del contenuto multimediale all'utente.

Lo scopo che si prefigge questa tesi è di creare un componente che permetta la bufferizzazione di parte del flusso multimediale sul client, in modo che nonostante la perdita di connessione la presentazione possa essere fluida e continua.

Il requisito fondamentale però è la qualità del servizio (QoS); nel nostro caso la qualità riguarda la presentazione del contenuto multimediale all'utente, che deve rimanere fluida nonostante l'handoff.

Quindi l'obbiettivo che ci siamo posti dovendo creare questo progetto è la trasparenza di tutte le operazioni effettuate dal client all'utente,

che deve poter godere del contenuto multimediale come se l’handoff non ci fosse stato.

CAPITOLO 2

2 Gestione dell’handoff

Per poter trattare le tematiche fondamentali riguardanti le politiche di gestione in caso di handoff bisogna prima introdurre alcune definizioni e spiegazioni basilari.

2.1 Definizioni di base

2.1.1 Flusso multimediale

Il flusso è una definizione che viene introdotta per indicare il recapito e la presentazione di un dato contenuto multimediale, dove per contenuto multimediale viene inteso un qualunque tipo di dato che varia con continuità rispetto al tempo.
Esempi di flussi multimediali possono essere ad esempio un video o un audio.

2.1.2 Unicast, Multi-Unicast, Broadcast e Multicast

L’IP supporta vari tipi di schemi di indirizzamento : Unicast, Broadcast e Multicast; il tipo di indirizzamento è indicato dall’ indirizzo IP del pacchetto.
Unicast è lo schema di indirizzamento più utilizzato oggi su Internet e descrive la trasmissione di un pacchetto da una sorgente ad un solo indirizzo destinatario (conosciuto anche come trasmissione punto a punto o end to end).

Multi-Unicast è una semplice espansione dell'unicast in cui i pacchetti vengono duplicati ed inviati a più destinatari e non solo ad uno.
Lo schema Broadcast descrive la trasmissione di pacchetti a tutti i destinatari su una particolare sottorete; anche se è più efficiente del Multi-Unicast (i pacchetti non vengono duplicati) è limitato dal vincolo di una singola sottorete.
Lo schema Multicast prevede che il Trasmettitore invii i pacchetti ad un solo indirizzo, quello della sessione Multicast. I destinatari (uno o più) si uniscono alla sessione chiedendo di ascoltare i pacchetti in arrivo sull'indirizzo associato alla sessione.
A questo punto è l'infrastruttura di rete che è responsabile di consegnare i pacchetti a tutti i destinatari.

2.1.3 Larghezza di banda e compressione

La larghezza di banda è il parametro che misura la portata del canale di trasmissione perché misura la quantità di dati che possono transitare nel canale nell'unità di tempo.
Questo parametro (che è misurato in bps, bit per secondo) è molto importante nella trasmissione di flussi multimediali perché questi richiedono una larghezza di banda molto ampia.
Per questo sono stati studiati algoritmi di compressione che limitino i requisiti di banda di un dato contenuto multimediale.
I componenti che si occupano di questa compressione e relativa decompressione si chiamano codec (Compression/DECompression).

La maggior parte degli algoritmi è detta “con perdita”, volendo con ciò indicare che non preservano perfettamente il contenuto multimediale; una frazione della qualità originale viene persa durante la compressione e non è più recuperabile.
Esistono 3 strategie generali che vengono sfruttate dagli algoritmi di compressione:

Ridondanza spaziale:
Questa strategia sfrutta la ripetizione di dati all’interno del frame considerato ed è molto utilizzata per video (mentre non è valida per un contenuto audio).
Ad esempio molte immagini hanno ampie regioni di uno stesso colore, si pensi ad uno sfondo oppure a vestiti di uno stesso colore.
Piuttosto che trasmettere informazioni riguardanti ogni punto di queste regioni si può trasmettere il colore e la dimensione della regione, riducendo la larghezza di banda necessaria alla trasmissione.

Ridondanza temporale
Questa strategia sfrutta il fatto che le differenze tra due frame video successivi oppure due campioni audio generalmente è molto piccola se paragonata alle dimensioni del frame o del campione, quindi piuttosto che inviare il nuovo frame è preferibile inviare le differenze con il frame precedente.

Questo approccio è generalmente molto efficiente sia per un contenuto video sia per un contenuto audio ma porta ad alcuni problemi; infatti dato che viene trasmessa la differenza tra il frame attuale e quello precedente la perdita di frame è molto più pesante (per questo motivo la qualità di un contenuto multimediale codificato con questo algoritmo cala con il passare del tempo).
Per risolvere questo problema la maggior parte degli algoritmi di compressione inviano ad intervalli prestabiliti un frame completo, chiamato KeyFrame.

Difetti della percezione umana

Gli organi visivi ed acustici dell'uomo hanno dei difetti che possono essere sfruttati da algoritmi di compressione.
Ad esempio un essere umano distingue parti dello spettro meno chiaramente di altre oppure sente meglio le componenti basse di un suono piuttosto che quelle alte.
Queste debolezze possono essere sfruttate ad esempio dedicando meno risorse alle regioni la cui percezione è inferiore.

2.2 Politiche ed architetture per la gestione dell'Handoff

2.2.1 protocollo di Prefetch

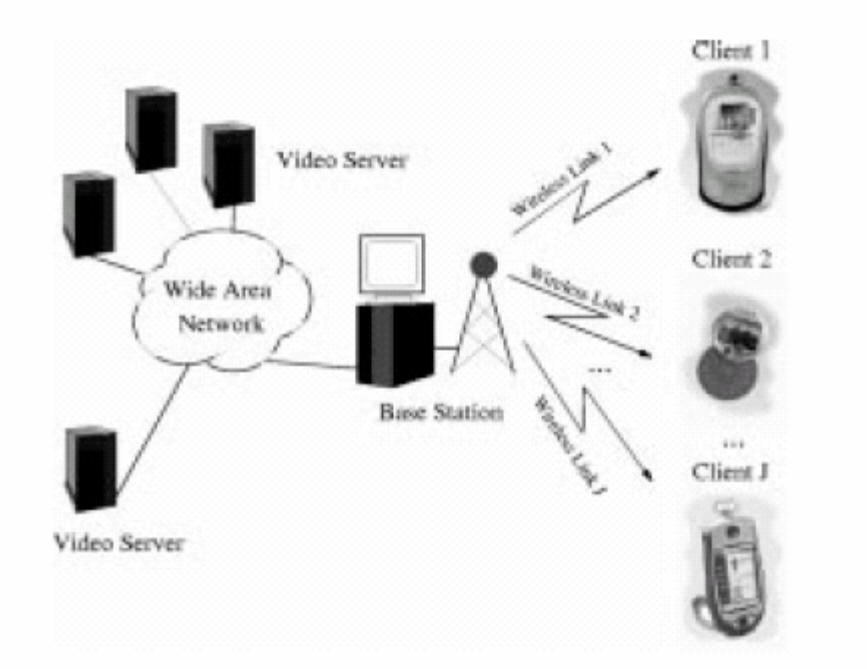


Figura 2: scenario, figura proveniente da [1]

La soluzione di prevedere un Buffer nel client è stata già sfruttata da altri protocolli, il lavoro di Frank Fitzek e Martin Reisslein [1] ad esempio sfrutta la presenza del buffer nel client per realizzare un sistema ad alte prestazioni, capace di gestire con efficienza più flussi diretti a diversi client mobili (figura 2).

Questo protocollo infatti utilizza una procedura chiamata JSQ (Join the Shortest Queue), la quale alloca più risorse di trasmissione ai client che hanno a disposizione meno frame nel loro Buffer e utilizza un sistema di sondaggio del canale (Channel Probing) per riuscire a modulare la trasmissione sulla connessione wireless.

La modulazione avviene allocando più canali di trasmissione ad un singolo client, ad esempio nella versione attuale della tecnologia UMTS è possibile allocare fino a 15 canali paralleli ad un singolo client.

La politica JSQ cerca di bilanciare la durata dei segmenti di video bufferizzati sui client, allocando più canali (e quindi inviando più pacchetti) ai client che hanno solo una piccola riserva di video nel loro buffer.

Il sondaggio delle capacità del canale è una funzione necessaria in questo scenario, infatti se il canale è molto disturbato il buffer del client è consumato velocemente, un protocollo basato solamente sulla politica JSQ allocherebbe più capacità a questo client, ma a causa del disturbo sul canale trasmissivo i pacchetti verrebbero comunque persi facendo consumare ulteriormente il Buffer del client.

La stazione base continuerebbe ad allocare sempre più risorse a questa trasmissione, tralasciando gli altri client e causando una starvation globale.

Questo problema viene risolto monitorando lo stato del canale trasmissivo e allocando le risorse di trasmissione anche in base a questo oltre che alla politica JSQ.

La presenza del buffer permette al client di continuare la presentazione del contenuto multimediale anche in caso di connessione disturbata o della sua perdita, e grazie alla politica JSQ garantisce che alla riconnessione il buffer del client venga riempito nuovamente.

2.2.2 Multi level Buffer Architecture

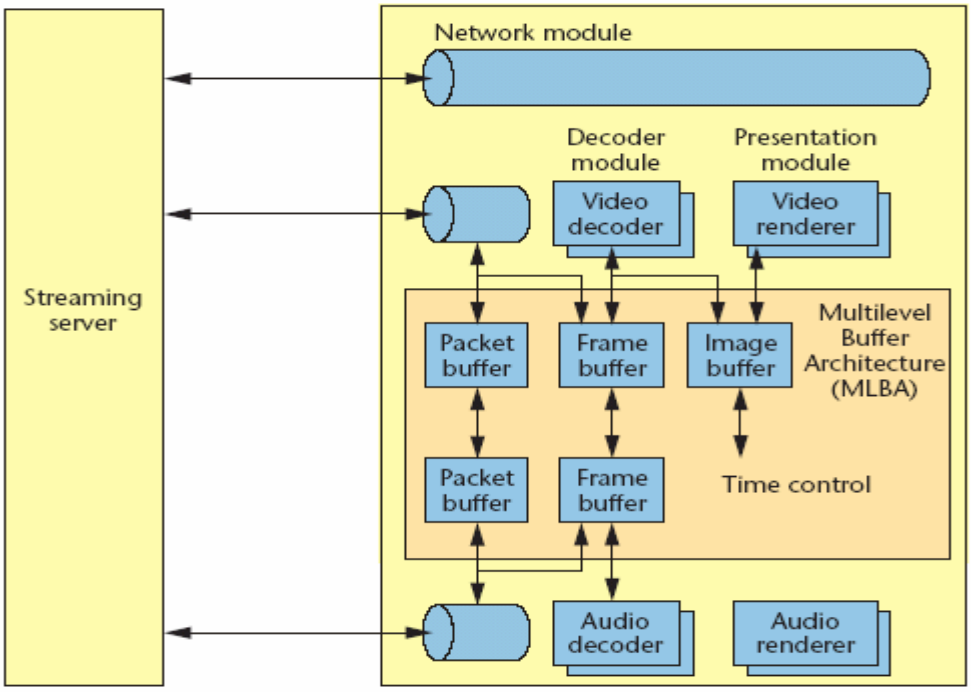


Figura 3: schema del MLBA client, preso da [2]

Questa architettura è stata introdotta dall'osservazione che in uno stream i tempi di decodifica di un frame (soprattutto nel formato MPEG4) sono molto variabili, la maggior parte dei frame ha un tempo di decodifica inferiore al framerate mentre alcuni hanno un tempo di decodifica di molto superiore.

Memorizzando in un buffer dedicato i frame già decodificati prima di presentarli il sistema può sfruttare il tempo rimanente quando la decodifica è più veloce del tempo di presentazione.

L'architettura, Figura 3, prevede tre tipi di buffer: un buffer dove verranno inseriti i pacchetti all'arrivo (Packet buffer), un buffer dove verranno inseriti i frame applicativi (un Frame può essere separato in

più pacchetti RTP) ed un buffer (Image buffer) dove verranno inseriti i frame decodificati.

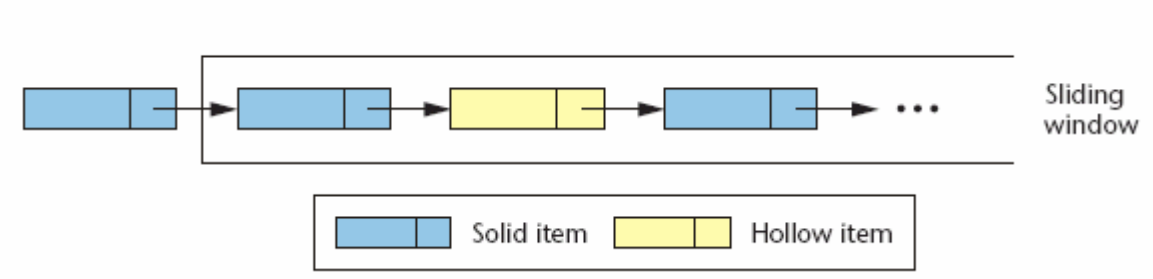


Figura 4: schema del Packet buffer, preso da [2]

Il Packet buffer (la cui struttura è schematizzata nella Figura 4) è implementato come una lista di oggetti solidi (Solid item) oppure oggetti vuoti (Hollow item).

Gli oggetti solidi contengono un pacchetto mentre quelli vuoti sono dei semplici segnaposto.

All'arrivo di un pacchetto il client calcola la sua posizione nel buffer in base al suo numero di sequenza, se due pacchetti adiacenti non hanno numeri di sequenza in successione vengono inseriti tra loro un oggetto vuoto.

Il client utilizza una sliding window per gestire l'arrivo in maniera disordinata di pacchetti, i pacchetti che sono fuori dalla finestra vengono considerati pacchetti persi, in caso contrario vengono inseriti al loro posto (eliminando eventualmente un oggetto vuoto).

I pacchetti vengono poi riassemblati in un frame applicativo ed inseriti nel Frame buffer, Figura 5.

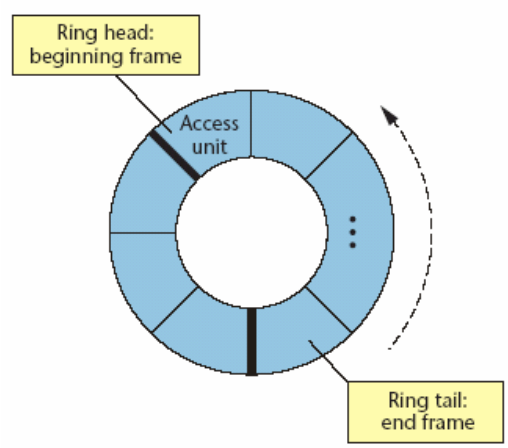


Figura 5: schema del Frame buffer, preso da [2]

Il Frame buffer utilizza una struttura ad anello per contenere i frame applicativi che devono essere decodificati ed inoltre garantisce in caso di HandOff di poter continuare la presentazione sul client in attesa della riconnessione.

L'Image buffer, Figura 6, contiene frame decodificati immagazzinati in una coda gestita con politica FIFO e pronti per essere presentati; questo permette di non avere problemi di presentazione anche se per alcuni frame il processo di decodifica impiega più tempo di quello che è il framerate, infatti nonostante il codec impieghi molto tempo per questa decodifica il processo di presentazione ha già a disposizione molti frame già decodificati da presentare.

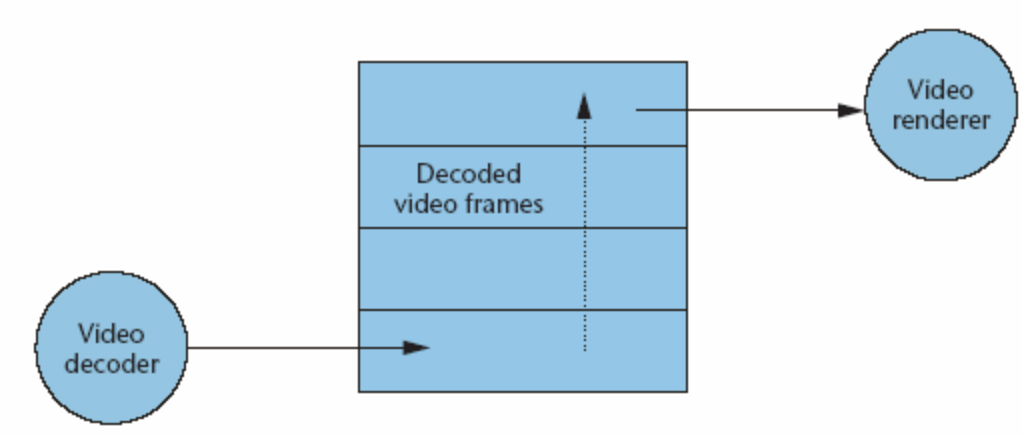


Figura 6: schema dell' Image buffer, preso da [2]

CAPITOLO 3

3 Tecnologia Utilizzata

3.1 Java Media Framework ed implementazione RTP del JMF

3.1.1 Java Media Framework

La tecnologia usata per sviluppare il oggetto di questo lavoro di tesi è il Java Media Framework,una libreria java per la gestione di oggetti multimediali.

Il motivo per cui è stato scelto questo framework ed anche il linguaggio java è la sua portabilità su piattaforme diverse che li rendono la soluzione perfetta per un sistema che deve essere eterogeneo e gestire diversi tipi di terminali, dai computer dotati di una connessione fissa ai PDA dotati di una connessione wireless.

Esistono quattro implementazioni del JMF, una che usa codice scritto puramente in java (Cross Platform Edition, non sempre la più efficiente in quanto la decodifica di molti stream video è un'operazione pesante in termini di cicli di CPU) ed altre (Solaris, Linux e Windows performance pack) che utilizzano plug-in scritti in C e implementati specificatamente per ogni piattaforma.

JMF mette a disposizione del programmatore molti metodi per gestire la distribuzione di un flusso multimediale (ad esempio un video).

In particolare JMF permette di programmare applicazioni multimediali utilizzando componenti ed API ad alto livello già fornite dal FrameWork stesso (ad esempio il Player, classe auto-configurante in grado di effettuare la visualizzazione di un flusso richiedendo solo

poche indicazioni da parte del programmatore) che semplificano la programmazione ma limitano l'espressività oppure utilizzare API di più basso livello che hanno una grande potenza espressiva ma la cui programmazione è molto più complessa.

DataSource

La classe DataSource fornisce un modello semplificato ed astratto del Media che può essere utilizzato come parte della catena di processing o di controllo.

La sua costruzione richiede la specifica di un protocollo e della locazione da cui il Media può essere ottenuto.

Le istanze del DataSource gestiscono il trasferimento del Media controllando una o più istanze di SourceStream, che rappresentano gli Stream del Media.

Pur essendo possibile creare direttamente un DataSource tramite i suoi due costruttori è in pratica inutile dato che il DataSource base non fornisce nessuna funzionalità, sono le sue sottoclassi (create utilizzando gli appropriati metodi della classe Manager) a possedere le funzionalità desiderate.

I metodi chiave di questa classe sono *connect()*, *disconnect()*, *start()* e *stop()*; i primi due aprono (o chiudono) la connessione al MediaSource specificato alla creazione del DataSource, gli altri due iniziano (o fermano) il trasferimento dei dati.

I DataSource possono essere classificati lungo due assi: un'asse è il modo con cui il trasferimento è realizzato (modalità push o pull) e

l’altro riguarda le unità con cui le informazioni sono trasferite (raw o buffer).

I PullDataSource hanno il trasferimento inizializzato e controllato dal client (ad esempio i protocolli http e FTP) mentre i PushDataSource hanno il trasferimento controllato dal server.

Per quello che riguarda il metodo con cui le informazioni sono trasferite il RawDataSource trasferiscono le informazioni come semplice flusso di byte, mentre i BufferDataSource incapsulano le informazioni dentro oggetti di tipo Buffer.

Due categorie per ciascuno dei due assi portano alla presenza di quattro sotto-classi di DataSource: *PullDataSource*, *PushDataSource*, *PullBufferDataSource*, *PushBufferDataSource*.

Due tipi speciali di DataSource possono essere creati attraverso metodi della classe Manager: cloneable e merged.

Il DataSource cloneable può essere clonato per permettere di processare contemporaneamente, mentre più DataSources possono essere unite creando un merged DataSource che contiene tutti i loro SourceStream.

Format

La classe Format permette di descrivere il formato del flusso multimediale in senso astratto; oggetti di tipo Format non forniscono informazioni sulla codifica oppure sulle temporizzazioni.

La classe Format è estesa da 3 classi più specifiche: AudioFormat, VideoFormat e ContentDescriptor.

Queste classi sono poi estese da classi specifiche per ogni tipo di formato (ad esempio YUVFormat, H263Format).
Ogni flusso ha associato un ContentDescriptor ed ogni Buffer ha un suo formato che può essere ottenuto attraverso il metodo *getFormat()*.
Ad ogni stadio di processo del flusso è associato un suo formato, ad esempio un filmato è inviato con il formato H263_RTP, poi è decodificato e convertito in YUV, infine è convertito in RGB per poter essere presentato all'utente.

Buffer

La classe Buffer è il contenitore che permette di trasferire dati da uno stadio di processo ad un altro.
Nel JMF il Buffer rappresenta un singolo frame del flusso multimediale, frame che secondo il livello in cui ci troviamo può essere o un frame applicativo o un frame RTP.
Un Buffer contiene solitamente o un frame completo oppure una sua parte oltre ad informazioni come il Formato del Frame, timeStamp (riferito all'istante in cui il frame dovrebbe essere presentato), lunghezza, numero di sequenza ed altre informazioni (tutte accessibili tramite metodi appropriati).

3.1.2 RTP e sua implementazione nel JMF

3.1.2.1 Introduzione a RTP

I pacchetti di dati sono trasferiti attraverso la rete grazie ai protocolli di trasporto, che sfruttano i servizi offerti dai sottostanti protocolli di rete (eg: IP) per far giungere i pacchetti ai loro destinatari.
Esistono due implementazioni dei protocolli di trasporto, TCP (Trasmission Control Protocol) e UDP (User Datagram Protocol).
Il protocollo UDP è un protocollo che utilizza una semantica may-be, cioè un pacchetto è inviato una sola volta e non n'è garantito l'arrivo, quindi ogni azione per garantire la Qualità del Servizio (arrivo dei pacchetti, richiesta di rinvio nel caso di perdita dei pacchetti) deve essere fatta a livello applicativo.
Il protocollo TCP è un protocollo che utilizza una semantica at-most-once, cioè se il pacchetto di dati arriva viene considerato al più una volta, inoltre il rinvio nel caso di perdita di pacchetti è gestito al livello del protocollo assieme alla gestione dell'ordine dei pacchetti, garantendo una maggiore Qualità di Servizio.
In caso di congestione della rete però il protocollo TCP rallenta di molto la sua trasmissione, rendendolo inadatto in questo caso ad un'applicazione in tempo reale come lo Streaming di un contenuto multimediale.
Per lo streaming una scelta usuale è quella del protocollo RTP (Real-time Transport Protocol) accoppiato all'UDP e supportato da JMF; il protocollo RTP fornisce metodi per creare un sistema di trasporto end-

to-end capace di supportare applicazioni in real-time ed inoltre pur non garantendo la Qualità del Servizio il protocollo supporta il monitoraggio della trasmissione.
I pacchetti RTP consistono di un Header di 12 Byte seguito da un Payload di lunghezza dipendente dal media trasportato.
L'Header contiene informazioni come il tipo di Payload, il time stamp, il numero di sequenza e l'identificatore della sorgente.
Al protocollo RTP (che si occupa del trasporto) è affiancato il protocollo RTCP (RTP Control Protocol), protocollo di controllo che permette di monitorare la QoS .
Il protocollo RTCP consiste in vari tipi di pacchetti, che contengono nell'Header sempre un Source Descriptor.

Sender Report - Pacchetto prodotto da partecipanti che hanno inviato pacchetti, contiene informazioni sui pacchetti inviati (numero totale e bytes totali) ed informazioni di sincronizzazione.
Receiver Report – Pacchetto prodotto da partecipanti che hanno ricevuto recentemente dati. Contiene informazioni come il numero di pacchetti persi e ricevuti e il più alto numero di sequenza ricevuto.
Source Description – Descrizione della sorgente del pacchetto.
Bye – Inviato da un partecipante che lascia la sessione.
App – Pacchetto che può essere utilizzato da un'applicazione per definire i propri messaggi specifici su RTPC.
La banda occupata da questi pacchetti RTCP è al massimo il 5% della banda totale a disposizione, infatti, inizialmente il protocollo era stato

pensato per connessioni di tipo multicast, perciò se non ci fosse stato questo limite per connessioni con molte macchine le informazioni di controllo avrebbero potuto saturare la banda.
Nell'implementazione attuale del JMF l'invio dei pacchetti RTCP è automatizzato ed il programmatore non può gestirlo; quindi per inviare informazioni di controllo specifiche dell'applicazione tra C/S è stato necessario utilizzare una terza connessione di controllo oltre a quella su cui è inviato lo stream e quella RTCP.

3.1.2.2 implementazione JMF di RTP

Nel JMF esistono 3 Packages che riguardano il protocollo RTP:
javax.media.rtp
javax.media.rtp.event
javax.media.rtp.rtcp

L'RTP supporta solo alcuni formati e content-type; per i content-type l'unico supportato è il raw_rtp, mentre per i formati video le scelte a disposizione sono 3: JPG_RTP, H261_RTP e H263_RTP.
Prima di proseguire bisogna introdurre la definizione di *Sessione RTP*; una Sessione è l'associazione tra più applicazioni che comunicano fra loro tramite RTP.
Una Sessione è identificata da un indirizzo di rete ed un paio di porte (una per i pacchetti RTP ed una per i pacchetti RTCP) nel caso di una sessione multicast, da due indirizzi di rete (trasmettitore e destinatario) nel caso di connessione Unicast.

3.1.2.2.1 RTPManager e ReceiveStream

L'architettura del JMF fornisce una classe che si occupa della gestione dell'RTP e della gestione delle sessioni, RTPManager; all'arrivo di una nuova connessione, attraverso l'uso di un paradigma event-driven avverte tramite l'invio di un evento le applicazioni che si sono registrate presso di lui.

RTPManager è una classe astratta (al contrario del Manger standard del JMF) e istanze di questa classe possono essere ottenute solo tramite il metodo statico getInstance().

Una volta che l'RTPManager è stato ottenuto si può inizializzare la sessione invocando il metodo *initialize()* e passandogli come parametro un SessionAddress (formato dell'indirizzo locale e dalla porta su cui dovrebbero arrivare i pacchetti RTP; non è necessario specificare la porta RTCP perché è presa di default quella subito successiva a quella RTP).

Inoltre il metodo *initialize()* permette anche di specificare parametri importanti come la frazione di banda da utilizzare per i pacchetti RTCP e se utilizzare una trasmissione criptata (al momento sono disponibili i metodi di codifica DES, TRIPLE DES e MD5).

Il metodo *addTarget()* è poi utilizzato per specificare l'altro lato della sessione attraverso un indirizzo ed una porta per il traffico RTP.

La chiamata a *addTarget()* apre la sessione permettendo il trasferimento dei pacchetti RTP e la creazione di rapporti RTCP.

Come un target può essere aggiunto può anche essere rimosso con il metodo *removeTarget()* tipicamente utilizzato quando la sessione è terminata.

In una connessione di tipo Unicast è necessario che entrambi i membri della connessione facciano un *addTarget()* prima che sia possibile inizializzare realmente la Sessione, perché una sessione unicast è un canale end-to-end identificato da due indirizzi di rete (locale e mittente/destinatario) e due porte (locale e mittente/destinatario) e per poter effettuare realmente la comunicazione, la sessione deve essere inizializzata correttamente da entrambi i lati.

L'RTPManager gestisce flussi che cadono in due categorie differenti: SendStream e ReceiveStream; i SendStream sono creati attraverso il metodo *createSendStream()* mentre i ReceiveStream vengono creati automaticamente appena il nuovo flusso viene ricevuto e possono essere ottenuti attraverso l'evento *NewReceivedStreamEvent.*

Il metodo *createSendStream()* crea un nuovo flusso a partire da un DataSource esistente e richiede due argomenti: il DataSource ed l'indice della traccia che si intende trasferire (1 per la prima traccia, 2 per la seconda e così via. 0 indica che si vogliono multiplexare tutte le tracce in una sola).

Lato client la cosa più importante è prevedere dei listener, cioè degli ascoltatori che vengono associati agli eventi RTP e RTCP lanciati dal RTPManager e che siano in grado di gestirli.

Gli eventi più importanti nel nostro caso sono gli eventi di Sessione e di Ricezione ed entrambe le interfacce listener (*SessionListener* e

ReceiveStreamListener) definiscono un singolo metodo *update()* che accetta un'evento della classe associata al listener.
Gli eventi di Sessione (*SessionEvent*) permettono di ricevere informazioni su nuovi partecipanti alla sessione RTP mentre i più importanti *ReceiveStreamEvent* riguardano l'arrivo o l'identificazione di nuovi flussi.
Delle sottoclassi di *ReceiveStreamEvent* quelle più interessanti ed importanti sono il *NewReceiveStreamEvent*, lo *StreamMappedEvent* e l'*InactiveStreamEvent*.
Il *NewReceiveStreamEvent* viene lanciato dal RTPManager quando arriva un nuovo flusso RTP, attraverso il metodo *getReceiveStream()* otteniamo il flusso da cui possiamo ottenere il DataSource del flusso in arrivo.
Lo *StreamMappedEvent* viene lanciato quando un flusso di origine sconosciuta viene identificato mentre l'*InactiveStreamEvent* indica che su questo *ReceiveStream* hanno smesso di arrivare pacchetti dati e di controllo; il timeout passato il quale viene lanciata questa eccezione è 5 volte l'intervallo medio di tempo tra l'arrivo un pacchetto RTCP ed un altro per lo specifico *ReceiveStream*.

3.2 PlugIn e catene di Codec

Una catena di PlugIn è una serie di PlugIn (interfaccia definita dal JMF) che vengono collegati uno all'altro per decodificare passo per passo i dati arrivati dalla connessione; il nome Plugin identifica tutti quei componenti passivi che accettano in ingresso dati in un formato

specifico e ne effettuano la transcodifica in un altro formato oppure la presentazione.

La gestione lato client dello streaming che consiste nell' ascoltare la connessione,creare i codec necessari alla decodifica dei pacchetti ed infine effettuare la decodifica e la renderizzazione.

Tale gestione può essere fatta utilizzando un Player (classe definita dal Framework),estendendone uno oppure creando una propria catena di plugin.

Ad esempio una volta ricevuta la connessione è necessario effettuare la suddivisione dello stream media nelle sue singole tracce, di questo si occupa il Demultiplexer.

Il Decoder che decodifica i pacchetti ricevuti mentre il Renderer ne effettua il rendering; tutti questi componenti sono esempi di PlugIn JMF e corrispondono alle interfacce Demultiplexer, Codec, Renderer, Effect.

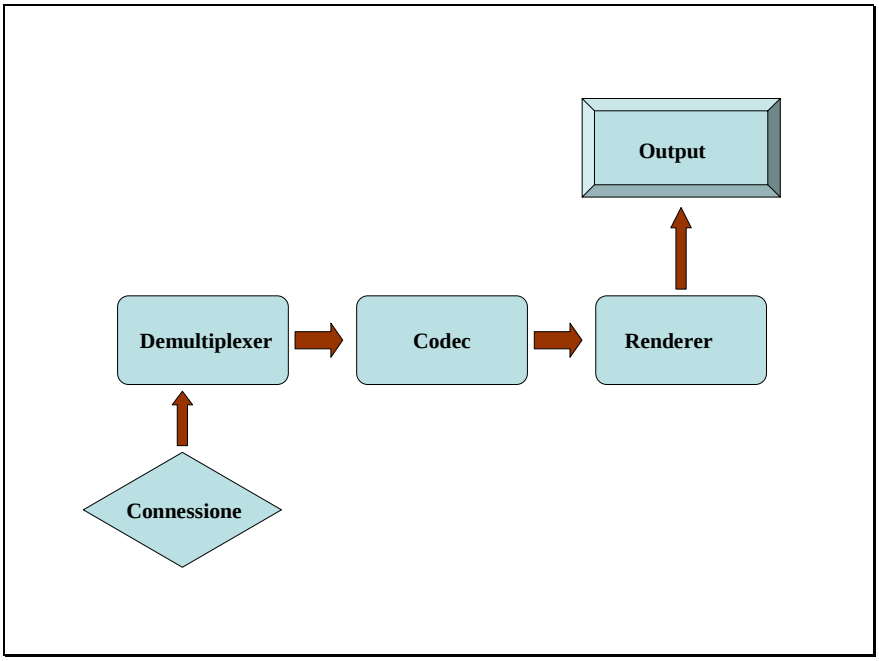


Figura 7: struttura di una catena di PlugIn

A questa catena base possono essere aggiunti diversi effetti e diversi codec, in base al formato del media, alla piattaforma ed al tipo di output desiderato.
Attraverso classi come la PlugInManager, che registra tutti i plugin installati su una piattaforma ed i loro formati di Ingresso/Uscita un Player è in grado di creare autonomamente la catena migliore per effettuare la renderizzazione video senza però permettere all'utente nessun grado di libertà nello specificare la catena.
Una classe che permette invece di utilizzare catene di PlugIn definite dall'utente è il Processor, classe che estende il Player.
Si noti fin d'ora che la configurazione dinamica della catena effettuata dal Player è molto utile, ma richiede tempo per essere portata a compimento, un overhead che non sempre è necessario, soprattutto se si conosce in anticipo il formato dello stream e di conseguenza se si conosce già la catena necessaria a decodificarlo.
Nello sviluppo dell'applicazione abbiamo tenuto conto di questo utilizzando una Custom Chain fissata, eliminando quindi l'overhead necessario per la configurazione della stessa.
La struttura interna del Player e del Processor è molto simile; in entrambi i casi un processo esegue le decodifiche del frame ricevuto passo per passo,.
In un simile ambiente, strutturato su un Thread singolo non è facile implementare un buffer che si situi fra un passo e l'altro della catena dato che il primo passo della catena (la lettura dei dati dalla

connessione) è sempre bloccante, quindi anche se fossero presenti frame nel Buffer il Thread si bloccherebbe rendendolo inutile.
Per questo motivo un' applicazione come quella che si vuole sviluppare che necessiti l'uso di un buffer non si può basare sulla semplice estensione della classe Player o Processor ma doveva deve gestire direttamente i plugin basandosi su un'implementazione concorrente.
Vediamo ora più nel dettaglio le funzionalità dei principali plugin:

3.2.1 Demultiplexer

Questa classe serve per separare un flusso nelle sue singole tracce componenti.
Il metodo più importante di questa classe è il metodo *getTracks()* che restituisce un'array di Tracks (classe che rappresenta la traccia).
Il metodo *start()* inizializza il demultiplexer ed è necessario chiamarlo prima che venga chiamato il metodo *readFrame()* sulle Tracks restituite.
Il metodo *readFrame()* della classe Track restituisce un Buffer che contiene un singolo Frame o una sua parte, questo metodo è bloccante e non ritorna fino a quando non viene ricevuto un Frame.

3.2.2 Codec

Un Codec è un plugin che processa un frame in ingresso e produce un frame in uscita; nel JMF non è solo una classe che trasforma il formato del buffer di ingresso in quello desiderato sul Buffer di uscita ma ogni operazione che viene eseguita su un Buffer in ingresso e produce un Buffer in uscita cade sotto la categoria di Codec.

Per poter utilizzare un Codec è necessario impostare il formato di ingresso attraverso il metodo *setInputFormat()* e quello di uscita attraverso il metodo *setOutputFormat()*; entrambi i metodi restituiscono il formato effettivamente scelto oppure *null* se il formato non è supportato.

E' possibile ottenere una lista dei formati in ingresso o uscita supportati dal Codec attraverso i metodi *getSupportedInputFormats()* e *getSupportedOutputFormats()*; il metodo *setInputFormat()* chiama automaticamente il metodo *setOutputFormat()* passandogli come parametro il primo della lista dei formati di output supportati.

Una volta settati i formati di Input ed Output è necessario chiamare il metodo *start()* per completare l'inizializzazione, a questo punto si può chiamare il metodo *process()* passandogli come parametri il Buffer che contiene il frame di Input ed un Buffer dove immagazzinare l' Output.

3.2.3 Renderer

L'interfaccia Renderer definisce un'entità che effettua il rendering di una singola traccia di un flusso su un determinato dispositivo.

E' possibile ottenere una lista dei formati in ingresso supportati dal Renderer attraverso il metodo *getSupportedInputFormats()*.

Una volta settati il formato di Input è necessario chiamare il metodo *start()* in modo da inizializzare correttamente il componente.

Il renderer è il passo finale di una catena di Processing quindi il suo metodo *process()* ha un solo Buffer in ingresso che contiene il frame di cui eseguire il Rendering.

3.3 Package Unibo

Il package Unibo è un package java creato dal DEIS che contiene classi che implementano i componenti necessari per creare e gestire le catene di PlugIn e permettere il controllo al livello del singolo frame.

Attraverso i componenti offerti da tale package è possibile implementare meccanismi di Buffering (come vedremo) oppure variare dinamicamente la stessa catena di PlugIn.

Una classe di importanza fondamentale è il CircularBuffer, che implementa l'astrazione di un buffer circolare che può essere interposto fra passi successivi della catena.

Gli elementi del CircularBuffer sono singoli Frame (istanze di ExtBuffer, classe che estende Buffer fornendo la capacità di gestire dati in formato nativo).

Questa classe gestisce al suo interno le problematiche legate alla gestione della mutua esclusione in caso di accesso concorrente.
La classe è strutturata come un buffer circolare con due puntatori: uno alla prossima cella in cui è possibile scrivere ed uno alla prossima cella da cui è possibile leggere.

3.3.1 package unibo.parser

Questo package contiene le classi che effettuano il Demultiplexing di un media.
La classe astratta *Parser* elenca i metodi caratteristici del Parser implementando quelli comuni a tutti e dichiarando astratti quelli che necessitano di implementazioni specifiche per ogni formato.
Le classi *RTPParser* e *QuickTimeParser*, sono due componenti che effettuano il Demultiplexing di un media nella forma di flusso RTP e file QuikTime.

3.3.2 package unibo.transcoder

Di questo package fanno parte le classi che effettuano la transcodifica (passaggio da un formato ad un altro).
L'interfaccia *Transcode* dichiara i metodi che devono essere implementati da ogni Trancoder e dichiara alcune costanti riguardanti il risultato della fase di transcodifica (PROCESS_FAILED, PROCESS_AGAIN, TRANSCODE_OK).
Ogni transcoder va configurato specificando il formato dei dati di ingresso.

3.3.3 package unibo.multiplexer

Questo package contiene classi che si occupano del mutiplexing dei frame appartenenti alle varie tracce del media.

Come per gli altri package esiste un'interfaccia base (*Multiplex*) che dichiara i metodi essenziali per ogni multiplexer.

Al momento l'unico multiplexer a disposizione è la classe *RTPMultiplexer* che crea un flusso che sarà poi possibile trasmettere via RTP.

Per inizializzare il componente è necessario specificare i formati di tutte le tracce in ingresso; tutti i frame delle tracce vengono multiplexati su un unico DataSource che può essere ottenuto con il metodo *getDataOutput().*

3.3.4 package unibo.rtp

Questo package contiene due classi RTPSender e RTPReceiver che incapsulano un'istanza di RTPManager e ne semplificano l'inizializzazione da parte dell'utente.

3.3.5 package unibo.thread

Le classi contenute in questo package realizzano i diversi stadi di una catena attiva per il processing dei frame delle tracce di un media basata su più Thread.

Ogni Thread si interfaccia con quello successivo attraverso un'istanza di CircularBuffer in cui frame possono essere inseriti o prelevati.
Al momento esistono tre classi, ParserThread, TranscodeThread e MultiplexerThread, che permettono di realizzare semplici catene di manipolazione.
Nell'implementazione attuale questi Thread eseguono la manipolazione su una singola traccia anche se sono previste future implementazioni multitraccia.

3.3.6 package unibo.chains

Di questo package fanno parte le classi che incapsulano componenti dei package parser, transcoder e multiplexer e realizzano catene di più componenti.
Non si tratta di componenti basati su più Thread ma di semplici incapsulamenti, in particolare senza nessun buffering al loro interno.

3.3.7 package unibo.server

Questo package contiene 3 semplici Server per l'invio di un contenuto multimediale.
La classe ClassicJMFServer è costruita attraverso un Processor del JMF, è quindi un normale server JMF.
SingleThreadServer è invece un componente che sfrutta i componenti del package unibo sfruttando le classi H263toRTPChain e RTPMultiplexer.

Questo server non effettua bufferizzazione interna e considera solo video monotraccia.
La classe MultiThreadServer implementa una catena simile a quella precedente con stage basati su più thread che comunicano tra loro attraverso CircularBuffer a cui accedono in maniera concorrente e mutuamente esclusiva.

3.3.8 package unibo.proxy

Questo package contiene classi che permettono l'avvio di un Proxy per i pacchetti RTP.
La classe ClassicJMFProxy realizza il proxy tramite le classi e le modalità fornite dal JMF, cioè attraverso l'uso del Processor.
MultiThreadProxy crea una struttura basata su un Parser RTP ed un Multiplexer, entrambi in versione threaded.
Il Parser estrae i frame dal DataSource e li inserisce in un CircularBuffer mentre il Multiplexer li estrae dal CircularBuffer e li inoltra.
Entrambi accedono al CircularBuffer in maniera concorrente e mutuamente esclusiva.

CAPITOLO 4

ANALISI E PROGETTAZIONE DELLA POLITICA DI BUFFERING

4.1 Analisi dei requisiti

Lo scenario descritto nel primo capitolo ci porta a dover cercare di risolvere il problema della perdita di pacchetti durante l'handoff senza degradare la presentazione del contenuto multimediale all'utente.

La soluzione a questo problema è la creazione un componente che permetta la bufferizzazione di parte del flusso multimediale sul client, in modo che nonostante la perdita momentanea di connessione la presentazione possa essere fluida e continua.

Dato che la nostra è un'applicazione mobile, che si basa quindi su una connessione 'volatile' un requisito utile è che il Buffer sia dinamico,in modo da poterne variare la lunghezza in base alla qualità della connessione, in particolare ridurla se la qualità aumenta o aumentarla se la qualità diminuisce.

Oltre a questo è necessario un metodo di comunicazione con il Server e di controllo del flusso, in modo da poter richiedere al server di rallentare la velocità a cui invia i Frame (se dobbiamo ridurre la dimensione del Buffer) o aumentarla (se ad esempio dobbiamo riempire velocemente il buffer del client).

Un requisito molto importante è la QoS, l'utente deve vedere lo stream come se la disconnessione non fosse mai avvenuta ed inoltre la qualità del video non deve peggiorare.

4.2 Progetto dei componenti

Dopo una analisi del JMF e delle classi Player e Processor abbiamo stabilito che per risolvere il problema posto dalla bufferizzazione non sarebbe bastato estendere una delle due classi, perchè entrambe le classi hanno al loro interno una catena di PlugIn basata su un singolo Thread quindi anche riuscendo ad inserire al loro interno un Buffer alla caduta della connessione il Thread verrebbe sospeso, rendendo inutile la presenza dei frame memorizzati.
L'implementazione del JMF (paradigma event-driven) implica la presenza di un Thread principale di controllo, che riceva dal RTPManager le segnalazioni della avvenuta connessione ed il DataSource.
Non potendo utilizzare un Player o un Processor l'unica possibilità era ricostruire passo a passo una catena di plugin che permettesse la decodifica ed il Rendering dei frame sull'output scelto, prospettando due scelte possibili: creare i vari Codec della catena in maniera automatica (seguendo l'esempio del Player) attraverso l'uso della classe Manager PlugInManager, che attraverso i suoi metodi può restituire tutti i PlugIn dei vari tipi (Renderer, Demultiplexer, Codec, Effect) in grado di elaborare un dato Formato in ingresso oppure basarsi su una Custom Chain, una catena costruita su un formato specifico, scelta molto efficiente in termini di OverHead ma meno trasparente.

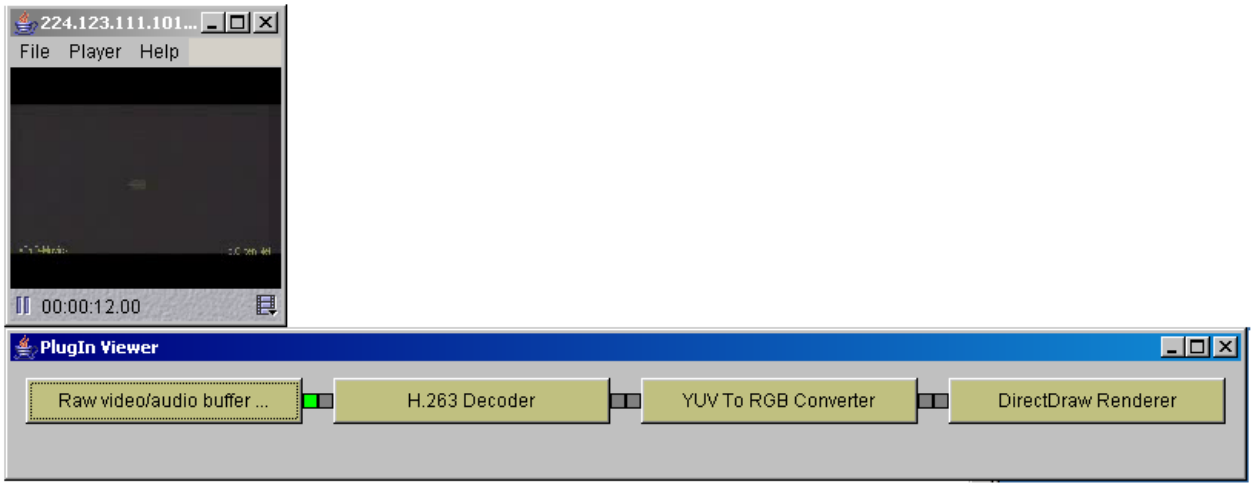
Si noti però che la prima scelta è decisamente costosa in termini di OverHead (si vedano in seguito i risultati sperimentali) dati i vari controlli e cicli che sono necessari per strutturarla.
La scelta più semplice sarebbe quella di affidare la costruzione della catena al PlugInManager, scelta più trasparente, sacrificando l'OverHead e l'efficienza, ma approfondendo le problematiche e l'analisi del FrameWork stesso abbiamo optato per la CustomChain.
Le intuizioni che ci hanno portato alla scelta sono due: in uno scenario come il VoD potrebbe farsi carico il middleware (MW) stesso di scaricare sul Client il Software necessario per decodificare correttamente il video nel caso esso non si trovi già sul Client e soprattutto il supporto attuale per lo streaming su RTP del JMF è limitato ai formati JPEG_RTP,H261_RTP,H263_RTP, limitando notevolmente la scelta possibile del formato per la trasmissione del Video.
Data la sua maggior compressione ed efficienza rispetto agli altri abbiamo deciso di utilizzare il formato H263, che permette un uso migliore della banda disponibile (condizione anche questa critica nello streaming video).

4.2.1 Catena dei Plugin

A questo punto abbiamo (grazie al programma JMStudio fornito dalla Sun insieme al FrameWork) analizzato la Catena dei plugin necessari per decodificare ed eseguire il rendering di un video trasmesso in formato H263 (si noti che la trasmissione avviene in formato H263, si

occupa il server della eventuale transcodifica dal formato iniziale del video a quello di trasmissione)

Il programma ci mostra grazie attraverso una gradevole interfaccia la catena dei plugin necessaria:



(figura 8) PlugIn Viewer del JMStudio che mostra i PlugIn utilizzati dal processor che effettua il Rendering del video

- com.sun.media.parser.RawBufferParser
- com.sun.media.codec.video.vh263.NativeDecoder
- com.sun.media.codec.video.colorspace.YUVToRGB
- com.sun.media.renderer.video.DDRenderer

Analizzando questa catena ci siamo accorti di due complicazioni: innanzitutto essa fa uso di un Renderer basato sulle librerie irect (DDRenderer) ed inoltre fa uso di un Decoder che utilizza del codice nativo, questo perchè sulla macchina era installata la versione Windows Performance Pack del JMF che sfrutta codice binario nativo per aumentare l'efficienza del framework.

Entrambi questi problemi impedivano la portabilità di questa catena su tutte le piattaforme, quindi si è dovuto fare un passo successivo, trovare una catena simile che però fosse completamente portabile.
Il problema del codice nativo è stato risolto utilizzando una classe presente nella versione CrossPlatform del FrameWork, la JavaDecoder presente nel package com.ibm.media.codec.video.h263.
Questa classe non utilizza librerie esterne in codici ottimizzati per la piattaforma ma utilizza codice completamente java, essendo quindi completamente portabile.
Utilizzando questa classe inoltre il Codec com.sun.media.codec.video.colorspace.YUVToRGB diventa inutile,in quanto il formato di uscita della JavaDecoder è già RGB.
Il problema del Renderer era quello di più facile soluzione, test effettuati su una macchina Solaris hanno mostrato che nella catena al posto del DDRenderer veniva utilizzata la classe AWTRenderer(com.sun.media.renderer.video.AWTRenderer), classe meno efficiente ma al contrario di quella basata sulle DirectX è completamente portabile su tutte le piattaforme.
A questo punto abbiamo ottenuto la seguente catena di PlugIn:

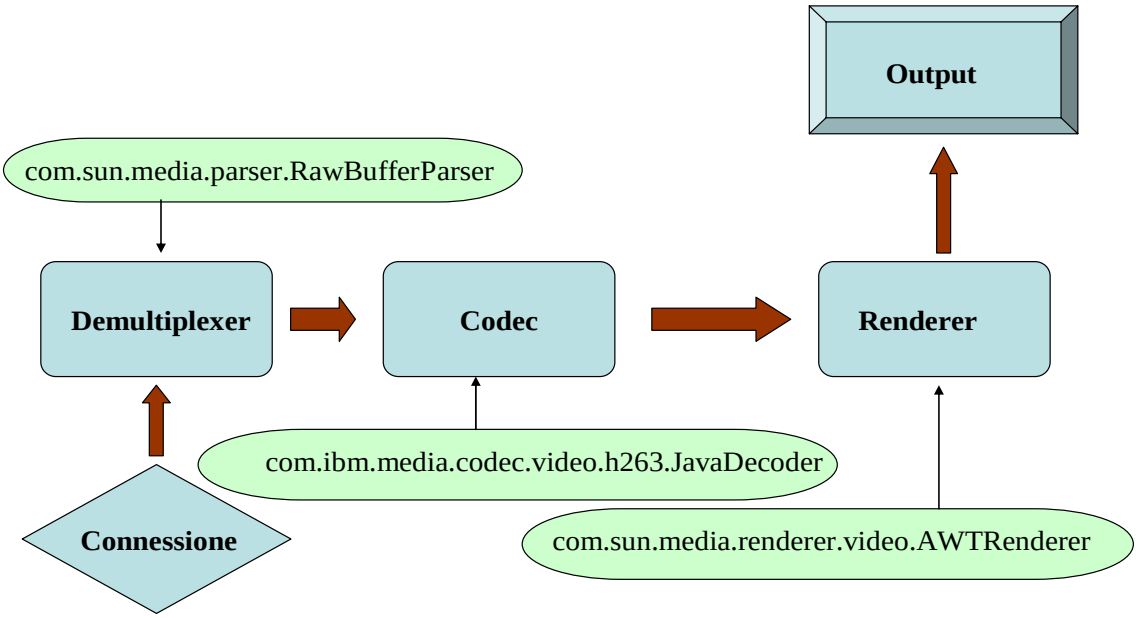


Figura 9: catena dei PlugIn per un ambiente Cross-Platform,utilizza classi senza implementazioni native

A questo punto dati i requisiti e la struttura del JMF si è pensato di strutturare il Client su più Thread concorrenti, un Thread di controllo, uno di lettura dati dalla connessione ed uno di rendering.

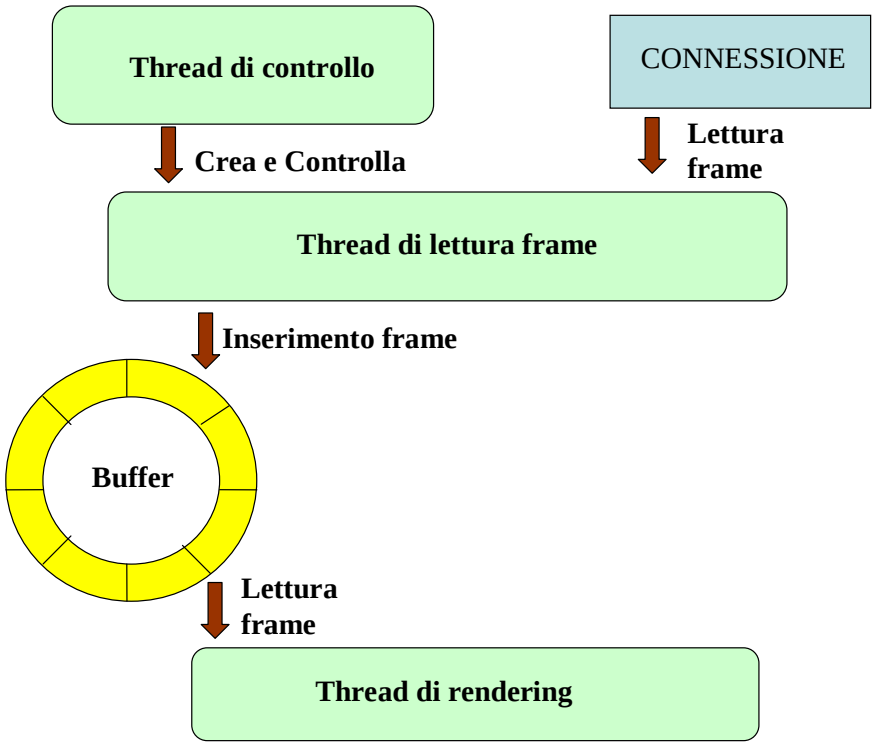


Figura 10: schema dei thread del client

4.2.2 Componente buffer

All'interno del package unibo è già fornita l'astrazione di un buffer circolare, la classe CircularBuffer, la cui struttura è formata da una coda di lunghezza fissa e che fornisce metodi per inserire e leggere frame dalla coda.

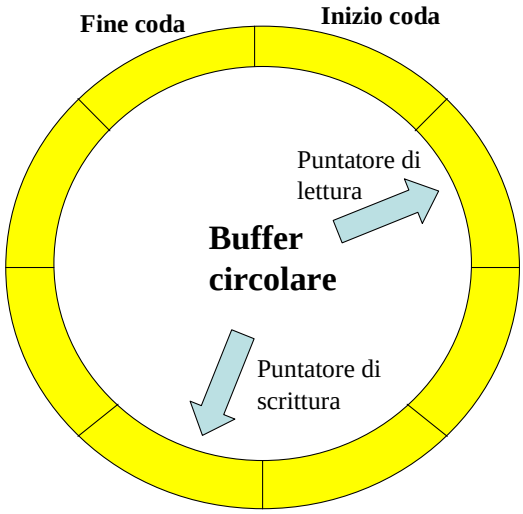


Figura 11: buffer circolare

Abbiamo quindi deciso di progettare il componente che implementa il buffer a partire da quello del package unibo.

Il componente deve fornire dei metodi che permettano di interrogarne lo stato (soprattutto se è pieno,vuoto o a metà), inoltre deve gestire al suo interno la comunicazione con il server e la sua richiesta di aumento o riduzione della velocità (in particolare se è pieno è comodo che venga automaticamente chiesto al server di ridurre la velocità oppure che venga chiesto automaticamente di aumentarla se il Buffer è pieno).

Inoltre è necessario prevedere un costruttore che forzi la dimensione della coda ad un certo numero di frame ed un altro invece che la forzi a contenere una determinata durata di tempo.

La gestione interna della comunicazione con il server potrebbe sembrare assegnare ad un componente delle responsabilità che non gli competono e che dovrebbero essere invece assegnate al Thread di controllo ma bisogna anche tenere conto, come vedremo tra poco, che per effettuare la riduzione della coda è necessario che il componente si coordini automaticamente con il server.

A questo punto l'unico metodo che è necessario implementare dato che i metodi di inserimento e lettura sono già implementati dal CircularBuffer è quello che permette di modificare la lunghezza della coda.

La trattazione di questo problema va divisa in due casi, cioè l'allungamento o la riduzione della lunghezza della coda.

L'allungamento è il problema più semplice, basta semplicemente aggiungere in coda i Buffer per i frame mancanti e riorganizzare i frame già presenti nella coda ed i puntatori ai Buffer da leggere e sui cui si può scrivere.

Il caso della riduzione è più complesso, infatti bisognerebbe creare una politica che permetta di decidere quali frame scartare ed eventualmente richiedere al Server il re-invio dei frame scartati.

Esiste poi un'altra possibilità volta a limitare il numero delle ritrasmissioni e migliorare l'uso delle risorse: fare si che la lettura consumi la coda (provocandone la riduzione).

Per realizzare ciò il client utilizzando il canale di controllo richiede al server di ridurre la velocità di invio dei frame (e quindi, dato che la velocità del processo di rendering è indipendente da quella di invio del

server, questo mantiene la lunghezza della coda costante, ma riduce il numero di frame presenti) ed una volta che il numero di frame presenti nella coda sia uguale od inferiore a quelli richiesti dall'utente è poi possibile ridurre la coda alla lunghezza desiderata, aggiornando i vari puntatori e chiedendo al server di riportare la sua velocità di trasmissione a quella normale.

4.2.3 Processo di controllo

Il Processo di Controllo deve permettere di registrarsi presso l'RTPManager per ricevere notifiche di connessioni su determinate porte e da determinati indirizzi; inoltre deve gestire le connessioni e le disconnessioni dei Server.

Per gestire la disconnessione stiamo stati costretti a basarci sul FrameWork, infatti il Processo che legge dati dallo stream nel caso non ci siano dati a disposizione viene messo in wait fino a quando non arriva un nuovo dato (la lettura dei frame dalla connessione è bloccante); Nel caso della disconnessione il Processo rimarrebbe in questo stato per un periodo indefinito quindi il Processo di controllo deve testare lo stato del lettore nel caso di una riconnessione e rimpiazzarlo con un altro se il suo stato è interrotto.

Inoltre il Processo di controllo deve intercettare gli eventi di BYE (Partecipante che lascia la sessione; nel nostro caso disconnessione del Server) ed in questo caso fermare il lettore.

4.2.4 Processo di lettura

Il Processo di lettura (che viene creato dal controllore all'arrivo di una connessione) si deve occupare di demultiplexare il DataSource nelle tracce che lo compongono e poi di leggere i frame ed inserirli nel Buffer fino o al raggiungimento di un frame EOM (End Of Media) nel qual caso si deve fermare oppure fino a che non viene fermato dall'esterno.

Inoltre abbiamo deciso di permettergli di creare il Processo di rendering al primo riempimento del Buffer; questo perché era necessario che il Renderer fosse creato una volta che il Buffer fosse pieno e questa responsabilità avrebbe appesantito di molto il Processo di controllo e logicamente non gli dovrebbe appartenere; naturalmente questa funzione del Lettore deve poter essere selezionabile dall'utente (in questo caso il Processo di controllo). Nella nostra situazione questo è utile perché permette alla disconnessione di fermare il Lettore e sostituirlo con un altro utilizzando sempre lo stesso Renderer.

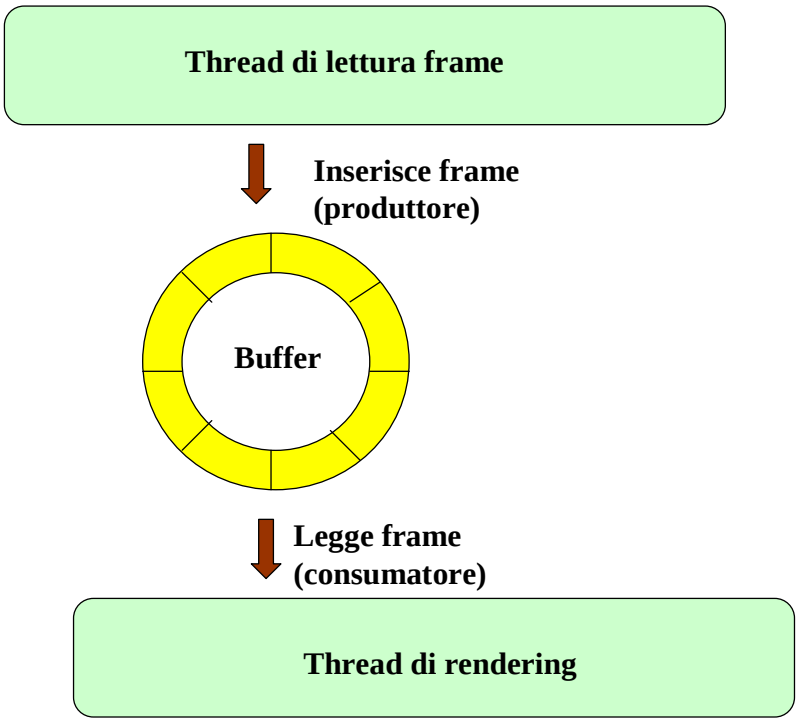


Figura 12: schema produttore/consumatore

4.2.5 Processo di Rendering

Il processo di Rendering alla creazione deve costruire il resto della catena, cioè il JavaDecoder ed l'AWTRenderer.

Una volta creata la catena deve cominciare un ciclo di lettura e decoding dei frame dal Buffer e deve eseguirne il Rendering.

Una volta eseguito il Rendering di un frame deve mettersi in stato di sleep per un numero di millisecondi pari a 1000/framerate del video che si sta mostrando all'utente.

Questo wait permette di mantenere il framerate, cioè mostrare tanti frame al secondo quanti sono quelli previsti dal filmato.

Inoltre al raggiungimento dell’ EOM il renderer si deve fermare e deve essere possibile fermare dall’esterno il processo.

CAPITOLO 5
IMPLEMENTAZIONE DELLA POLITICA DI BUFFERING

5.1 CircularBuffer

La nostra soluzione si basa CircularBuffer del package unibo, ossia un buffer ad anello dotato di due puntatori,uno (tail) al prossimo slot libero ed uno (head) al prossimo slot le cui lettura è possibile.

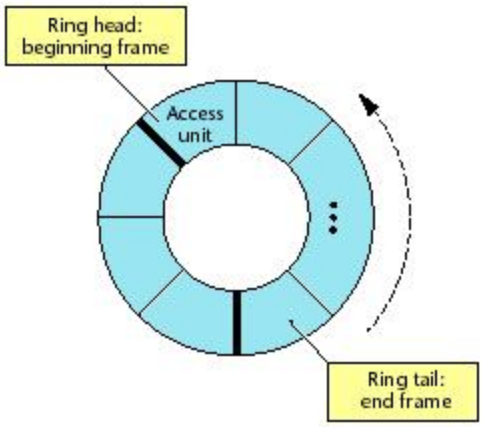


Figura 13: struttura di un Buffer Circolare con puntatori di Head e Tail, preso da [2]

Nel java il CircularBuffer è realizzato attraverso un array statico di Buffer, in cui inizialmente i puntatori head e tail puntano entrambi al primo elemento dell'array.

All'avvenuta scrittura (chiamata del metodo getEmptyBuffer() che restituisce un oggetto Buffer in cui inserire i dati e writeReport() che riporta l'avvenuta scrittura e segnala il Buffer come leggibile) il puntatore di Tail viene spostato all'elemento successivo,e così via,fino a raggiungere nuovamente il puntatore di head (nel caso il puntatore superi la dimensione dell'array automaticamente viene riportato all'inizio), a quel punto la chiamata al metodo CanWrite() restituisce

false, indicando che un'ulteriore scrittura nel CircularBuffer è impossibile.
Ad ogni lettura effettuata (chiamate successive dei metodi Read() e ReadReport()) il puntatore di head viene spostato all'indice successivo,fino a raggiungere il puntatore di tail, a questo punto la chiamata al metodo CanRead() restituisce false, indicando l'impossibilità di una ulteriore lettura.
Questa soluzione ha il problema di essere statica (basata su un'array) e di necessitare nel costruttore del numero dei Buffer (ossia dei Frame) che deve contenere,impedendo una creazione basata ad esempio sulla lunghezza temporale totale del CircularBuffer.
Per rispettare i vincoli di progetto questa classe doveva essere modificata, purtroppo una simile modifica non poteva essere implementata facendo una semplice sottoclasse del CircularBuffer quindi è stato necessario creare una nuova classe, il QueableCircularBuffer, che rispondesse ai nostri requisiti.

5.2 QueableCircularBuffer

5.2.1 Struttura, metodi e costruttori

La nostra implementazione del CircularBuffer è basata sulla stessa struttura interna del CircularBuffer, in particolare anche essa ha al suo interno i puntatori di lettura e scrittura e si occupa della gestione dell'accesso concorrente e della mutua esclusione.
Per prima cosa si è pensato di modificare il sistema in cui vengono inseriti i dati all'interno della coda, creando un unico metodo

Write(Buffer b) che copiasse (per riferimento) il Buffer passato come parametro all'interno di uno slot libero della coda invece di utilizzare i due metodi offerti dal CircularBuffer.
Questa strada è stata interrotta quando ci siamo accorti che un simile metodo influiva negativamente sulle prestazioni, quindi abbiamo mantenuto la compatibilità con la classe del JMF mantenendo i metodi *getEmptyBuffer()* e *writeReport().*
E' stato necessario modificare anche il costruttore, infatti è stato aggiunto un costruttore capace di costruire un QueableCircularBuffer passandogli come parametro un indicatore della lunghezza in nanosecondi del Buffer.
Naturalmente il componente non può sapere in anticipo la durata del singolo frame del video (che verrà passato solo in seguito e che è variabile in base al formato del video) quindi il costruttore crea una coda di due soli elementi,sarà il metodo *writeReport()* chiamato quando verranno inseriti i frame nella coda che calcolerà la lunghezza media dei frame e riallocherà la coda in modo da poter contenere tanti frame la cui durata complessiva sia uguale a quella desiderata dall'utente.
Inoltre,dato che in fase di progetto avevamo deciso di far cominciare il rendering solo quando il buffer era stato riempito sono stati inseriti dei metodi che permettono all'utente di conoscere parzialmente lo stato della coda; in particolare un metodo che permette di sapere se è piena (*IsBufferFull()*).

E’ stato inoltre creato un metodo che permette di inviare messaggi di controllo al Server sfruttando la connessione di controllo specificata dall’utente.
Questo metodo è necessario per poter implementare il controllo della velocità a cui il Server invia i Frame (naturalmente utilizzando un server appropriato, creato a partire dal SingleThreadServer del package unibo.server),utilità necessaria come verrà mostrato in seguito (è stato quindi necessario creare dei costruttori che avessero come parametri i dati della connessione di controllo,cioè l’indirizzo IP del Server e la porta di controllo).

Figura 14: schema della classe QueableCircularBuffer

5.2.2 Gestione dinamica della lunghezza della coda

Perchè la presenza del QueableCircularBuffer sia utile in caso di disconnessione dal server è necessario che la lunghezza temporale del buffer sia maggiore del tempo necessario al server per riconnettersi, tempo che dipende da numerosi fattori tra cui la qualità stessa della connessione alla rete, quindi è necessario che la lunghezza temporale del buffer non sia fissata a priori,ma possa essere variata al variare delle condizioni in cui si esegue lo streaming,occorrono quindi metodi per poter variare dinamicamente la lunghezza della coda,metodi che sono stati inseriti all'interno dell'implementazione del QueableCircularBuffer.

Inizialmente abbiamo sostituito l'array statico di Buffer con un ArrayList dinamico,in modo da poterne variare la lunghezza tramite il suo metodo *add()*.

Poi è stato creato il metodo *SetSize(int n)* che implementa le politiche di variazione (il parametro comunicato dall'utente indica la nuova lunghezza della coda).

Esistono due possibili scenari nella variazione della lunghezza, il primo scenario considera che la lunghezza desiderata della coda sia maggiore di quella attuale,il secondo che sia inferiore.

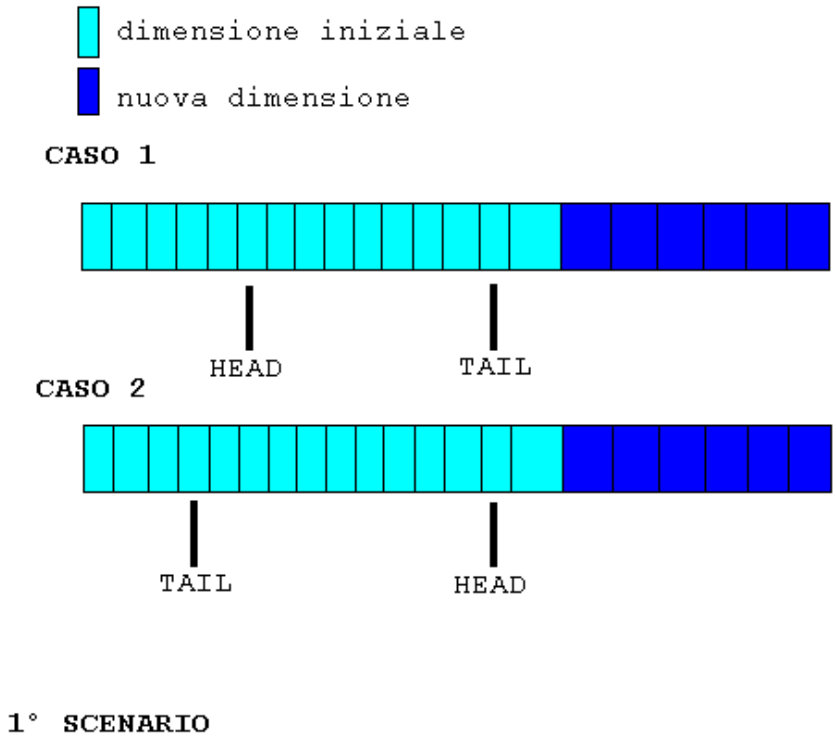


Figura 15: schema dei casi del 1° scenario

Nel primo scenario esistono due casi: se il puntatore di scrittura è maggiore di quello di lettura basta allargare l'ArrayList inserendo alla fine di esso i nuovi oggetti Buffer ed aggiornare il campo che indica la dimensione della coda,nel caso contrario (tail minore di head) è necessario spostare dall'inizio dell'ArrayList alla coda tanti Buffer da riempire completamente la parte dell'ArrayList che è stata aggiunta ed in seguito aggiornare Tail in modo da puntare la prima posizione libera della coda.

In questo scenario è necessario chiedere al Server di aumentare la velocità con cui invia i frame, utilizzando il metodo di controllo introdotto in precedenza.

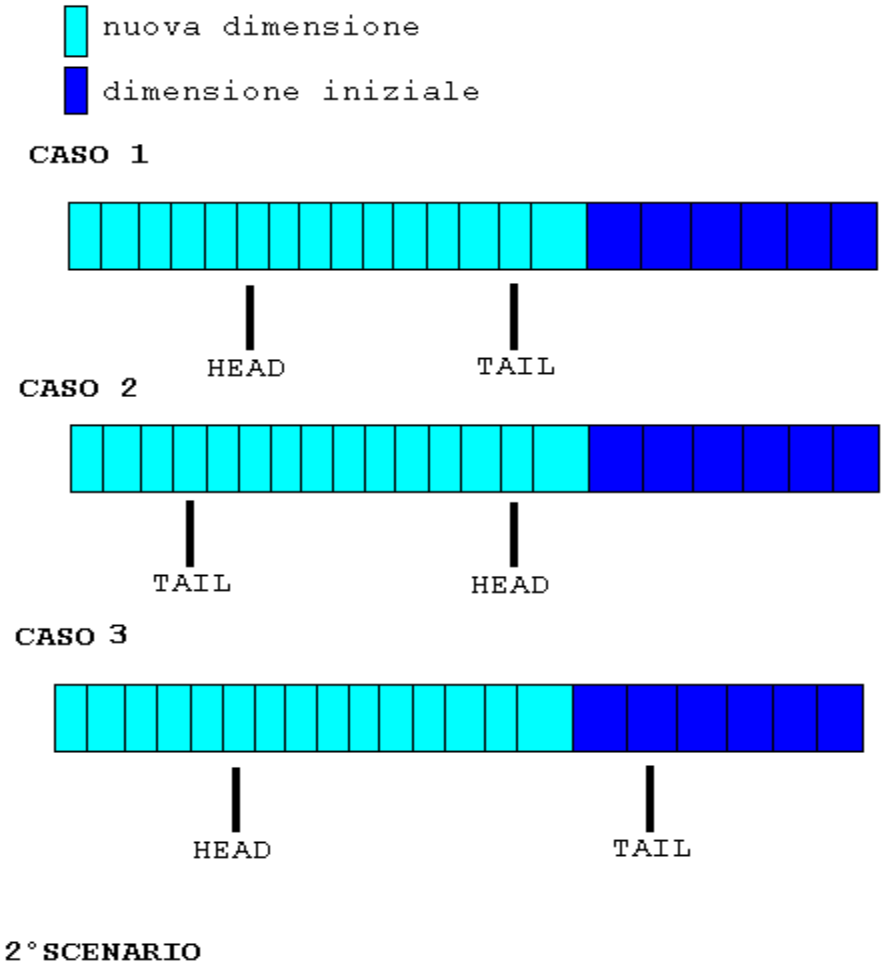


Figura 16: schema dei casi del 2° scenario

Nel secondo scenario l'utente desidera una coda di lunghezza inferiore a quella desiderata (ad esempio perchè si trova in una zona in cui la qualità della connessione wireless è migliore rispetto a quella precedente).

Se il puntatore di lettura è minore od uguale a quello di scrittura (primo caso) e Tail non si trova nella zona da ridurre possiamo effettuare la riduzione tranquillamente, in quanto in fondo alla coda abbiamo Buffer vuoti o già letti.

Se Tail si trova nella zona da rimuovere (terzo caso) il processo è più problematico, la soluzione attuata consiste nel chiedere al server di

ridurre la velocità a cui invia i dati in modo da portarci nuovamente al caso 1, a quel punto il server può ricominciare ad inviare i Frame alla velocità iniziale.
La soluzione del caso 2 è simile a quella del caso 3, si chiede al server di ridurre la velocità a cui invia i Frame fino a quando i Buffer contenenti dati ancora non utilizzati siano inferiori o uguali a quelli che sia possibile contenere nella coda di nuova lunghezza, solo allora viene creata una nuova coda ed i Buffer vengono allocati all'inizio di essa,e vengono riaggiornati i puntatori di Head e Tail.
A questo punto si richiede al Server di ricominciare a mandare i dati alla velocità standard.
Naturalmente i metodi di variazione dinamica della lunghezza della coda sono inutili se il server non è in grado di modulare la velocità a cui manda i Frame.
Nel caso non si disponesse di un simile componente una soluzione potrebbe essere rallentare o accelerare impercettibilmente la velocità di PlayBack dello Stream, soluzione comunque molto limitata in quanto non è possibile aumentare o ridurre oltre un certo limite il FrameRate del video senza che l'utente se ne accorga e la qualità del video non ne venga rovinata oppure frapporre un proxy fra client e server.

5.3 Struttura del Client

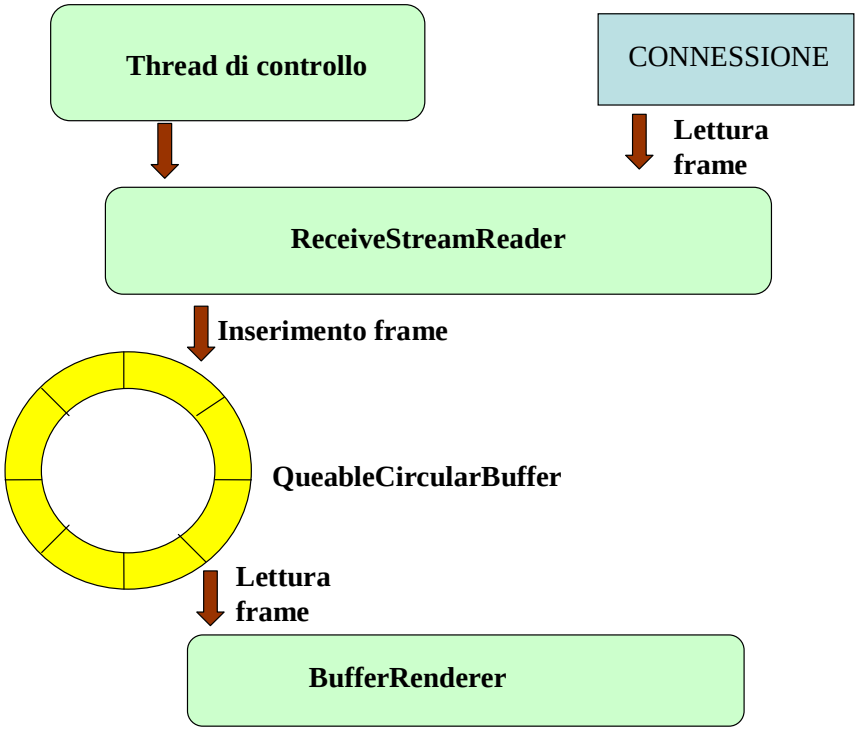


Figura 17: schema della struttura su più thread del Client

La Figura 17 mostra la struttura che è stata creata per il Client, tutte le classi che formano questa struttura (ad eccezione del QueableCircularBuffer e del controllore) sono estensioni della classe Thread per poter implementare un componente concorrente.

Purtroppo in fase implementativa ci siamo accorti di una incompatibilità della versione Cross Platform del JMF con quella nativa, incompatibilità che rende impossibile il funzionamento della catena non nativa su una macchina in cui sia installato un performance pack del FrameWork, incompatibilità dovuta al codice sorgente della classe JavaDecoder: infatti questa classe verifica se è possibile caricare in memoria la libreria “jmvh263” (che è presente nelle

versione nativa), se il caricamento è possibile la classe JavaDecoder lancia una eccezione e termina la sua esecuzione (probabilmente è un metodo per “suggerire” al programmatore di usare codice nativo).
Questo problema non ci interessa se limitiamo la nostra azione al campo dei PDA, per cui non esiste una versione nativa e che non sono in grado di caricare le librerie ma potrebbe costituire un problema se vogliamo usare la stessa catena su sistemi diversi, magari sui quali sia presente la versione nativa del FrameWork.
Naturalmente si può fare la stessa considerazione della CustomChain, cioè che sia il MiddleWare a preoccuparsi di fornire il Software (e quindi la catena) adatti ad eseguire il rendering sul Client, naturalmente questo comporta che il MiddleWare debba sapere se sul sistema è già installata o no una versione del FrameWork oppure se esiste una versione nativa di esso per la piattaforma considerata.
Questo problema può essere risolto anche implementando un meccanismo di controllo che si accorga di questa situazione ed adatti la catena di conseguenza.
Dato il minimo OverHead che questo comportava abbiamo deciso di usare la 2° soluzione, cioè di creare un meccanismo che monitorasse il sistema e potesse scegliere la catena più adatta.
A questo punto il componente ha a disposizione due catene, una se si trova a dover funzionare su un’ambiente in cui sia installata la versione Cross-Platform del FrameWork (fig) ed una se ha a disposizione un’installazione nativa dello stesso(fig).

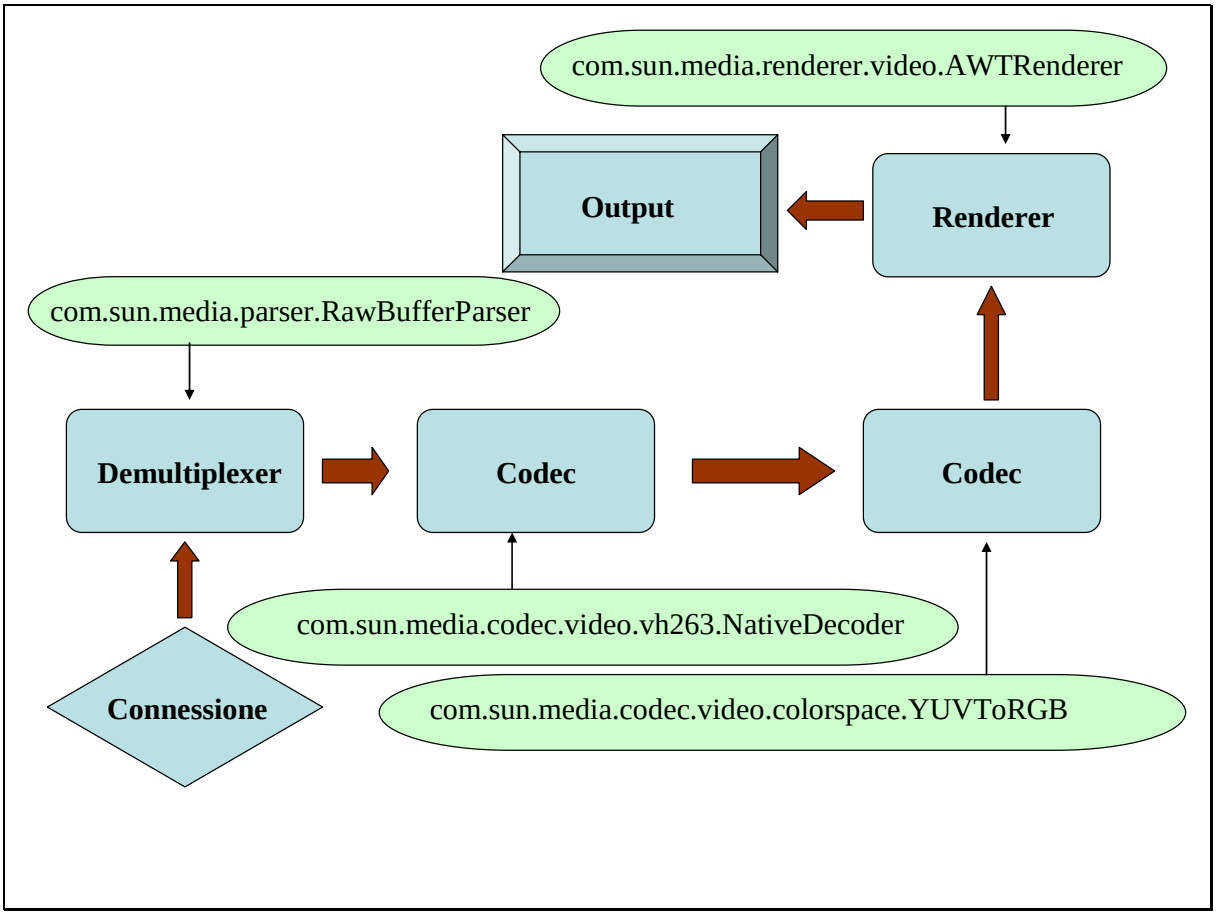


Figura 18: catena dei PlugIn per un ambiente in cui sia installata la versione nativa del FrameWork

5.3.1 Thread di controllo : RTPclient

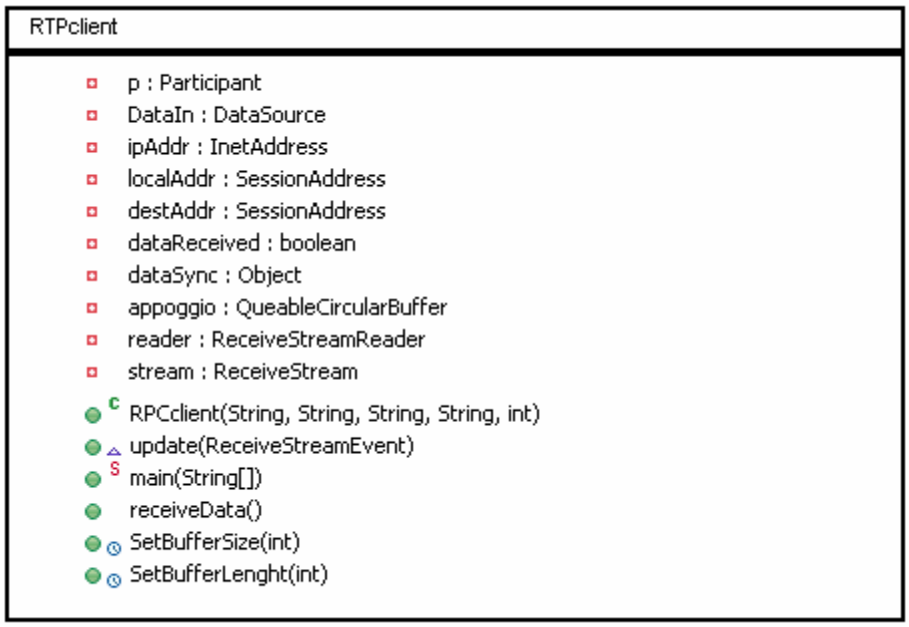


Figura 19: schema della classe RTPclient

Il thread di controllo è stato implementato nella classe RTPclient.
Questa classe fornisce innanzitutto un *main()* che permette di lanciare il Client, nell'implementazione corrente questo main ha 5 argomenti: la lunghezza desiderata del Buffer (in numero di frame),l'indirizzo IP del Server, la porta RTP del Server, la porta di controllo del Server e la porta locale a cui il Client si deve agganciare.
Per prima cosa il *main()* crea un QueableCircularBuffer e un oggetto della classe RTPclient, che ottenuto un riferimento ad un'istanza del RTPManager inizializza la sessione e il target(l'indirizzo del Server e la sua porta RTP) e si registra come ascoltatore degli eventi di Sessione e di flusso.
Poi il thread rimane in attesa fino all'arrivo di un nuovo Stream.
A questo punto possono esserci due casi: o il server invia prima i dati dello stream e poi si identifica tramite l'RTCP oppure il contrario.
Nel primo caso è subito possibile ottenere il DataSource dallo stream e viene subito creato il thread che eseguirà la lettura dei frame dal flusso (e gli viene passato come argomento il QueableCircularBuffer in modo che ci possa immagazzinare i frame estratti dal flusso).
Nel secondo caso è invece necessario aspettare l'arrivo dei frame RTP per poter ottenere il DataSource.
Inoltre questa classe è anche un ascoltatore dell'evento di flusso inattivo (*InactiveStreamEvent*), questo evento viene lanciato dal RTPManager quando il flusso dati è stato fermato o non arrivano dati sul flusso per un lungo periodo di tempo.

Questa classe fornisce anche due metodi che permettono di modificare la lunghezza della coda del QueableCircularBuffer, metodi pubblici a disposizione dell'utente.

5.3.2 ReceiveStreamReader

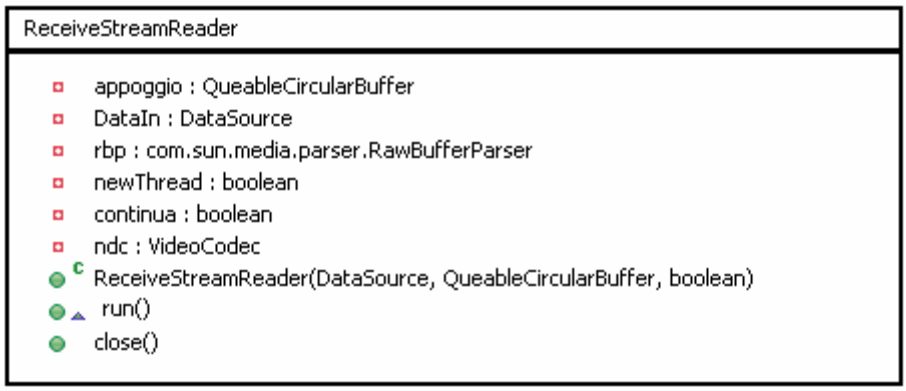


Figura 20: schema della classe ReceiveStreamReader

Questa classe che estende *Thread* rappresenta il thread che effettua il demultiplexing del flusso, estrae i frame dalla traccia e li inserisce nel QueableCircularBuffer.

Innanzitutto il costruttore necessita di 3 argomenti: il DataSource da cui estrarre i frame, un QueableCircularBuffer in cui immagazzinare i frame estratti ed un booleano che indica se creare o no il successivo passo della catena, cioè il Thread di rendering.

Infatti è necessario creare il Thread di rendering una volta che il QueableCircularBuffer sia stato riempito in modo da poter garantire la qualità richiesta dall'utente; abbiamo deciso per una questione di responsabilità che fosse questo thread ad occuparsi di monitorare il QueableCircularBuffer e creare il Renderer, ma abbiamo deciso di

fornire all’utente la possibilità di bloccare questa funzione e di creare il renderer a parte.
Questo è possibile perché in realtà il ReceiveStreamReader ed il Renderer non hanno bisogno di conoscersi, basta che condividano lo stesso QueableCircularBuffer in un approccio Produttore / Consumatore e questo è possibile sia se il Renderer viene creato dal ReceiveStreamReader (passandogli come argomento il suo QueableCircularBuffer) oppure dal thread di controllo passandogli come argomento lo stesso QueableCircularBuffer passato al ReceiveStreamReader.
Uno dei due altri metodi forniti da questa classe è il metodo *run()* che ridefinisce il metodo omonimo della classe Thread.
Questo metodo costruisce innanzitutto il Parser (*RawBufferParser*) e da quello estrae le tracce che compongono il flusso; poi comincia un ciclo infinito in cui estrae i frame dalla traccia e li inserisce nel QueableCircularBuffer.
Se il frame è un EOM (cioè indica che la trasmissione del contenuto multimediale è terminata) il ciclo viene fermato.
Inoltre appena il QueableCircularBuffer viene riempito per la prima volta il ReceiveStreamReader si occupa di creare il Thread di rendering ed il Frame (*java.AWT.Frame*) ed il Panel (*java.AWT.Panel*) su cui verrà visualizzato il contenuto multimediale.
L’accesso mutuamente esclusivo al QueableCircularBuffer viene eseguito dal seguente codice:

```
synchronized(appoggio)
                    {
                        while(!appoggio.canWrite()) {
                            try {
                                    appoggio.wait();
                                }
                            catch (Exception e) { }
                            }
                        In=(Buffer)appoggio.getEmptyBuffer();
                        appoggio.notifyAll();}
```

Dove “appoggio” è l’istanza del QueableCircularBuffer.

Come si vede se il QueableCircularBuffer è pieno (*canWrite()* vale *false*) il thread si ferma in attesa o del timeout o di essere svegliato dal Renderer, lasciando libero il lock sul QueableCircularBuffer in modo che il renderer possa accedervi.

Una volta risvegliato prende l’istanza di Buffer su cui scrivere e sveglia tutti i thread in attesa sul QueableCircularBuffer prima di lasciare la sezione critica.

Il *writeReport()* viene similmente chiamato all’interno di una sezione critica in modo da segnalare l’avvenuta scrittura.

L’unico altro metodo contenuto nella classe è un metodo syncronized *close()* che non fa altro che terminare il ciclo e chiudere la traccia ed il Parser.

5.3.3 BufferRenderer

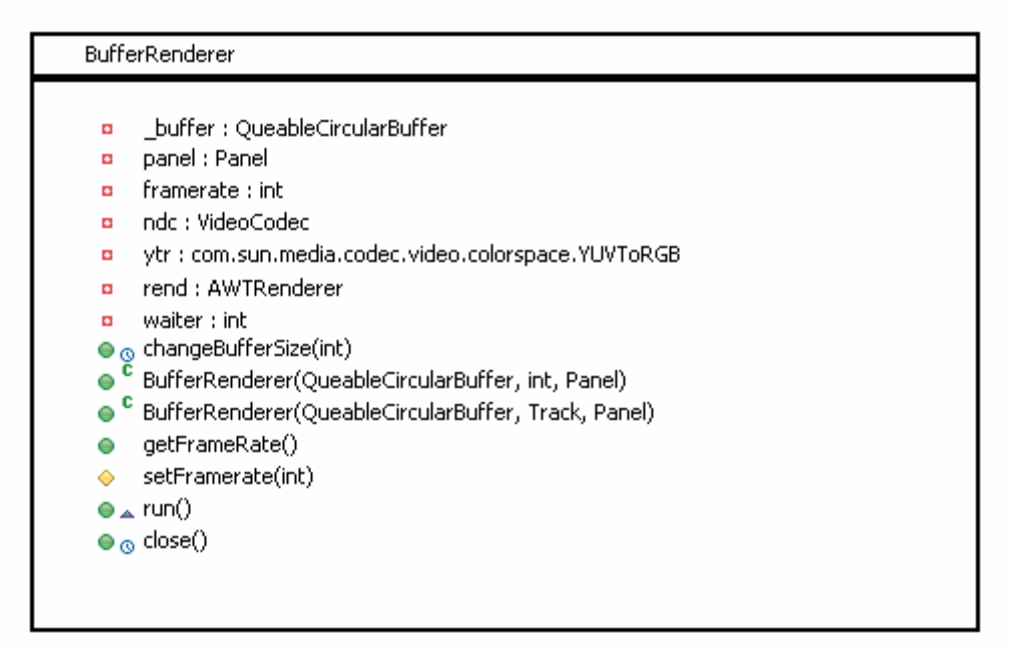


Figura 21: schema della classe BufferRenderer

Questa classe rappresenta il thread di Rendering, cioè la classe che esegue la transcodifica ed la presentazione.

Innanzitutto questa classe ha due costruttori: uno che accetta come argomenti un QueableCircularBuffer da cui estrarre i frame, un intero che indica il FrameRate del contenuto multimediale ed un Panel (*java.AWT.Panel*) su cui effettuare la presentazione ed un altro che ha come parametri un QueableCircularBuffer, una Track (classe che rappresenta la traccia, che contiene metadati come il framerate, la lunghezza ed altro) ed un Panel.

Esistono poi due metodi pubblici, *getFrameRate()* che restituisce un intero che indica il framerate e *setFrameRate(int)* che forza il framerate al valore intero passato come parametro.

All'interno del componente il FrameRate viene trasformato in un'intero (utilizzando la formula 1000/frameRate) che indica i millisecondi che devono intercorrere tra due presentazioni.

Esiste poi un metodo synchronized chiamato *changeBufferSize(int)* che incapsula il metodo *changeSize(int)* del QueableCircularBuffer e porta la lunghezza (in frame) della coda del QueableCircularBuffer alla dimensione specificata dall'intero passato come parametro.
Il metodo *close()* forza la terminazione del Rendering e la chiusura di tutti i PlugIn coinvolti nel transcoding dei frame.
L'unico altro metodo contenuto nella classe è il metodo *run()* che ridefinisce l'omonimo metodo della classe Thread e che contiene il vero e proprio corpo del Renderer.
Questo metodo innanzitutto crea un JavaDecoder, decoder di pacchetti RTP non nativo, scritto interamente in java e lo inizializza con il formato H263_RTP; nel caso questa creazione non riesca significa che l'ambiente in cui si sta lavorando dispone di una versione nativa, quindi viene creato un NativeDecoder.
Nel caso esista la catena nativa occorre un altro Codec, il JUVToRGB che viene inizializzato e messo in esecuzione; poi in entrambi i casi viene creato il Renderer vero e proprio, un'istanza di AWTRenderer che viene inizializzata per eseguire la presentazione sul Panel passato come parametro nel costruttore.
A questo punto inizia un ciclo in cui vengono letti i Buffer dal QueableCircularBuffer e vengono processati; se tutto ha avuto successo il Thread va in sleep per i millisecondi associati al framerate desiderato dall'utente a cui vanno naturalmente sottratti i tempi necessari alla decodifica ed alla presentazione; se infatti non si facesse ciò il tempo di wait globale (tempo in cui il processo è fermo

sommato al tempo necessario alla decodifica ed alla presentazione) sarebbe superiore al tempo di wait necessario per mantenere il framerate, rallentando la presentazione del contenuto multimediale.
Di una cosa di fondamentale importanza bisogna tener conto: nel caso ci siano dei sottoframe (cioè un singolo frame viene spezzettato in più Buffer che vengono inviati in maniera sequenziale) non bisogna fare una pausa dopo la presentazione ma devono essere processati uno dopo l'altro rapidamente in modo da non influire sulla qualità della presentazione all'utente.
Questo si può verificare controllando il Flag RTP_MARKER: se è uguale a 0 non ci deve essere wait, in modo da ridurre al minimo il tempo necessario alla codifica sequenziale di tutti i sottoframe e non rovinare la qualità della percezione dell'utente.
Si è però notato che questo influisce notevolmente sulla qualità dell'occupazione del QueableCircularBuffer nel PDA; infatti lo scheduler dei processi aveva dei problemi a gestire accessi rapidi al QueableCircularBuffer da parte dello stesso processo, portando ad una starvation del ReceiveStreamReader che veniva eseguito in maniera meno prioritaria.
Questo si è risolto mettendo uno wait anche nel caso di Flag RTP_MARKER uguale a 0, wait inferiore a quello specificato dal framerate ma sufficiente a permettere allo scheduler di gestire entrambi i processi senza degradare né le prestazioni del QueableCircularBuffer né la qualità della presentazione all'utente.

Nel caso si ricevesse un EOM (End Of Media) il ciclo di lettura dei frame dal QueableCircularBuffer verrebbe terminato e verrebbero fermati i plugin della catena, fermando la presentazione in quanto il contenuto multimediale è stato completamente presentato.

CAPITOLO 6

6 - Risultati sperimentali

Per verificare l’efficacia del nostro progetto abbiamo effettuato alcune sessioni di test, che riuscissero ad evidenziare le reali prestazioni del prototipo realizzato.
Abbiamo individuato 4 tipi di test importanti; il primo è servito per poter impostare al meglio parametri del buffer come dimensione e temporizzazioni, il secondo per analizzare le prestazioni globali del client mentre gli altri sono serviti per analizzare l’efficienza del nostro progetto e confrontarlo con una soluzione basata su componenti standard JMF.

6.1 – percentuale di frazionamento dei frame applicativi

Per poter prevedere un buffer delle dimensioni giuste è necessario conoscere anche la percentuale di frazionamento dei frame applicativi. Infatti nonostante sia possibile conoscendo il framerate del contenuto multimediale prevedere la durata in secondi del buffer (ad esempio per un video con un framerate di 8 Fps un buffer di 80 frame conterrebbe 10 secondi di filmato) la frammentazione dei frame applicativi in più frame RTP porterebbe questo valore ad essere più basso.
Una serie di test fatti lato server e lato client ha evidenziato una percentuale di frazionamento massima del 22% e minima del 14%

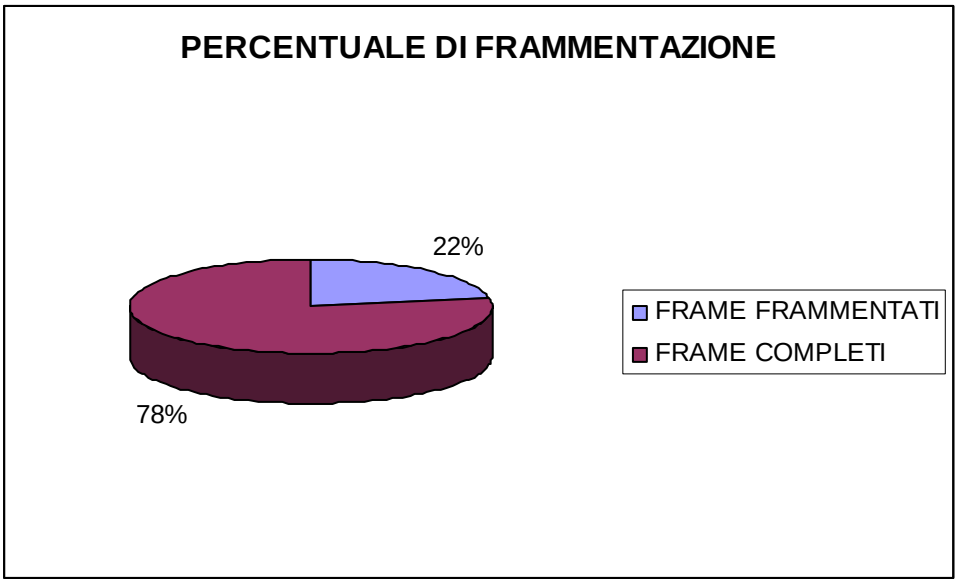


Figura 22: grafico della percentuale di frame frammentati/frame completi, in totale sono stati trasmessi 108 frame

La figura 22 mostra la percentuale di frame frammentati in un test eseguito su un PDA HP5500.

Questo primo test rivela una percentuale di frammentazione del 22%, su 108 frame trasmessi.

Lo stesso test eseguito sulla stessa macchina su 300 frame trasmessi (figura 23) ha rilevato una percentuale di frame frammentati pari al 14% che in un ulteriore test si attestava attorno al 16% (con 550 frame trasmessi, figura 24)

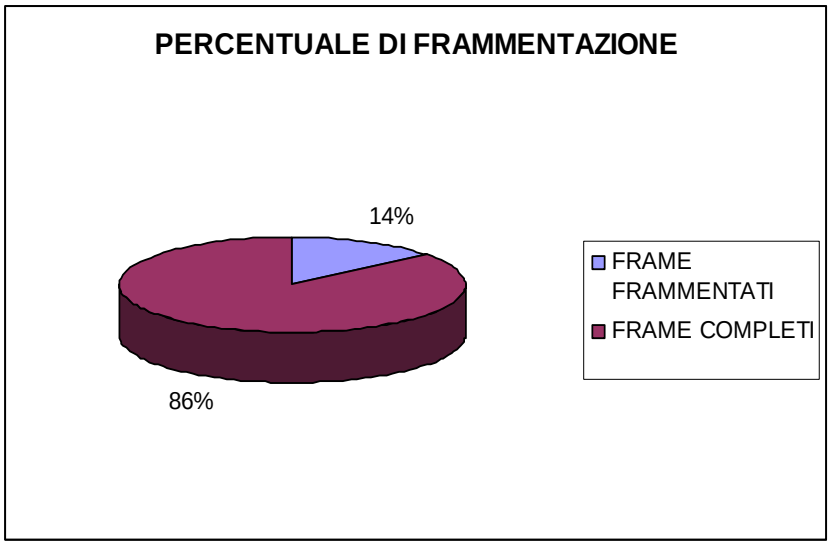


Figura 23: grafico della percentuale di frame frammentati/frame completi, in totale sono stati trasmessi 300 frame

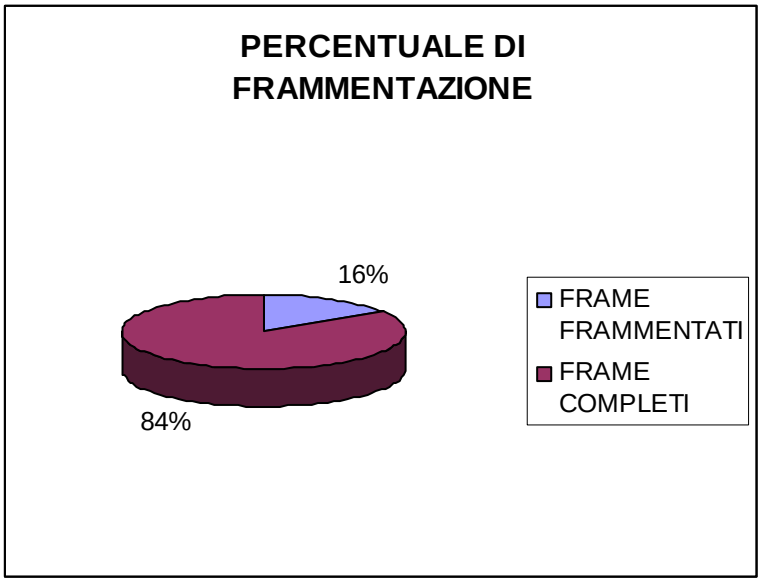


Figura 24: grafico della percentuale di frame frammentati/frame completi, in totale sono stati trasmessi 550 frame

Una volta completati questi test abbiamo testato i tempi di handoff con varie schede wireless, in modo da avere a disposizione dei dati per impostare la durata del buffer.

La figura 25 riporta in dettaglio i tempi relativi allo stesso handoff, da questi test si può notare come la qualità della scheda incida fortemente sulla durata dell'handoff e sulla conseguente perdita dei pacchetti.

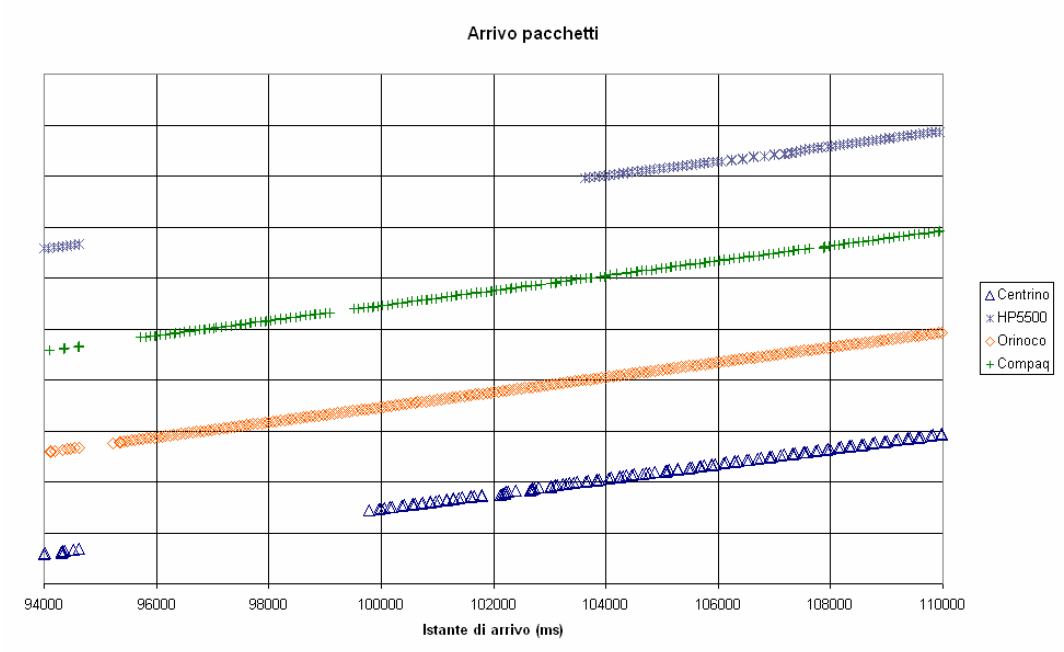


Figura 25: dettaglio dei valori dei tempi di handoff per schede WiFi divfferenti

6.2 – Tempi di decoding e rendering

Questa serie di test è volta ad analizzare la durata dei tempi di decoding e di rendering sul client, in modo da poter calibrare al meglio le temporizzazioni.

La serie di test eseguita sul PDA (figura 26) ha portato ad ottenere un tempo medio di decoding inizialmente pari a 81 ms (media pesata) per i primi 23 frame trasmessi (si noti che gli intervalli più lunghi si hanno sui primi frame, quando la catena è stata appena inizializzata, mentre in seguito i tempi di decoding si attestano sui 30 - 31 ms a frame).

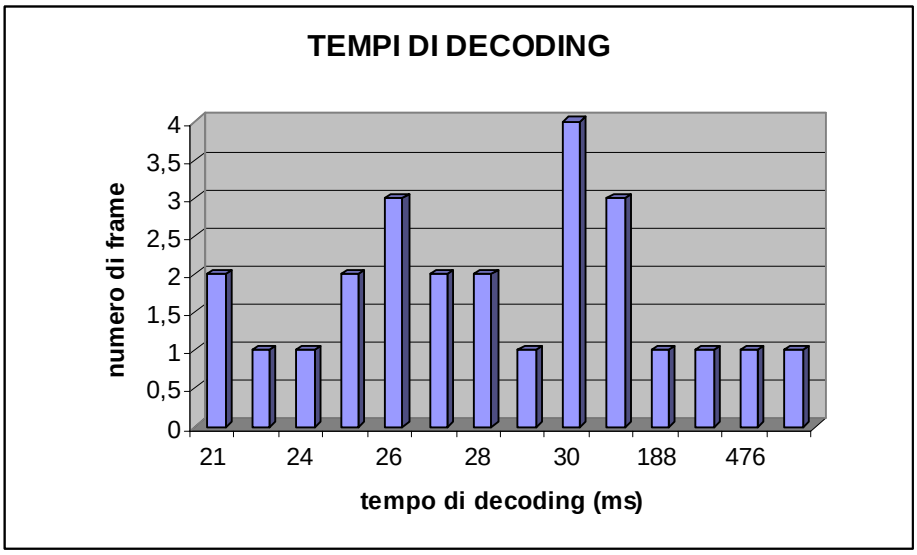


Figura 26: distribuzione dei tempi di decoding per 23 frame trasmessi, test eseguito su PDA

Lo stesso test eseguito su un pc ha portato ai seguenti risultati:

Hardware:

processore pentium III 800

256 Mb di Ram

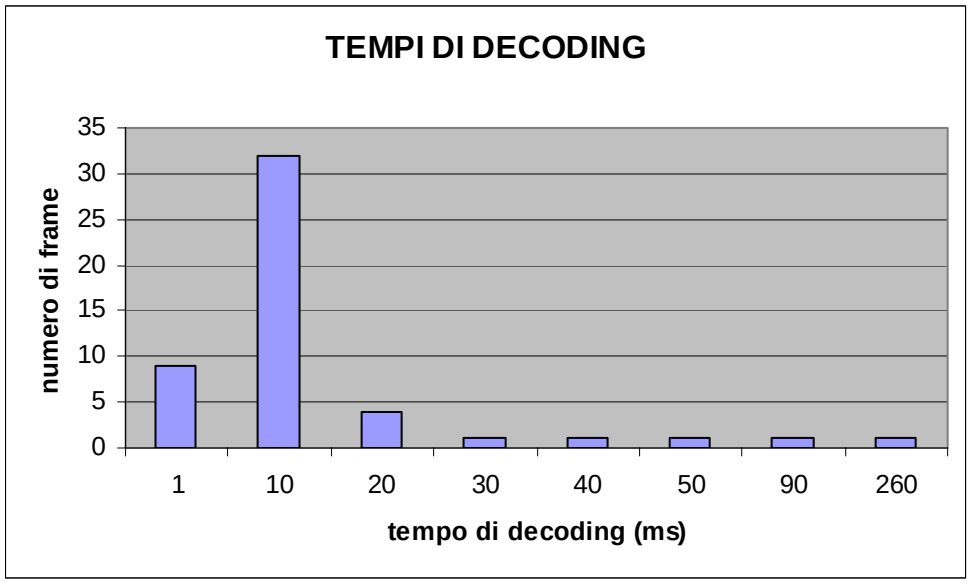


Figura 27: distribuzione dei tempi di decoding per 50 frame trasmessi, test eseguito su PC

Il valore medio del tempo di decoding si attesta sui 17 ms (si noti che anche in questo caso i tempi più lunghi si sono verificati all'inizio della decodifica del flusso mentre in seguito il tempo si è attestato attorno ai 10 ms) (figura 27).

Questi test indicano un framerate massimo raggiungibile sul PDA, infatti dato che i tempi di decodifica si attestano sui 30 ms il framerate massimo raggiungibile è di 30 Fps.

Si noti che nel caso di frammentazione devono essere decodificati più frame RTP per ottenere un frame applicativo, quindi per questi frame applicativi il tempo di decodifica globale dipende dalla frammentazione.

Sono stati anche eseguiti test sui tempi di creazione della catena utilizzando una custom chain oppure la soluzione standard del JMF.

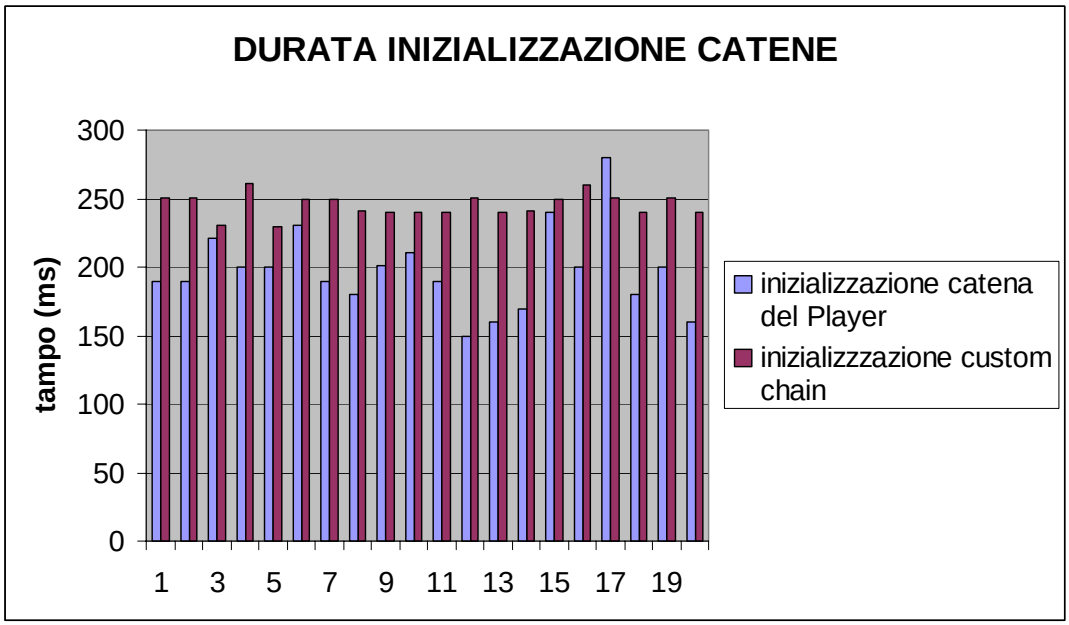


Figura 28: tempi di inizializzazione delle catene

Questi test sono stati eseguiti su un PC con configurazione:

hardware:

processore intel Pentium II x86 333 MHz

ram 163134 Kb

e hanno dato i risultati sperimentali riassunti nella figura 28.

Sono stati effettuati 20 test con entrambe le catene, il tempo medio di creazione per la catena costruita dal Player è stato di 245.45 ms, mentre per la custom chain è stato di 202.15 ms.

6.3 – occupazione e durata del buffer

Questa serie di test, effettuati su un PocketPc HP55500, è volta ad analizzare le prestazioni e l'efficienza del componente progettato.

Si è voluto analizzare l'occupazione del Buffer e la sua durata in caso di HandOff.

I test sono stati eseguiti inviando un filmato ad un framerate di 8 Fps e muovendosi lungo un percorso prestabilito a velocità costante.

Il primo test è stato eseguito con la versione del client che non effettuava pause tra il decoding di pacchetti frammentati, questo portava come è stato detto a problemi con lo scheduler, come si evince dal grafico in figura 29.

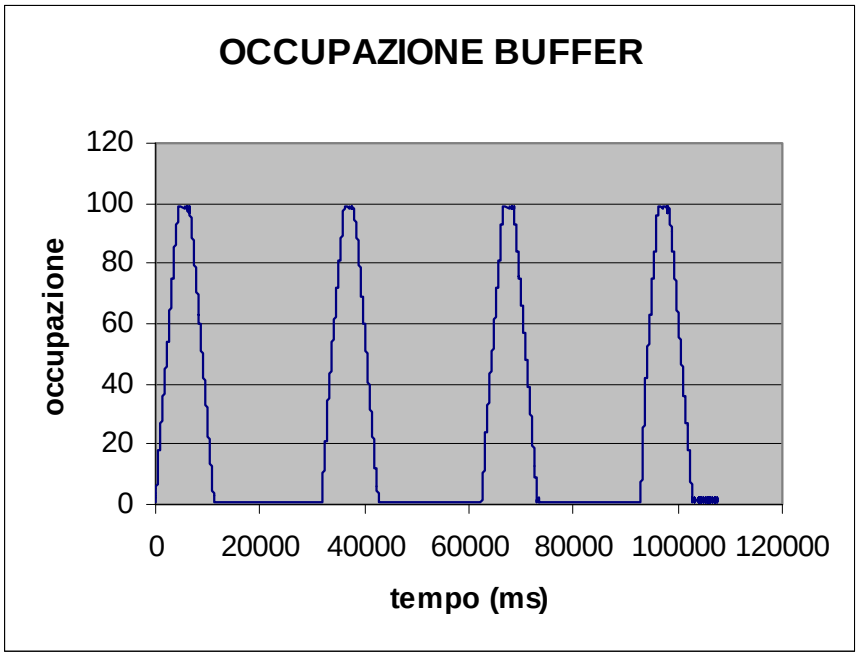


Figura 29: grafico dell'occupazione del buffer senza pause

In questo caso lo scheduler bloccava completamente uno dei due thread, o il produttore o il consumatore, fino al totale riempimento o consumo del buffer.

Inserendo le pause anche tra la decodifica di sotto-frame la prestazione si avvicina molto di più a quella desiderata (figura 30).

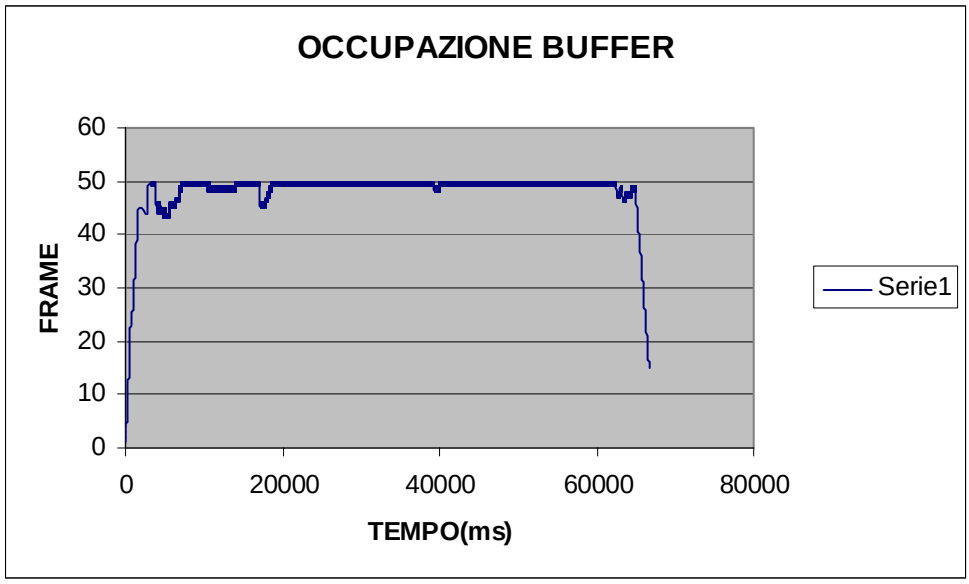


Figura 30: occupazione buffer con pausa

Questo test è stato eseguito utilizzando un buffer con coda di 50 frame.

I test seguenti mostrano oltre all'occupazione del buffer anche l'arrivo dei pacchetti RTP, mettendo in luce i momenti di perdita dei pacchetti e di Handoff.

Il test in figura 31 mostra come l'occupazione del buffer (la cui coda è inizializzata per contenere 250 frame) sia quasi sempre vicina al 100% mentre nel grafico della ricezione dei pacchetti si possono notare gli istanti di perdita della connessione.

Questi istanti in cui non vengono più ricevuti pacchetti rappresentano l'handoff e in loro presenza si può vedere che il buffer viene consumato molto velocemente.

Date le grandi dimensioni del buffer (250 frame) rispetto al framerate la durata del filmato bufferizzata è superiore al tempo di handoff, quindi la presentazione che si ottiene è continua e fluida.

Si noti che al momento del test il server utilizzato non supportava la ritrasmissione dei pacchetti persi, quindi nonostante fosse fluida la presentazione presentava dei buchi dovuti alla mancanza di frame.

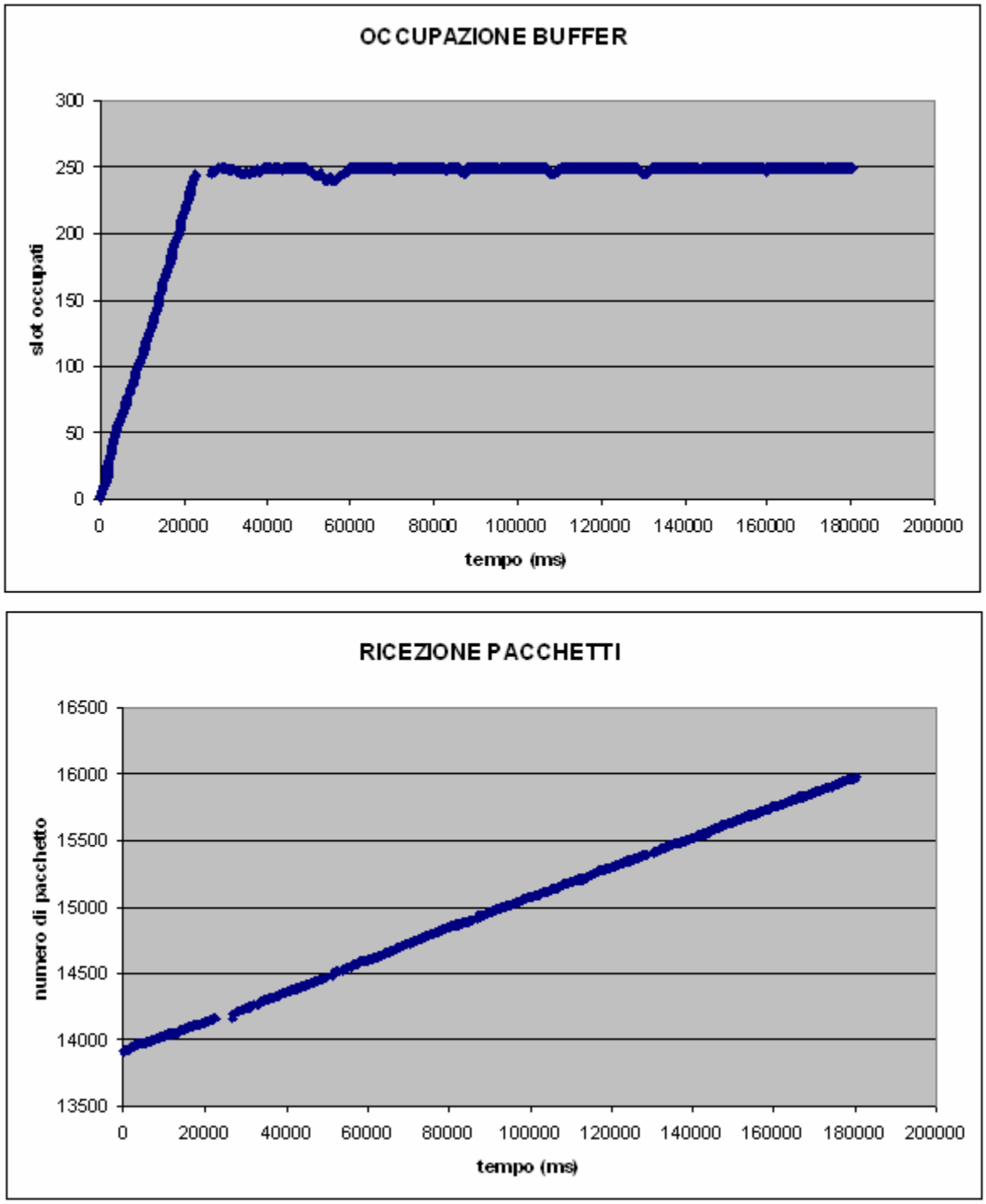


Figura 31: occupazione del buffer e ricezione pacchetti RTP

Si è cercato di risolvere il problema modificando il client e facendo in modo che in caso di perdita di pacchetti comunichi al server il numero di pacchetti persi e ne chieda la ritrasmissione.

Se il server soddisfa la richiesta il client invalida parte dei frame presenti nel QueableCircularBuffer, cioè quelli che il server ha inviato prima che gli arrivasse la richiesta del client e che quindi non sono in ordine con quelli presenti nella coda.

Il server è stato quindi modificato per poter supportare la ritrasmissione, la figura 32 mostra un dettaglio del grafico dell'occupazione del buffer sia lato client che lato server e gli istanti in cui vengono effettuate le richieste di ritrasmissione dei pacchetti.

Si può notare che in corrispondenza agli istanti in cui il client richiede la ritrasmissione di pacchetti si hanno dei picchi verticali nel buffer del server, che indicano l'avvenuta ritrasmissione, questi picchi compaiono anche lato client e indicano l'invalidazione di parte dei frame del buffer.

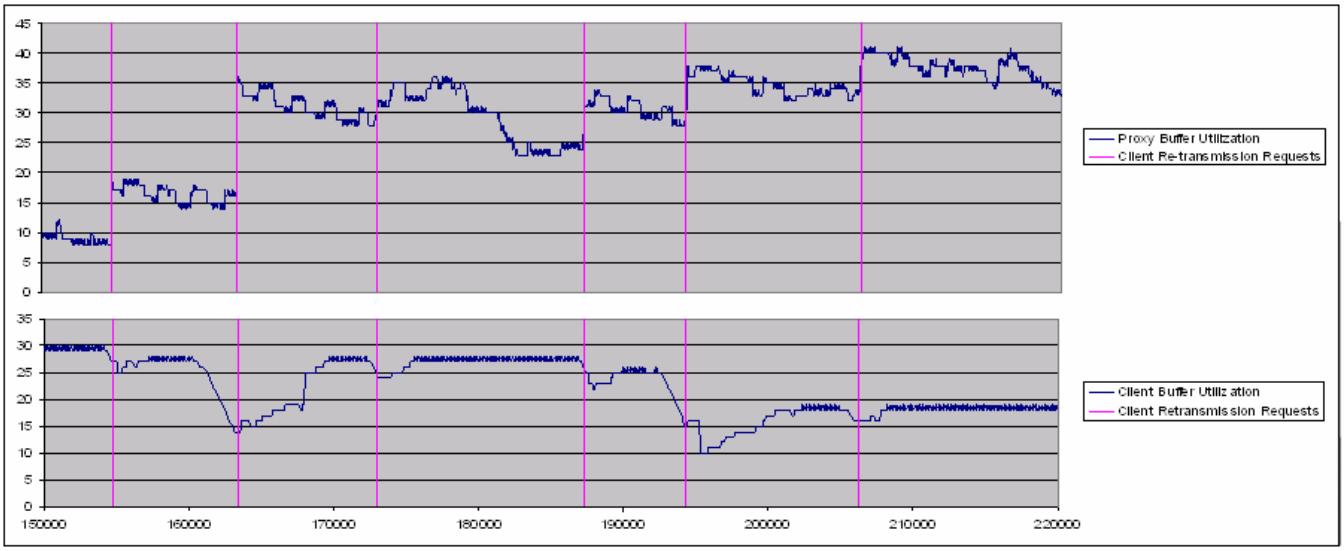


Figura 32: utilizzazione dei buffer lato client e server e istanti di richiesta di ritrasmissione

Il tempo di filmato bufferizzato nel client è in linea con quello che ci si potrebbe aspettare conoscendo il framerate del video trasmesso; in realtà esso è inferiore di circa il 16-20% a causa della frammentazione dei frame applicativi in più frame RTP.

Gli ultimi test eseguiti sul PDA sono tesi a controllare la memoria occupata dal nostro client e confrontarla con quella utilizzata da una soluzione basata sui metodi standard del JMF (che non prevede la presenza di un buffer).

La memoria totale a disposizione del PDA è 128 Mb di cui 64 allocati per la memorizzazione dati e 64 utilizzati per l'elaborazione.

Inizialmente la memoria utilizzata per l'elaborazione era pari a 10 Mb.

I test rilevano che la memoria allocata per l'esecuzione del nostro client è in media 6,133 Mb mentre per la soluzione standard viene allocata una memoria pari a 4 Mb (figura 33).

La lunghezza della coda considerata per i test è di 100 frame, aumentando la dimensione a 250 frame l'occupazione sale a 6,833Mb.

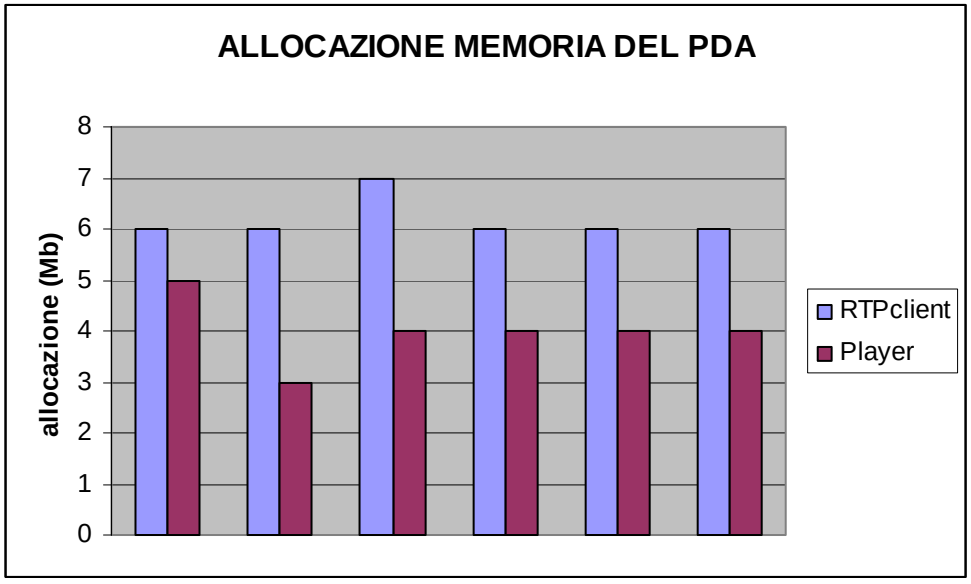


Figura 33: allocazione della memoria utilizzando l'RTPclient ed un player

Conclusioni

Il nostro lavoro ha cercato di sviluppare un client per streaming multimediale che potesse fare fronte alla perdita momentanea di connessione che si verifica in reti WiFi durante gli handoff fra celle contigue.

La tesi ha portato allo sviluppo di un client che facendo uso di un sistema di buffering e coordinandosi opportunamente con il server può far fronte alla momentanea perdita di connessione senza bloccare la presentazione del contenuto multimediale all'utente.

L'applicazione che è stata sviluppata offre buone prestazioni, migliora la fluidità della presentazione anche se deve essere accompagnata da un server che supporti opportunamente il rinvio dei pacchetti persi durante la disconnessione.

Abbiamo inoltre ottenuto dei risultati che permettono di stimare la percentuale di frammentazione dei pacchetti RTP e consentono una migliore calibrazione della durata del buffer del client.

Riteniamo che l'utilizzo del nostro client possa aumentare la QoS per servizi di stream video erogati in reti WiFi perchè permette all'utente di ottenere una presentazione fluida la cui qualità non è degradata dalla perdita momentanea di connessione.

Ci sono diverse linee possibili di sviluppi futuri, ad esempio al momento l'applicazione è stata sviluppata per poter utilizzare contenuti multimediali organizzati su una singola traccia, ma potrebbe

in futuro essere estesa per poter trattare contenuti multitraccia con i relativi problemi di sincronizzazione.

Inoltre il sistema di controllo può essere migliorato in modo da prevedere l'eventuale perdita di connessione ed allargare preventivamente il buffer, per poi restringerlo a connessione ripristinata.

Bibliografia

[1] Frank H. P. Fitzek, Martin Reisslein: “A Prefetching Protocol for Continuous Media Streaming in Wireless Environments”, IEEE Journal on Selected Areas in Communications, 2001

[2] Haifeng Xu, Joe Diamand, Ajay Luthra: “Client Architecture for MPEG-4 Streaming”, IEEE Multimedia, 2004

[3] Alejandro Terrazas, John Ostuni, Michael Barlow: “Java Media APIs: Cross-Platform Imaging, Media, and Visualization”, 2001