- INTRODUZIONE -

Gli scenari internet legati alla multimedialità sono in continua espansione e questo è chiaramente visibile osservando i servizi che la rete ci propone e notando dove molti sviluppi tecnologici tendono ad indirizzarsi attualmente. In questa tesi studieremo il mondo dei servizi di *Video on Demand* (VoD) quindi servizi progettati per la condivisione e il download di video all'interno di una rete di calcolatori. In particolare vogliamo capire come sia possibile migliorare questi servizi velocizzando l'accesso alle risorse e vogliamo capirlo attraverso l'analisi di alcune strategie orientate verso il *Proxy Caching*, elaborate affinchè il proxy possa alleggerire il server mantenendo nella sua cache parte delle risorse condivise.

Vogliamo capire quali risultati possono essere ottenuti attraverso la corretta implementazione di una suddetta politica, non solo in termini teorici, ma anche a livello pratico implementandola su uno scenario reale. Questo scenario ci viene offerto, appunto, da una piattaforma realizzata alla facoltà di Ingegneria di Bologna appositamente per i servizi di VoD; su di essa integreremo la nuova politica per capirne gli eventuali vantaggi.

La tesi è strutturata in 6 capitoli: il primo introduce brevemente alcuni scenari di VoD in Internet, il secondo capitolo analizza in dettaglio le politiche alle quali abbiamo accennato poco fa mentre il terzo capitolo presenta la piattaforma sulla quale vogliamo implementane una. Questo problema viene trattato in tre differenti capitoli: il quarto per il progetto della politica stessa, il quinto per i dettagli sulla sua implementazione e il sesto riguardo l'integrazione all'interno della piattaforma analizzata nel terzo capitolo. Segue infine una conclusione mirata a riassumere quanto spigato e mostrato nell'arco di tutta la tesi.

- CAPITOLO I -
INTRODUZIONE AGLI SCENARI DI "VIDEO ON DEMAND" IN INTERNET

1.1 Introduzione

Da quando Internet si è diffuso, nuovi sistemi e nuove soluzioni sono state progettate per offrire ai milioni di clienti connessi in tutto il mondo la migliore qualità del servizio possibile e servizi sempre nuovi ed innovativi. Sicuramente l'erogazione di servizi multimediali (audio e video) è stata protagonista e ispiratrice di una buona percentuale di questo processo negli ultimi anni, soprattutto grazie al forte avanzamento di ambienti peer-to-peer (e.g. Gnutella, Kazaa ...) che consentono lo scambio di file condivisi tra migliaia di utenti connessi in tutto il mondo, in modo assolutamente decentralizzato.

La distribuzione di queste tipologie di contenuti introduce, però, notevoli difficoltà per la mole dei dati che si scelgono di distribuire e comporta dunque maggiori tempi di attesa per gli utenti. Tali difficoltà non hanno però mai sminuito la loro grande utilità tanto che i servizi di Video On Demand (VoD) risultano tuttora un problema attuale e stimolante.

Se a questo scenario andiamo ad aggiungere l'allargamento di Internet a tutti quegli utenti non "convenzionali" come palmari, cellulari e reti wireless, il problema diventa allora ancora più complesso.

Uno scenario tipico da analizzare può essere certamente quello di un ipotetico sito che propone ai suoi clienti una rassegna dei migliori Goal delle partite di calcio giocate in giornata. Il caso migliore e più semplice da gestire è certamente quello di un solo utente che dal suo P.C. di casa si connette e, beneficiando di una rapida connessione (o di tanta pazienza), fa un download del video e al termine lo guarda. La situazione però potrebbe peggiorare... Siamo di domenica sera e a termine partita molti utenti decidono di riguardarsi il goal della vittoria; si connettono al sito e richiedono un download del video scelto. La politica deve ovviamene fare in modo che i ritardi siano i minori possibili e garantire al più presto il

servizio ai numerosi clienti. La cosa peggiora se alcuni utenti richiedono il video dal loro cellulare con una connessione GPRS e quindi con protocolli e latenze diversi da tutti gli altri. Vedremo tra breve alcune politiche implementate per far fronte a queste problematiche.

1.2 Proxy come supporto per la comunicazione tra Client e Server

Come abbiamo detto questi sistemi di informazione richiedono accessi in tempo reale e condivisione di documenti salvati in server distribuiti. E' chiaro che gestire la comunicazione tra *client* e *server* con un supporto intermedio (*Middleware*) può fornire dei vantaggi notevoli. Decidiamo quindi di utilizzare un supporto intermedio che si pone a metà strada tra la comunicazione e gestisce alcune problematiche.

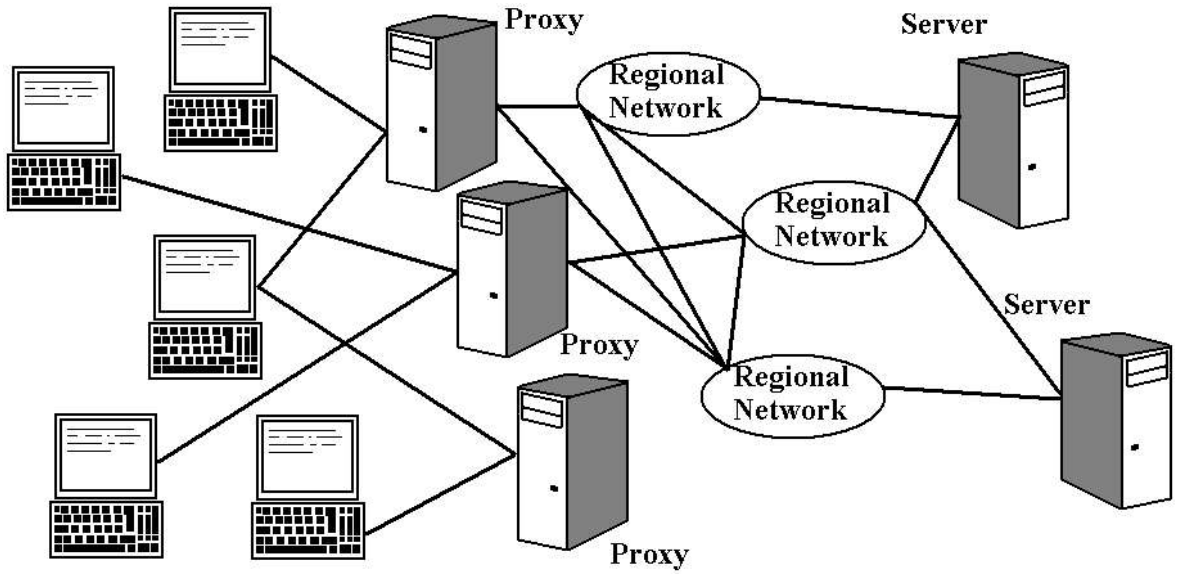

Fig1.1 – Esempio di architettura che utilizza proxy per la comunicazione client-server

Un proxy si presenta come un piccolo server che si pone tra un gruppo di client e server; necessita di uno spazio di caching per il salvataggio dei file proporzionale al numero di client e sostanzialmente minore rispetto a quello del server. In ogni modo il salvataggio di questi dipende anche dalla popolarità (quindi dal

numero di richieste) riguardanti lo stream stesso e dalla larghezza di banda disponibile nel tragitto proxy-client. Un proxy può aumentare in modo significativo la qualità del servizio per clienti che possono beneficiare di larga banda malgrado la presenza di un collo di bottiglia nel tragitto tra client e server; è quindi chiaro che il principale utilizzo dei proxy multimediali avviene in contesti di controllo di congestione. Ma i vantaggi nell'utilizzo di un proxy non si vedono solo nella gestione di difficili situazioni di traffico per client con larga banda, ma anche in casi di client con bande più ridotte; infatti esso gestisce il supporto di accessi asincroni, abbassa le latenze di startup e diminuisce il carico lato server. Sempre riguardo i client con piccola banda (non necessariamente P.C. ma anche palmari o cellulari) il proxy multimediale mantiene spesso video di dimensione (e qualità) diverse consentendo una maggiore fluidità del traffico e permettendo un download più veloce di un video meno pesante.

Gli stream real-time possiedono diverse proprietà che possono essere sfruttate proprio attraverso un proxy: infatti a causa della sua durata e grandezza un oggetto multimediale non viene mai spedito intero attraverso la rete (vedremo i vantaggi tra breve), inoltre gli stream multimediali sono in grado di modificare la loro grandezza aggiustando la qualità (e quindi la loro pesantezza) come abbiamo sottolineato poco fa.

Come è stato mostrato nel disegno precedente tutto il traffico di stream viene fatto passare attraverso proxy. Il sistema è sempre *end to end*, come nel caso di di un normale protocollo client-server e non c'è quindi bisogno di alcun supporto aggiuntivo dalla rete. Ovviamente il proxy è sempre associato ad un gruppo di client che si trovano nelle vicinanze. Lo stesso proxy intercetta tutti i flussi e gli stream riguardanti i client ai quali è associato e opera un adeguato controllo della congestione e della qualità.

Come introdotto precedentemente un file multimediale è solitamente troppo grande per essere spedito interamente lungo la rete; una delle proposte di miglioramento nell'ambito del caching multimediale è stata quindi quella di fare caching di prefisso (*Prefix Caching*). Immaginiamo di avere un file video che decidiamo di dividere in due parti in modo arbitrario; chiamiamo la prima parte *prefisso* e la seconda *suffisso*. Pensiamo ora ad uno scenario

composto da un gruppo di client che ricevono video da stream di un singolo proxy. Quando avviene una richiesta al server, dopo che viene istanziato il proxy, avviene lo stream del prefisso sul proxy stesso e poi al client. Se il prefisso non esaurisce il video allora viene fatto lo stream del rimanente suffisso. E' chiaro che se un secondo client richiede il medesimo video si ricerca prima se il suo prefisso si trova sulla cache di qualche proxy nelle vicinanze e si scarica quindi da esso evitando di dover richiedere uno stesso stream al server.

Nei prossimi capitoli ci occuperemo delle problematiche relative alla gestione della cache sul proxy, ed in particolare al *Prefix Caching* studiandone i benefici e come sia possible, a volte, migliorare l'uso delle risorse nel complesso scenario del VoD.

- CAPITOLO II -
SCHEMI E PROTOCOLLI DI CACHING SU PROXY

2.1 Introduzione

Come abbiamo sottolineato nel capitolo precedente la presenza del proxy è fondamentale nello scenario che stiamo studiando; cerchiamo quindi di analizzare quali sono i metodi più usati per sfruttare i proxy nei servizi VoD. Concentriamoci ora su un solo proxy che interfaccia diversi client a un server e analizziamo i costi di un ipotetico download con gli schemi proposti da [1].

PARAMETRO	SIGNIFICATO
N	Numero di Video
L_i	Lunghezza del video *i* espressa in secondi
b_i	Larghezza di banda del video *i* espressa in bit al secondo
u	Caching Grain
n_i	Grandezza del video *i* espressa in Caching Grain
f_i	Probabilità di accesso al video *i*
R_i	Frequenza di richiesta del video *i*
R	Frequenza di accesso al repository del Server
S	Grandezza della cache del proxy espressa in Caching Grain
v_i	Lunghezza espressa in secondi del prefisso del video *i* salvato su cache
v	Vettore dei salvataggi : $V = \{v_1, v_2, \ldots, v_n\}$
c_s	Costo di trasmissione per bit sul cammino server - proxy
c_p	Costo di trasmissione per bit sul cammino proxy-client
$C_i(v_i)$	Costo di trasmissione per unità di tempo per il video *i* quando un suo prefisso di lunghezza v_i è in cache

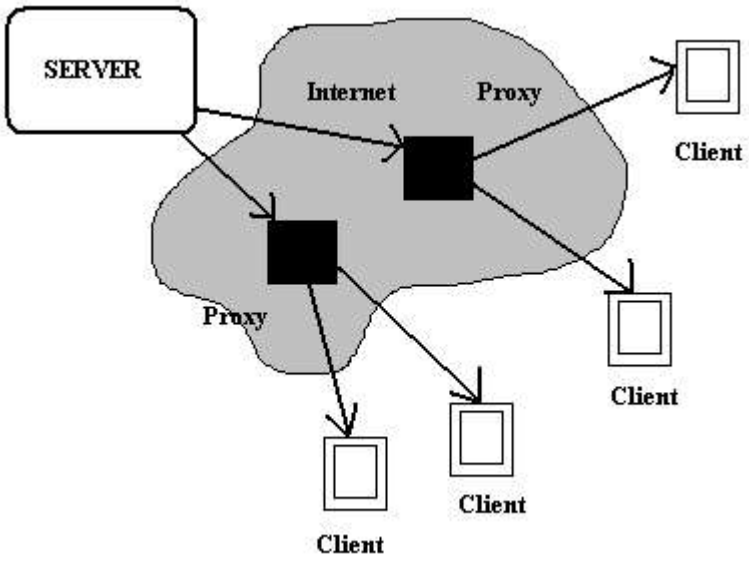

Fig2.1 – Esempio di architettura che utilizza proxy per la comunicazione client-server

Consideriamo un server con un repository di N video ognuno a bit-rate costante e di conoscere a priori (magari grazie a un monitoraggio del sistema) le probabilità di accedere ad uno dei video. Ogni accesso al repository dei video ha una probabilità f_i di richiedere il video i , ed in particolare: $f_1 + f_2 + ... + f_i = 1$

Definiamo poi R_i la frequenza di accesso al video i e R la frequenza di accesso al repository dei video, allora risulterà che

$$R_i = R\ f_i \qquad (\text{Per } 0 < i < N)$$

Definiamo ora il concetto di *caching grain* che rappresenta la più piccola unità di cache e simboleggia l'unità di misura per le nostre allocazioni; infatti la grandezza del video i e della cache sono espressi in *caching grain*. Ogni video è poi caratterizzato da una banda (*bps*) e una lunghezza in secondi.

$$n_i\ u = b_i\ L_i$$

Ipotizziamo ora che il proxy possa salvare S unità

$$b_1\ v_1 + b_2\ v_2 + ... + b_i\ v_i < u\ S$$

Definiamo a questo punto una serie di possibili prefissi per il

video i

$$A_i = \{m_i \mid 0 < m_i < n_i\}$$

dove m_i rappresenta la grandezza del prefisso in *caching grain* e

$$m_i\ (u / b_i)$$

rappresenta invece la sua lunghezza in secondi. Il nostro scopo è quello di cercare di massimizzare il salvataggio dei prefissi minimizzando i costi di trasmissione.

Quello che faremo ora sarà di andare ad analizzare alcuni schemi di trasmissione che sfruttano il caching del prefisso del proxy come parte integrante per offrire la migliore qualità del servizio sia in ambienti unicast che multicast nel path proxy – client.

2.2 Politiche Unicast e Multicast per il Prefix Caching

1. Sbatch (Unicast Suffix Batching)

L'SBatch è un semplice schema basato su percorsi tra proxy e client unicast. Immaginiamo che ad un tempo 0 arrivi una richiesta per un video i, quindi il proxy apre subito uno stream verso il client per trasmettergli il prefisso poi l'SBatch schedula la trasmissione del suffisso dal server al client.

Nel caso di qualsiasi altra richiesta in arrivo nel intervallo $(0, v_i]$ il proxy anticipa il suffisso $(L_i – v_i)$ in arrivo al nuovo client affinchè non ci sia una nuova richiesta al server; in questo modo multiple richieste di suffisso vengono soddisfatte assieme. Il prefisso invece viene scaricato appena possibile dal proxy che lo mantiene memorizzato in cache. I costi di questo schema si semplificano con la seguente equazione:

$$C_i(v_i) = c_s \frac{L_i - v_i}{1 + R_i v_i} + R_i b_i$$

Se $v_i = 0$ ($v_i = L_i$) non è possibile servire multiple richieste.

2. UPatch (Unicast Patching With Prefix Caching)

Immaginiamo che, come prima, arrivi una richiesta per il video *i* da un client ad un tempo t1 = 0. Ipotizziamo ora che arrivi una seconda richiesta nell'intervallo di tempo [v*i* < t2 < L*i*]; a questo punto il proxy può comandare una nuova richiesta del suffisso per intero direttamente dal server oppure può schedulare un patch della parte di suffisso che gli manca [t2 – v*i*] dal server mentre sull'altro canale riceve il suffisso assieme al primo client dallo stream corrente

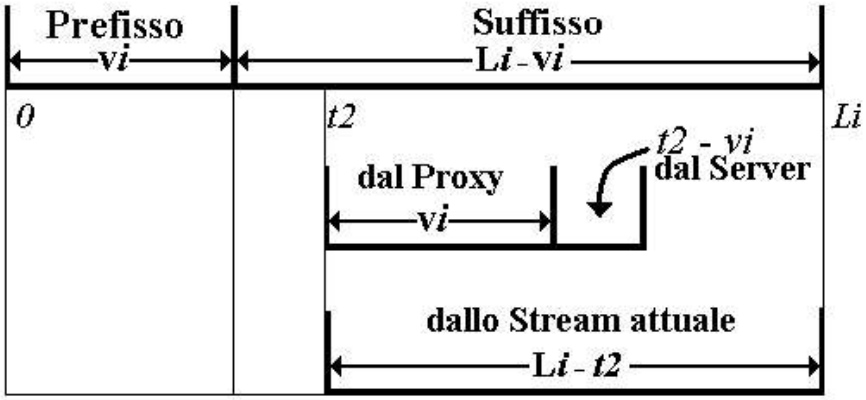

Fig2.2 – Schema della politica UPatch

La decisione di trasmettere un patch o il suffisso per intero dipende da un *Suffix Threshold* (G*i*), una soglia misurata dal principio del suffisso; se una richiesta arriva all'interno di G*i* allora si opta per un patch, altrimenti viene richiesta una nuova trasmissione del suffisso per intero dal server. I costi si semplificano con l'equazione sotto esposta:

$$C_i(v_i) = c_s R_i b_i \frac{\frac{R_i G_i^2}{2} + L_i - v_i}{1 + R_i(v_i + G_i)} + c_p R_i b_i L_i$$

La scelta di G*i* viene fatta per minimizzare i costi di trasmissione per un video *i*.

Se v*i* = 0 non è possibile servire multiple richieste.

3. MPatch (Multicast Patching With Prefix Caching)

Fino ad ora abbiamo considerato schemi adatti ad un path proxy – client unicast, adesso vogliamo concentrarci sulle problematiche multicast. Immaginiamo che, come in precedenza, giunga una richiesta ad un tempo 0. Il server inizia la trasmissione del suffisso al proxy ad un tempo v*i* e il proxy trasmette i dati ricevuti ai client multicast. Le richieste successive possono o leggere dallo stream corrente utilizzando un canale separato per il recupero dei dati mancanti, oppure far ri-schedulare una nuova trasmissione dal server. A questo proposito scegliamo una soglia T*i* ; se una richiesta arriva ad un tempo t tale che 0 < t < T*i*, allora si procede in uno dei seguenti casi a seconda del rapporto tra v*i* e T*i* :

Caso 1 (T*i* < *vi* < L*i*) Il client riceve il segmento [0, t2] da un canale separato unicast dal proxy e il segmento (t2, L*i*] dallo stream multicast corrente.

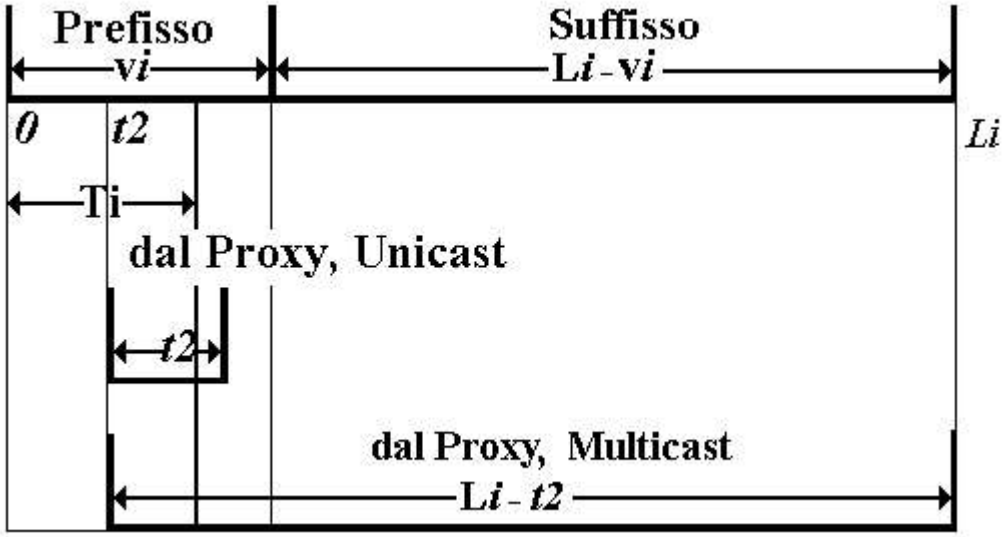

Fig2.3 – Schema della politica MPatch

Quindi i costi si semplificano così :

$$g_i(v_i) = \frac{R_i b_i}{1 + R_i T_i}\left[(L_i - v_i)\, c_s + L_i c_p + \frac{R_i v_i^2}{2} c_p\right]$$

Caso 2 ($0 < T_i < L_i$) Se *t2* si trova tra 0 e v_i allora il meccanismo è il medesimo del primo caso ; se invece *t2* si trova tra v_i e la soglia T_i allora il client riceve il segmento $[0, v_i]$ da un canale separato unicast dal proxy e riceve il segmento $(t2, L_i]$ dallo stream multicast attualmente in corso, poi il segmento $(v_i, t2]$ giunge dal server al client via unicast. Riassumiamo i costi nella tabella che segue :

$$g_i(v_i) = \frac{R_i b_i}{1 + R_i T_i}\left[(L_i - v_i)\, c_s + L_i c_p + \frac{R_i v_i^2}{2} c_p + \frac{R_i (T_i - v_i)^2}{2}(c_{is} + c_p)\right]$$

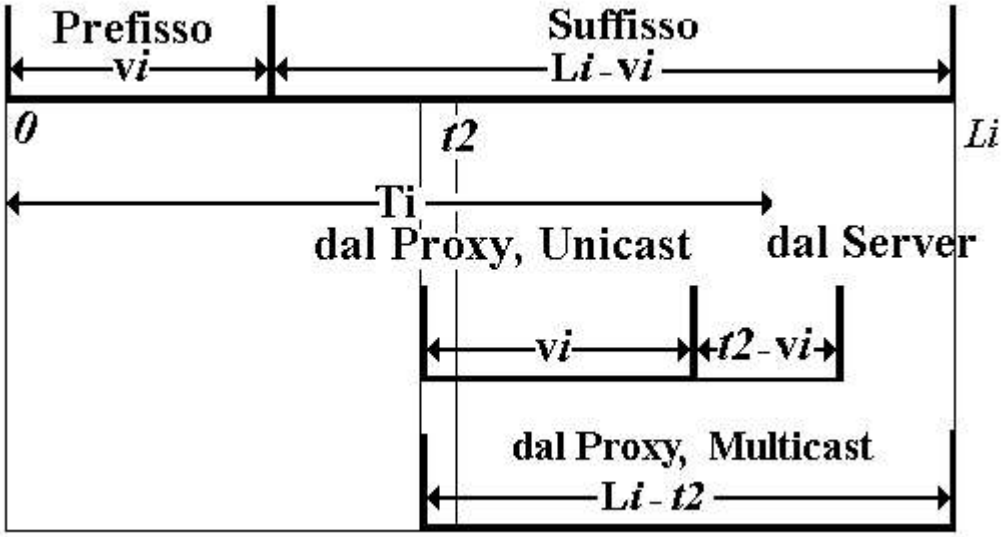

Fig2.4 – Schema della politica MPatch

Se definiamo $h_k(v_i)$ come il minimo costo di trasmissione nei casi $K = 1, 2$ allora :

$$h_k(v_i) = \min(T_i)\{\, g_k(v_i, T_i),\ 0 < T_i < L_i \,\}\ ,\ k = 1, 2$$

Quindi per un certo prefisso v_i, avremo un costo di :

$$C_i(v_i) = \min\{\, h1(v_i), h2(v_i) \,\}$$

4. MMerge (Multicast Merging With Prefix Caching)

A questo punto possiamo implementare una politica adeguata di

merging magari integrando il merging dello stream con il caching sul proxy. Ci basiamo su una politica di tipo *Closer Target* la quale sceglie lo stream che è giunto prima di tutti gli altri e che è ancora nel sistema, come prossimo bersaglio. Nel nostro caso questa politica viene usata per la gestione del caching del prefisso sul proxy (che poi trasmetterà al client) mentre il suffisso non presente sulla cache del proxy viene trasmesso dal server il più tardi possibile. Definiamo p_j la probabilità di una richiesta di un prefisso da j secondi per il video i (con $0 < j < L$) ; allora il costo di trasmissione per il video i diventa :

$$C_i(v_i) = \sum_{j=1}^{v_i} j \cdot p_j \cdot b_i \cdot c_p + \sum_{j=v_i+1}^{L_i} (j \cdot (c_p + c_s) - v_i \cdot c_s) \cdot p_j \cdot b_i$$

Notiamo infine che in MPatch e MMerge lo stream del video i viene da una singola sorgente (Server o Proxy), quindi entrambi riducono il carico del controllo Multicast.

2.3 Risultati sperimentali e Test

A questo punto, dopo aver visto e descritto queste quattro diverse tipologie, analizziamo i risultati sperimentali; consideriamo un repository di 100 video lunghi 2 ore, con la stessa larghezza di banda e definiti con una probabilità di accesso O = 0.27.

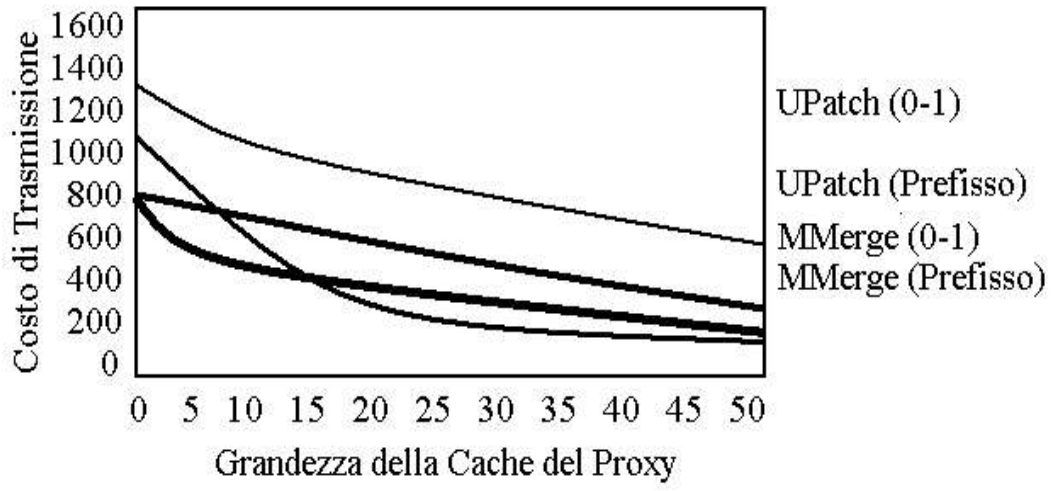

Fig2.5 – Studio del comportamento delle poliche nel caso di caching 0-1 o di prefisso

Rappresentiamo la cache del proxy come come una precentuale del repository (*r*) dei video e stabiliamo che una unità di misura *caching grain* contiene un minuto di dati.

Come primo esperimento, seguendo le specifiche appena introdotte, confrontiamo le differenze tra un caching 0-1 (il video è memorizzato totalmente e senza prefisso) e un caching con prefisso; come vediamo dal grafico **Fig2.5**, se c*p* = 0 e R = 100 richieste/min, allora l'utilizzo di caching con prefisso migliora notevolmente il servizio.

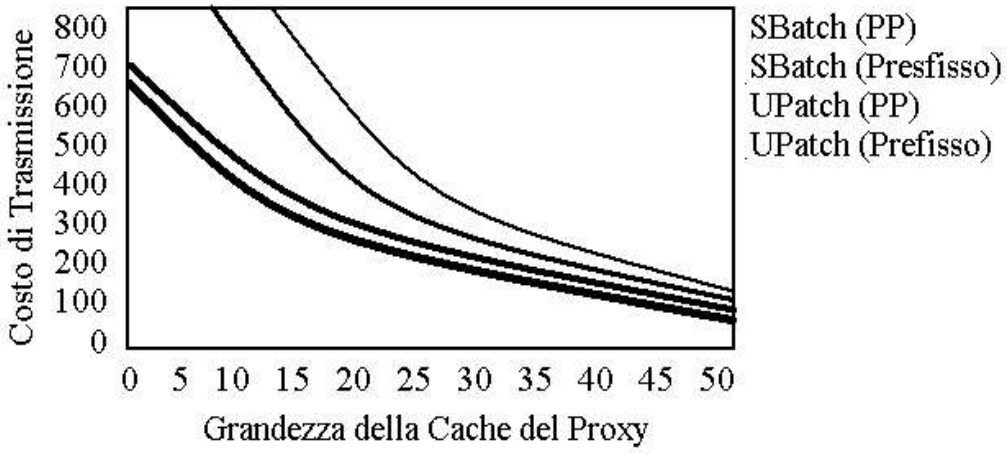

Fig2.6 – Studio del comportamento delle politiche unicast nel caso di caching 0-1 o Proportional Priority

Nel grafico **Fig2.6** abbiamo invece messo a confronto la politica di cache del prefisso appena vista con una *Proportional Priority* (PP) dove la grandezza della cache del proxy è proporzionale alla grandezza del video e alla sua probabilità di accesso (rimanendo sempre nei limiti massimi di spazio consentiti dalla cache) e usando R = 30 rich./min. e c*p* = 0 . Come si può chiaramente notare l'UPatch riduce i costi in modo sostanziale rispetto l'SBatch, soprattutto in caso di proxy con piccole cache.

Questo grafico è sviluppato supponendo che sia stata scelta un'ottima soglia per l'UPatch; in caso contrario i risultati subirebbero dei peggioramenti. Se non si riesce a garantire quest'ultima condizione allora conviene orientarsi sull'SBatch il quale risulta anche notevole in caso di cache del proxy molto grande, garantendo le stesse performance del UPatch (con ottima soglia) ma con maggiore semplicità.

Il grafico **Fig2.7** puntualizza meglio l'allocazione della cache del Proxy nel caso di UPatch con cache del prefisso. Scegliamo, anche

in qusto caso, c*p* = 0 dato che i costi di trasmissione sul cammino proxy-client non dipendono dalla allocazione della cache; infatti si utilizza lo stesso schema per minimizzare i costi sul cammino server-proxy (che è risulta essere indipendente dal valore di c*p*). Il grafico sopra illustrato risulta molto simile per l'Sbatch: quando la cache del proxy è piccola vengono memorizzati solo i video più richiesti mentre se la cache è più grande si memorizza un numero maggiore di video. A seconda del valore di R la grandezza per la memorizzazione di un dato video aumenta in funzione della sua probabilità di acceso; per valori di R molto alti avviene una distribuzione più uniforme su tutti i video. Questo fatto rappresenta una sostanziale differenza rispetto al *Proportional Priority* Caching.

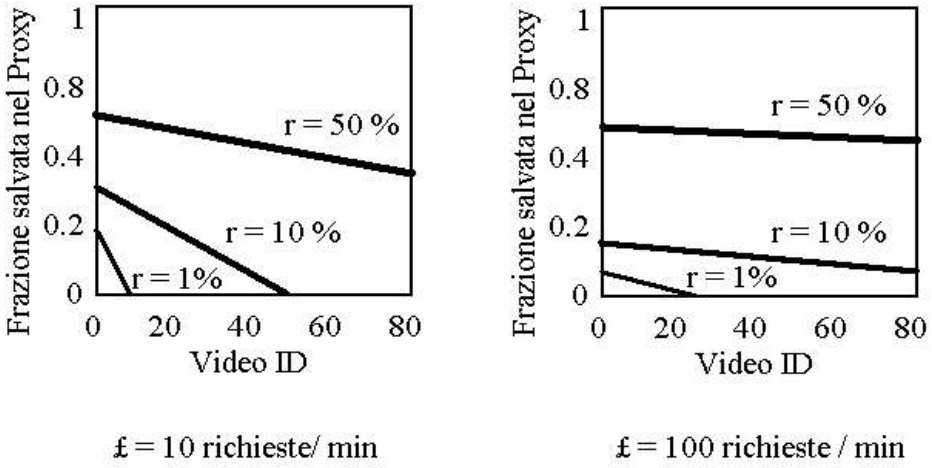

Fig2.7 – Allocazione della cache del proxy nel caso di UPatch con cache di prefisso

Passiamo al caso Multicast con c*p* = 0.5 e R = 30 richieste/min :

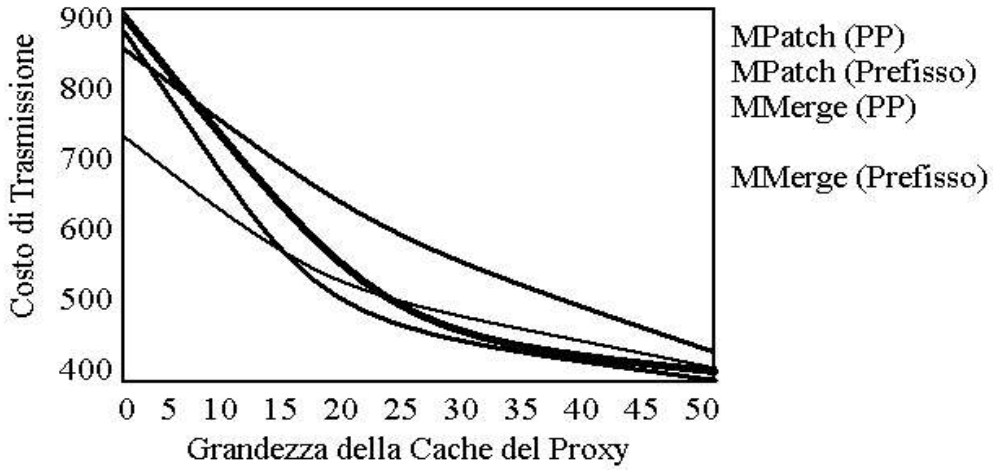

Fig2.8 – Studio del comportamento delle politiche multicast con caching di prefisso o Proportional Priority

Notiamo chiaramente che le uniche differenze appaiono nel caso di MMerge su *Proportional Priority* Caching e MMerge su caching standard del prefisso quando la cache del proxy è di ridotte dimensioni; infatti nel caso di r = 1% abbiamo una differenza del 20% sui costi di trasmissione rispetto al *Proportional Priority* Caching. E' interessante notare come l'MMerge fornisca in alcuni tratti prestazioni inferiori al MPatch, in particolare per Proxy Cache di grandezza dal 25% al 50 %.

Analizziamo ora l'allocazione della cache del proxy per Mpatch e MMerge nel caso di caching standard del prefisso. Nel caso c*p* = 0 allora i costi sul cammino proxy-client sono nulli e quindi non ci sono differenze nell'allocazione ; nel caso c*p* non sia nullo ci sono delle differenze. In particolare nel caso dell'MPatch la soglia tende a crescere col diminuire delle probabilità di accesso, quindi alcuni video meno richiesti possono esigere prefissi di dimensioni maggiori rispetto a video più popolari. Nel caso di MMerge è lo spazio di cache associato ad un video a diminuire se diminuisce la popolarità del video stesso ma, nel caso R sia sufficientemente grande la grandezza della cache può invece aumentare nel caso di una diminuzione della popolarità del medesimo video. Questo avviene perchè la lunghezza del prefisso (e quindi la frequenza di accesso) diminuisce al diminuire della popolarità.

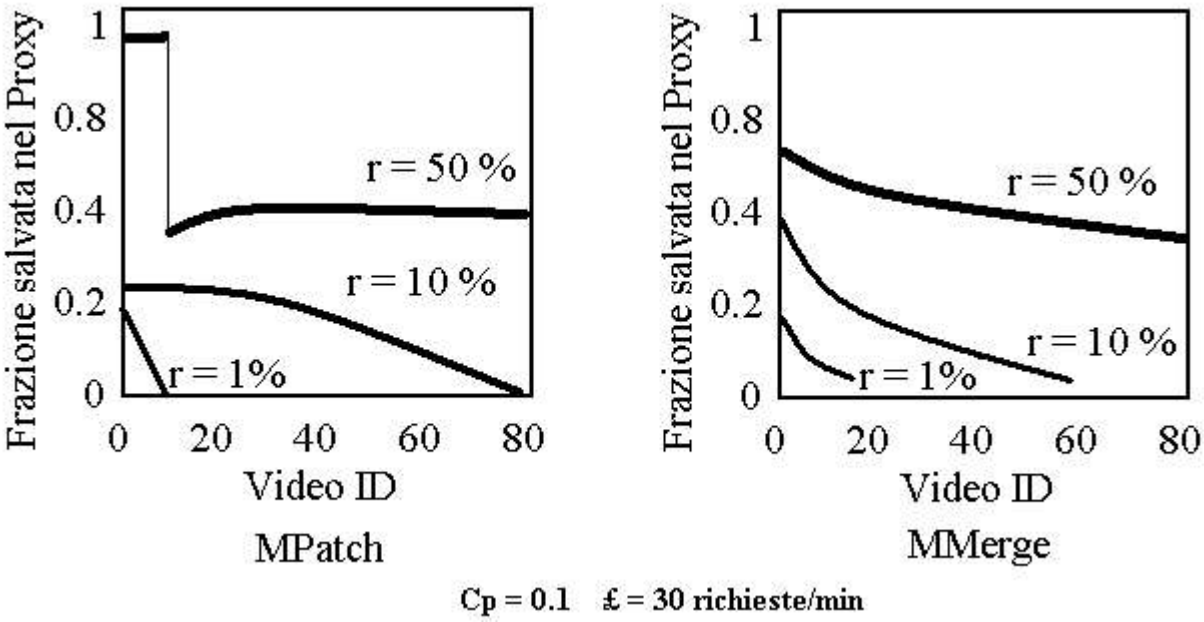

Fig2.9 – Allocazione della cache su proxy nel caso di Mpatch e Mmerge con caching di prefisso

A questo punto abbiamo tutti i mezzi per operare un confronto più ad alto livello, cioè tra gli algoritmi basati su scenari unicast e quelli invece concepiti per scenari multicast. Possiamo subito constatare che nel caso $c_p = 0$ la scelta di un multicast nel tragitto proxy-client si dimostra una scelta ottimale, occorre però analizzare anche il caso in cui $c_p > 0$; il grafico seguente ci mostra i risultati dato un valore di $c_p = 0.1$ e R = 10 richieste/min. Come è chiaro MPatch e MMerge risultano assolutamente migliori di UPatch.

Il secondo grafico invece mostra lo studio dei costi a seconda del variare della frequenza di accesso al repository del server in termini di richieste al minuto impostando un valore di r = 10%. L'andamento delle curve multicast riconfermano le potenzialità di MPatch e MMerge, in particolare abbiamo una riduzione dei costi del 61% nel caso di 100 richieste al minuto rispetto all'UPatch. Sono quindi chiari i benefici dell'utilizzo del multicast nel tragitto proxy-client.

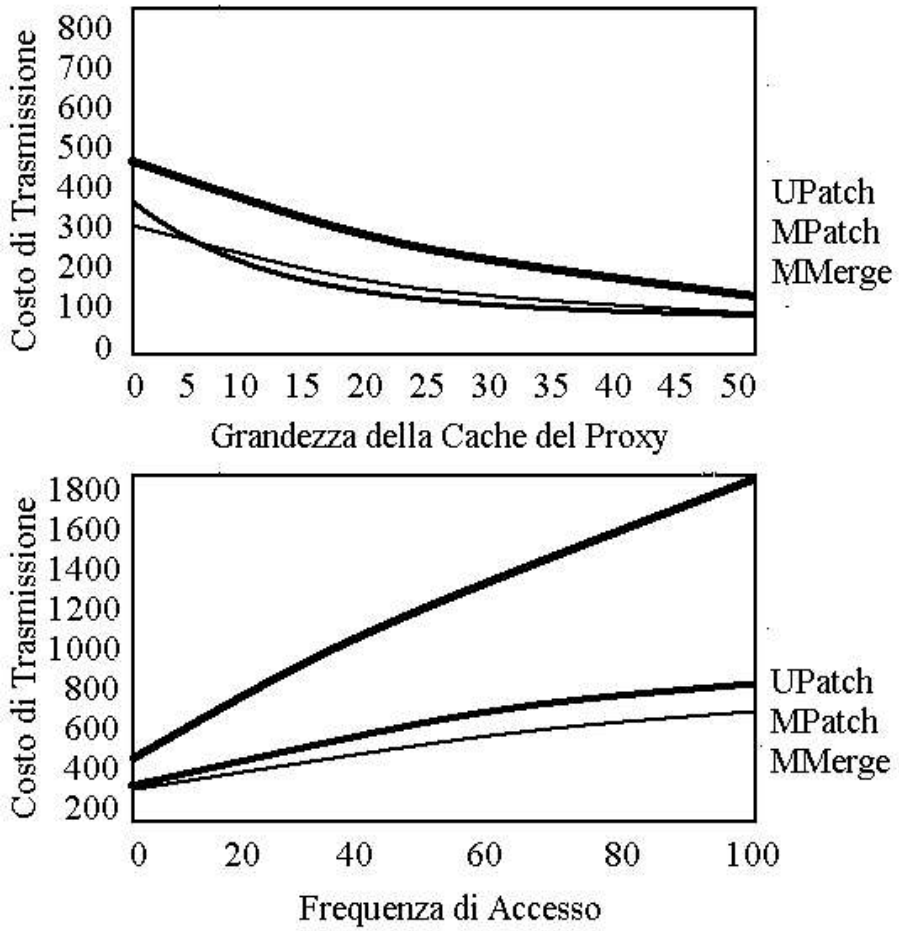

Fig2.10 – Analisi dei rapporti tra costo di trasmissione, frequenza di accesso e grandezza della cache del proxy nel caso di politiche Unicast e Multicast

2.4 Riassumendo

I risultati degli esperimenti illustrati ci hanno dimostrato che:

1. Con la stessa grandezza del proxy l'utilizzo del prefix-caching comporta un sostanziale risparmio nei costi di trasmissione rispetto al caching del video nella sua completezza. Anche con piccole cache (10%, 20%) si ottengono sostanziali benefici.
2. Nel caso di cache del prefisso bisogna stare attenti al tipo di trasmissione, alla frequenza di accesso e al costo di trasmissione sul sentiero proxy-client perchè influenzano sulla allocazione. In particolare non risulta vantaggioso un *Proportional Priority* Caching nel caso di alte frequenze di accesso al repository del server.
3. Nel caso di servizi unicast si possono ottenere sostanziali miglioramenti semplicemente ragionando sugli schemi di trasmissione e sul caching del prefisso anche se un servizio multicast si dimostra sempre ottimale.

- CAPITOLO III -
MUMOC

3.1 Introduzione

Come abbiamo visto nei precedenti capitoli le casistiche coprono e risolvono molti problemi migliorando i servizi ma, come in tutti gli ambiti di ricerca, non esistono soluzioni pronte per ogni problema; è quindi necessario adattare e rinforzare le conoscenze ottenute per il miglioramento di un determinato scenario.

Introduciamo quindi *Mobile agent based Ubiquitous multimedia Middleware Open Caching* (MUMOC), una piattaforma studiata per la gestione di servizi VoD basata sul caching di prefissi e metadati. Nasce come progetto dell'Università di Ingegneria di Bologna basato sull'implementazione del precedente SOMA e MUM.

Secure and Open Mobile Agent (SOMA) è un sistema ad Agenti Mobili sviluppato con Java ai fini di sicurezza e interoperabilità. Innanzitutto SOMA è basato su un modello sicuro e propone una serie di meccanismi per costruire e rinforzare una flessibile politica dei sicurezza, ma è anche in grado di operare con differenti ambienti come CORBA e MASIF. Infine SOMA dispone delle astrazioni necessarie alla scalabilità in uno scenario globale ed è configurabile e gestibile dinamicamente anche attraverso interfacce Web.

Mobile agent based Ubiquitous multimedia Middleware (MUM) è un middleware sviluppato per rispondere alle richieste di dinamicità, apertura e configurabilità necessarie in uno scenario mobile che coinvolge sia terminali standard che wireless. Inoltre MUM tratta a livello middleware sia la gestione delle risorse che i servizi oltre a studiare l'utilizzo degli Agenti Mobili in caso di gestione della Qualità del Servizio (QoS), continuità della sessione e configurazione di servizi multimediali.

Immaginiamo che alla facoltà di Ingegneria di Bologna venga implementato un servizio di video-registrazione delle lezione con la possibilità di download per gli studenti. Bob decide di scaricare la lezione della mattina perchè non ha ben capito alcuni argomenti, quindi dal suo portatile attaccato alla biblioteca centrale fa una richiesta al server.

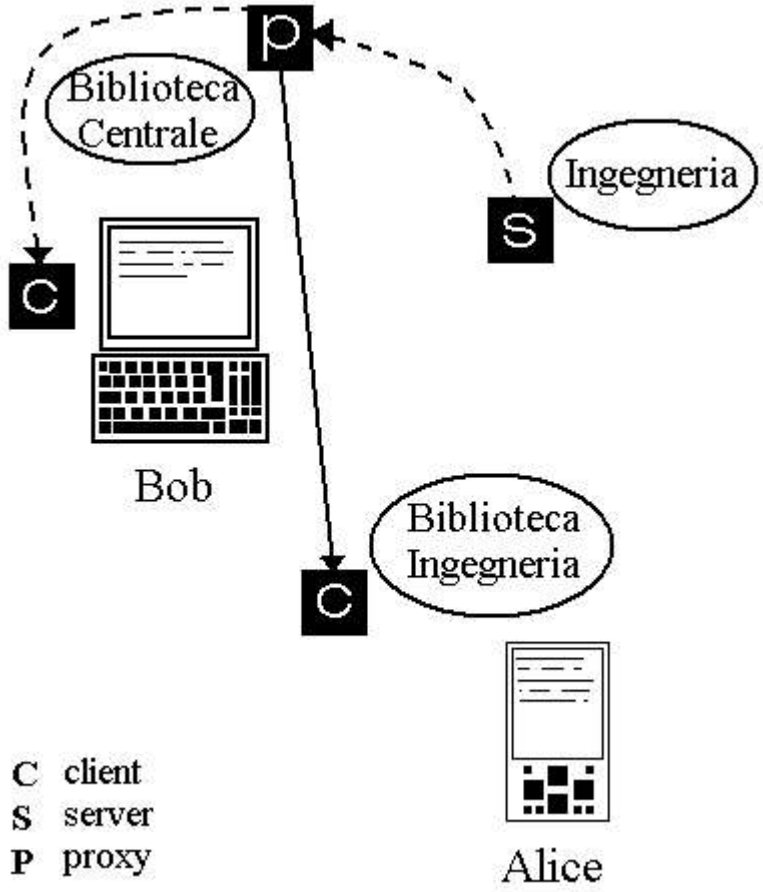

Fig3.1 – Scenario di esempio per il funzionamento di MUMOC

Il server riceve la richiesta e istanzia un proxy sul quale viene memorizzato il prefisso assieme ad un documento di metadati che descrive il prefisso stesso. Dopo un tempo *t* Alice decide di scaricare la medesima lezione perchè è arrivata tardi alla mattina e non ha potuto seguirla; quindi dalla biblioteca di ingegneria richiede il download del video dal suo palmare. Il sistema intercetta la richiesta e si accorge, interrogando i metadati salvati, che è presente il prefisso del video richiesto sul nodo proxy dal quale Bob ha scaricato poco prima. MUMOC allora fa il download per Alice possibilmente salvando anche nel nodo della biblioteca di Ingegneria (dalla quale Alice è connessa). Poi avviene il download del suffisso direttamente dal server della facoltà.

Il nucleo della distribuzione di MUMOC è basata su un gruppo di host fissati interconnessi da una LAN broadband. L'organizzazione di questi host è ad albero di cui i client giocano solo il ruolo di foglia.

MUMOC riduce il carico sul percorso server-proxy non solo attraverso il caching del prefisso ma anche grazie ad una originale soluzione di batching basata su proxy : *Suffix Batching*. Questa tecnica è pensata per ambienti unicast (differentemente dalle classiche tecniche di batching che vengono implementate solo per schemi multicast) introdotti nei capitoli precedenti. Riprendendo il caso precedente immaginiamo di considerare una richiesta proveniente dal nodo della libreria centrale; il prefisso viene immediatamente mandato al client e viene schedulata una richiesta di stream del suffisso dal server. In caso di richieste successive da parte di client connessi localmente viene prontamente fatto uno stream del prefisso in cache e si anticipa il suffisso da un canale separato. Questo approccio mira a ridurre i costi sul path server-proxy assumendo che i costi di trasmissione proxy-client siano significativamente più bassi. E' poi necessario notare come in MUMOC i metadati rappresentino un punto fondamentale: ad ogni richiesta il sistema controlla i relativi metadati nel repository distribuito per comandare appropriatamente le trasmissioni.

3.2 Analisi del Modello

A questo punto possiamo scendere più nel dettaglio analizzando l'architettura di MUMOC basata appunto sull'architettura Client-Proxy-Server analizzata nei due precedenti capitoli. Il middleware è organizzato in due sezioni fondamentali : *Local Prefix Cache* e *Metadata Browser*.

Il *Metadata Browser* svolge una ricerca e ritorna i metadati relativi al video richiesto secondo determinati criteri. Questo modulo non solo fornisce funzioni per le query ma anche per l'inserimento e la cancellazione di metadati nei repository. Ovviamente i metadati sono basati su tecnologia XML. *Local Prefix Cache* invece mantiene i prefissi nei nodi interposti tra client-server; è sostanzialmente basato su due componenti, una indirizzata al caching e l'altra invece come rinforzo all'SBatch, gestendo sapientemente i problemi di client con limitate capacità di memorizzazione attraverso un componente detto *SBatch Shadow Proxy* (SBSP) dedito a questo ruolo.

Vediamo ora più in dettaglio l'architettura del sistema :

1. Modulo "Content Cache"

E' un modulo che si occupa del download del prefisso, estende le strategie di replacement ed estende SBatch, come precedentemente introdotto. Il nucleo del sistema consiste in un *Caching Enable Proxy* che viene istanziato dinamicamente per ogni richiesta VoD. Questo proxy prevede a sua volta 3 componenti :

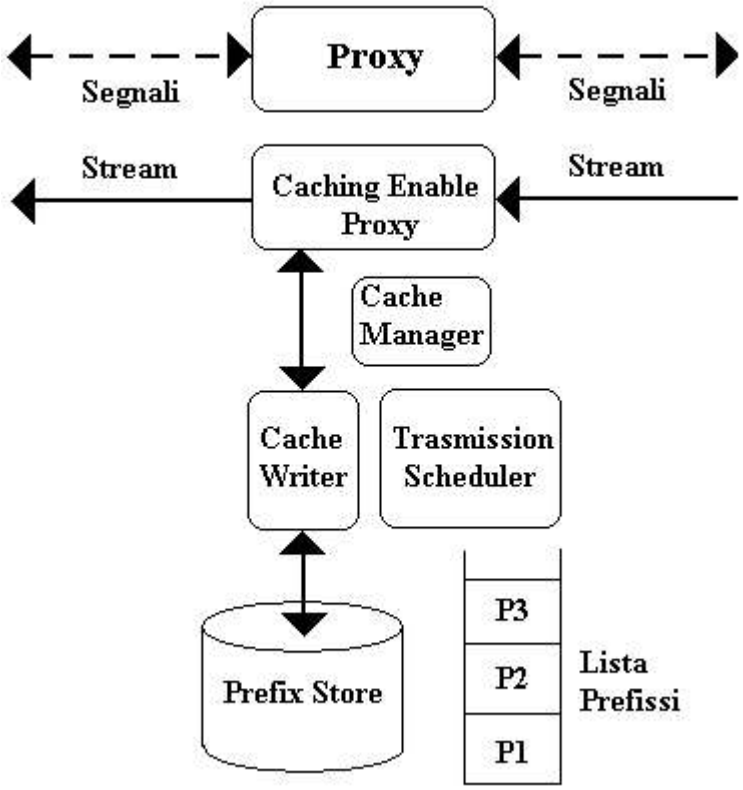

Fig3.2 – Rappresentazione del modulo "Content Cache" di MUMOC

a. Cache Manager – Gestisce tutti gli eventi importanti (come ad esmpio l'arrivo del suffisso al proxy) oltre agli accessi in scrittura alla cache.

b. Cache Writer – Semplicemente salva gli stream VoD che giungono dal Server.

c.Trasmission Scheduler – Gestisce le richieste lato server e lato client.

Come dimostrato nei capitoli precedenti le potenzialità

nell'utilizzo di un prefisso abbinato ad una politica adeguata è di notevole beneficio; tuttavia, siccome la lunghezza del prefisso non influenza in modo significativo i nostri costi di trasmissione, è stato deciso di adoperare un minimo slot di cache sul proxy di 3 secondi.

Come politica di replacement MUMOC usa la *Proportional Priority*, che assegna ad ogni prefisso un numero di slot nella cache dipendente dalla popolarità del VoD e dalla sua grandezza.

Come già detto MUMOC si basa su politica SBatch che estende in modo originale: nel caso di una richiesta di VoD non possa essere soddisfatta servendosi di un SBatch allora MUMOC attiva un SBSP agent che si occupa di risolvere il problema per l'invio del prefisso. Poi anche il suffisso viene fatto passare dallo stesso agente prima di giungere al client. Questo sistema non richiede buffering da parte del client e comporta l'implementazione di un solo canale di trasmissione tra il client e l'agente SBSP.

2. Modulo "Metadata Browser"

Il *Metadata Browser* gestisce l'albero del sistema attraverso la comunicazione tra i rami e le foglie e anche tra altri *Metadata Browser*.

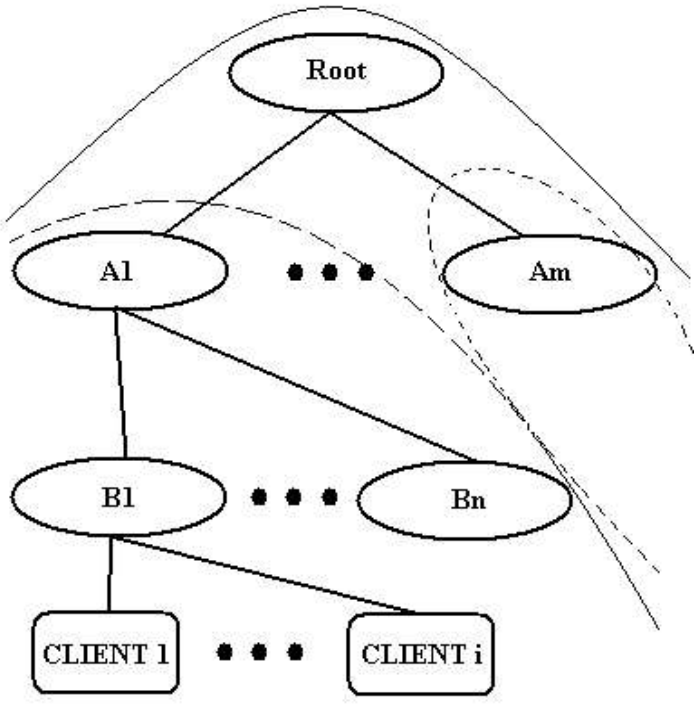

Fig3.3 – Rappresentazione della distribuzione del Metadata Browser di MUMOC

Ogni nodo mantiene i matadati per tutti i VoD salvati nei suoi sotto-alberi (ad esempio, A1 contiene tutti i metadati B1 ... Bn). Per quanto riguarda i client essi si comportano come *Metadata Browser Client* e rappresentano client leggeri che possono navigare tra i metadati dell'albero senza però poter avere un proprio repository locale (in modo da non sovraccaricare il loro limitato spazio).

Ogni volta che avviene una richiesta viene fatta un query nel nodo locale che, se non comporta risposta, viene propagata nell'albero fino al raggiungimento di un risultato.

Se un prefisso viene eliminato allora vengono eliminati anche i suoi metadati prima localmente e poi, attraverso la propagazione di una query, globalmente.

Per una buona interoperabilità MUMOC sfrutta la tecnologia MPEG7 con *Dublin Core* per la descrizione dei frammenti di VoD presenti nella rete. Infatti *Dublin Core* garantisce un'alta interoperabilità con grandi librerie Web di risorse e nel caso non sia in grado di descrivere un complesso contenuto multimediale allora interviene lo standard MPEG7.

3.3 Analisi di MUMOC

3.3.1 Standard implementativi di MUMOC

In questo paragrafo studieremo come sono progettate le diverse parti di MUMOC, e come sono state realizzate a livello di implementazione.

Innanzi tutto bisogna sottolineare che la comunicazione tra i vari nodi avviene attraverso l'utilizzo di una serie di messaggi prestabiliti a priori dal sistema stesso che vengono intercettati dai vari nodi attraverso delle *Protocol Unit*; queste interpretano i messaggi e svolgono l'azione richiesta con metodi appropriati.

La trasmissione dei messaggi tra un nodo e un'altro avviene attraverso dei canali di comunicazione Socket mentre la trasmissione dei video attraverso canali RTP per mezzo di componenti fissi chiamati *VideoAgent* che hanno appunto il compito di realizzare lo streaming.

Vediamo ora i componenti della piattaforma MUMOC:

3.3.2 Client MUMOC

Il componente principale del client è ovviamente un *Video Agent*. Esso ha lo scopo di gestire la connessione e il download da server del video; in particolare sfrutta una *Protocol Unit* per l'esecuzione delle operazioni.

La stessa *Protocol Unit* accetta le richieste comandate dall'utente mentre un manager detto *Session Manager*, deve inizializzare i due componenti appena illustrati e comandare opportunamente l'operazione adeguata. Ogni richiesta trasmessa attraverso questo componente viene fatta sotto-forma di metadato; ogni metadato comprende tutte le principali informazioni del video secondo gli standard *Dublin-Core* ed *MPEG-7* (contenenti: nome del video, autore, copyright, informazioni di compressione ...), e viene passato da un nodo ad un altro sino al momento della trasmissione.

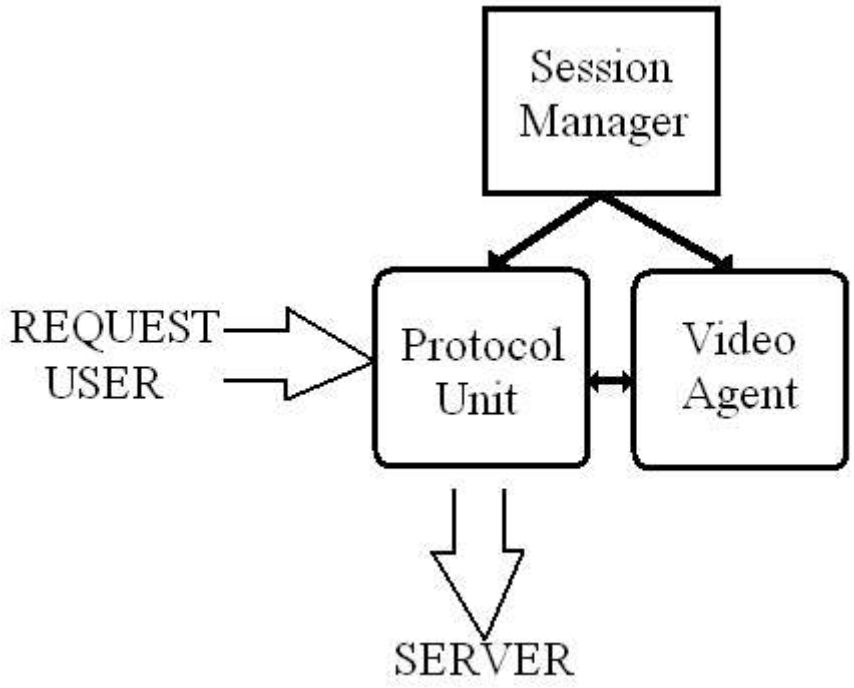

Fig3.4 – Schema a blocchi dei componenti che formano il Client MUMOC

All'atto della richiesta, il client richiede immediatamente la trasmissione al server e attende l'inizio dello streaming.

3.3.3 Proxy MUMOC

Come abbiamo appena visto, il client semplicemente riceve una

richiesta per il download di un video e conseguentemente invia il metadato corrispondente al server. Il proxy, trovandosi nel tragitto server - client, intercetta la richiesta attraverso una apposita *InProtocol Unit* e la considera. *ProxyManager* è semplicemente è il manager del proxy; esso inizializza una *Protocol Unit* per la comunicazione con il nodo successivo e comunica attraverso la *OutProtocol Unit*. Infine ottiene lo stream di dati da passare al client attraverso uno *Strem Receiver* .

Quando una richiesta viene intercettata dal proxy, la *InProtocol Unit* comanda immediatamente una ricerca del video segnalato nel metadato prima in locale e nel caso negativo su tutti i nodi attraverso il Metadata Browser, come vedremo tra breve nel paragrafo successivo.

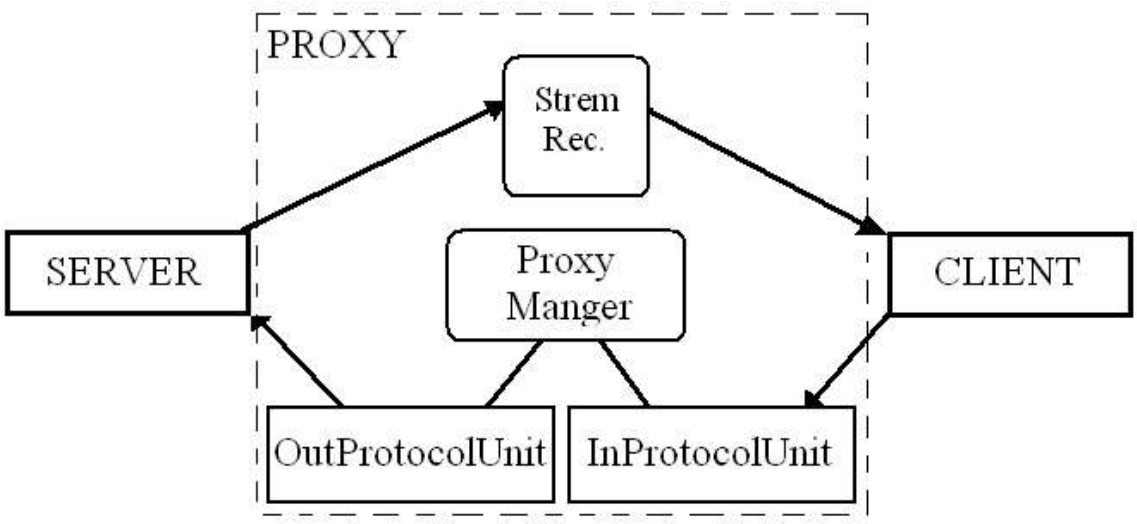

Fig3.5 – Schema a blocchi dei componenti che formano il Proxy MUMOC

3.3.4 Metadata Browser

ServiceManager è il componente principale del Metadata Browser; esso inizializza i componenti per le politiche, per il caching e per le connessioni. E'presente poi un *Cache Manager* che rappresenta una sorta di *Protocol Unit* per i metadati, infatti ha il compito di soddisfare opportunamente le richieste. Quest'ultime vengono intercettate da un'apposito componente detto *Connection Maker*. Ovviamente tra le operazioni possibili ci sono anche quelle di ricerca e di gestione della cache di metadati. Infine il

ServiceManager inizializza anche un *PoliciesManager* per la gestione delle politiche.

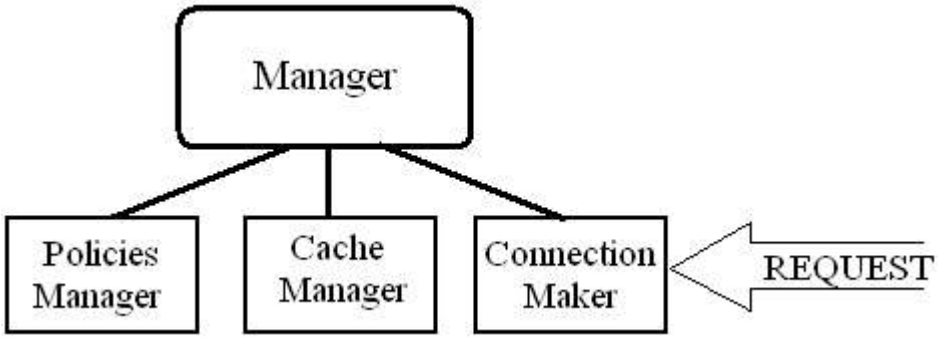

Fig3.6 – Schema a blocchi dei componenti che formano il Metadata Browser di MUMOC

Attraverso il package Metadata Services viene fatta una ricerca sui nodi dell'albero MUMOC ottenendo i metadati delle presentazioni trovate che coincidono con la richiesta fatta.

3.3.5 Server MUMOC

La struttura implementativa del server non è diversa da quella degli altri nodi: *Server Manager* è il gestore che inizializza un agente e una *Protocol Unit*. Quest'ultima è semplicemente la *Protocol Unit* del server che esegue, come accade per client e proxy, le richieste pervenutele via Socket e comanda le trasmissioni delle presentazioni attraverso il *Video Agent*.

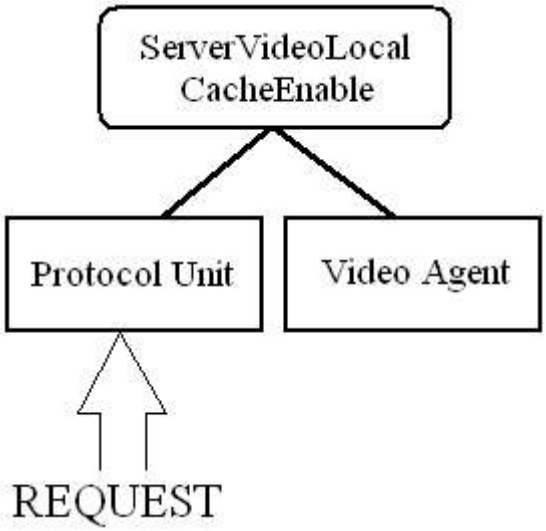

Fig3.7 – Schema a blocchi dei componenti che formano il Server MUMOC

Abbiamo quindi studiato la piattaforma sulla quale dobbiamo costruire il nostro progetto: la realizzazione SBatch per MUMOC. Tratteremo questo discorso in due capitoli : il primo comprendente i dettagli progettuali e il secondo contenente le tecniche realizzative ed implementative; infine integreremo il tutto nella piattaforma appena vista.

- CAPITOLO IV -
ARCHITETTURA DI UN SBATCH PER MUMOC

4.1 Progetto dello Scheduler

Studiando l'SBatch durante i primi capitoli abbiamo capito quale sia il suo funzionamento e le sue potenzialità; cerchiamo ora di progettarlo affinchè possa essere integrato nell'architettura di MUMOC.

Prima di tutto notiamo che sarà necessario un componente apposito che tenga traccia delle richieste dei diversi client che usufruiscono del servizio e che le serva correttamente. In uno scenario reale i client mandano le loro richieste su proxy (1) il quale comanda il download del prefisso (se presente in cache) quindi il proxy stesso che ha accettato la richiesta del client schedulerà la richiesta del suffisso (2) immediatamente richiedendo che lo streaming venga fatto dopo un'intervallo di tempo che si ipotizza sufficiente per terminare lo streaming di prefisso. Il componente che si occupa della gestione di queste richieste è lo *Scheduler*.

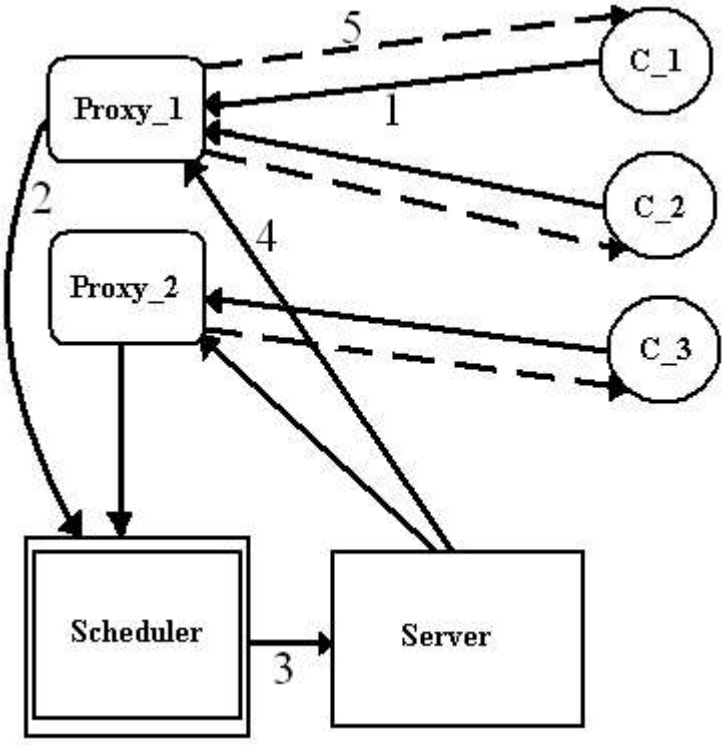

Fig4.1 – Modello di localizzazione dello Scheduler in un tipico scenario multi-client

Vediamo ora in dettaglio quale sarà il suo funzionamento: continuamente si mette in attesa di una richiesta e quando questa arriva ne tiene traccia e attende per un tempo definito dalla richiesta stessa per poi sollecitare (3) lo streaming di suffisso da server a proxy (4) che giungerà infine client (5) dopo il prefisso fomando quindi una presentazione completa.

Nel caso giunga una richiesta mentre si sta considerando ancora la prima si considera se il tempo di attesa della nuova richiesta è minore di quella già considerata oppure no. Nel primo caso si interrompe la prima richiesta e si serve la nuova attendendo opportunamente poi ri-servendo la prima per il tempo occorrente. Nel secondo caso si continua a considerare la prima richiesta per poi assecondare la seconda ad un tempo adeguato. Nel disegno seguente sono spiegate entrambe le modalità.

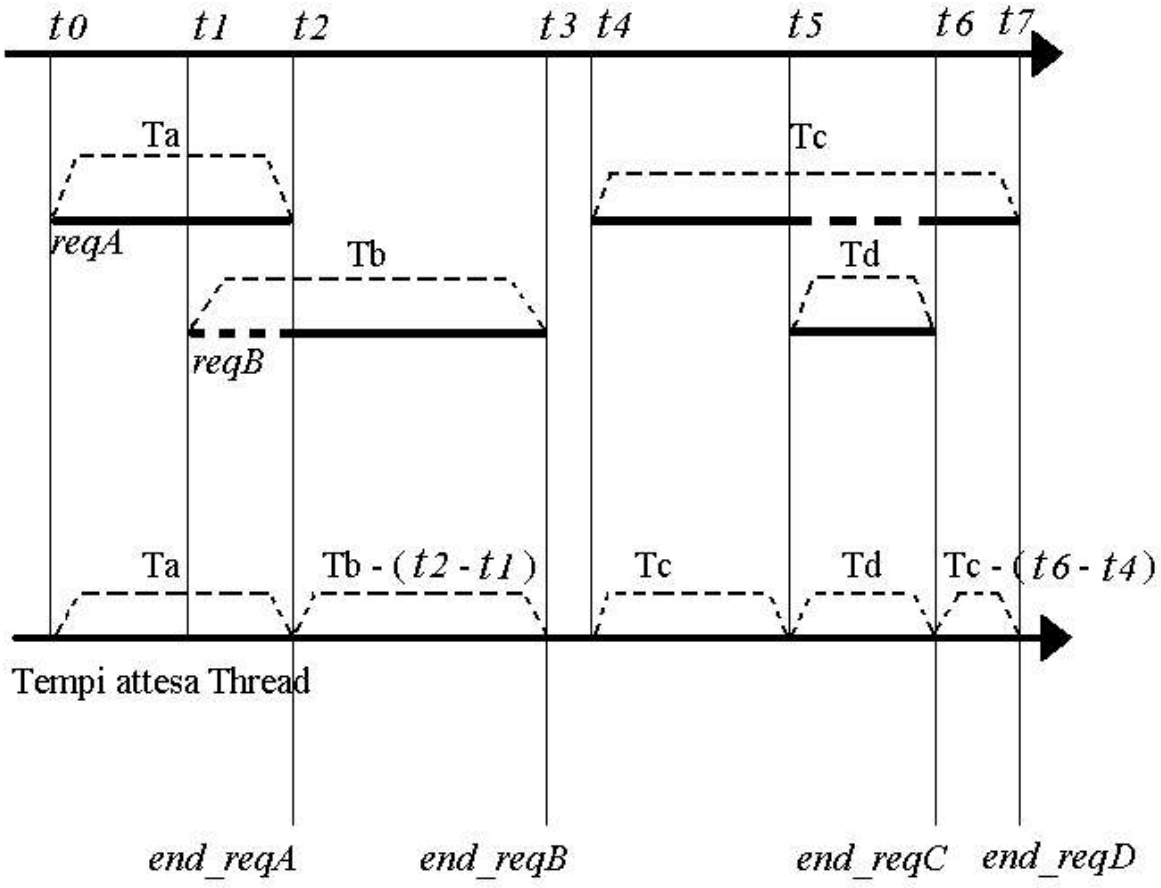

Fig4.2 – Schema del comportamento dello Scheduler nel caso di richieste concorrenti

Risulta quindi chiaro che lo scheduler non deve avere ritardi e al momento voluto dalla varie richieste deve richiamare opportunamente l'inizio dello streaming da server. Lato client non si

devono percepire latenze e il flusso dovrebbe essere continuo, come se l'intero video provenisse da una singola sorgente.

Ragionando sull'architettura ipotizziamo che le richieste vengano fatte su uno *SchedulerManager* che si occupa dell'architettura a livello generale organizzando le richieste all'interno di una lista ordinata e gestendo il sistema come spiegato prima secondo le casistiche considerate. In secondo luogo servirà un semplice componente detto *SchedulerCore* che continuamente prende le richieste dalla lista e aspetta per un tempo opportuno inoltrando poi la richiesta al server per far partire lo streaming.

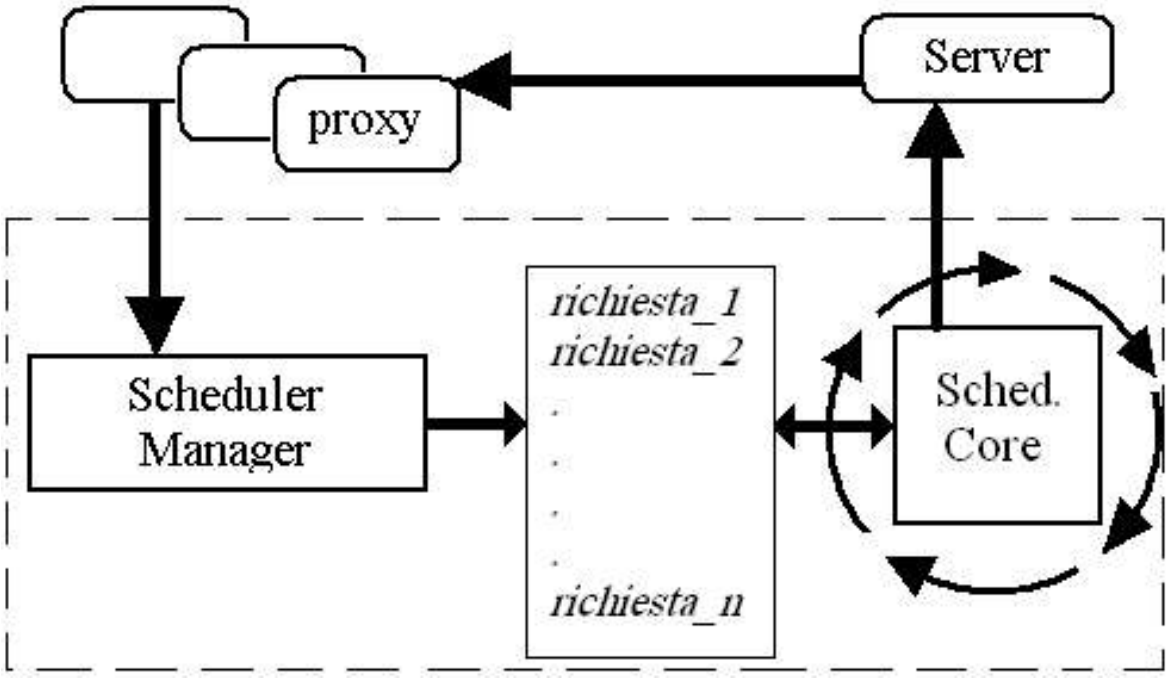

Fig4.3 – Schema progettuale dell'architettura dello Scheduler

Nel modello che abbiamo spiegato, quindi, le richieste vengono soddisfatte nei tempi da loro previsti; è comunque possibile realizzare lo *SchedulerCore* secondo un modello differente detto a *Pollong*.Il *Polling* è basato sulle verifiche ripetute dello stato di una chiamata dall'interno del ciclo dello *SchedulerCore* e costituisce la soluzione meno efficiente per la gestione delle richieste, poiché comporta lo spreco di risorse, a causa delle verifiche ripetute dello stato delle varie richieste. Risulta quindi più opportuno un modello a sincronizzazione come quello spiegato poco prima in quanto alleggerisce tantissimo il sistema in termini di *Overhead*.e consente

di soddisfare richieste numerose in modo efficiente e meno pesantemente di una soluzione a *Polling*.

4.2 Progetto dei componenti per la gestione del Merging

Come abbiamo largamente introdotto precedentemente, attraverso l'SBatch fondiamo prefisso e suffisso (quindi due diversi file) da due host remoti in un unico video lato client; è perciò necessario progettare una struttura adeguata che esegua il *merging* dei due video presenti in remoto.

Decidiamo perciò di operare a livello di frames grazie ad una serie di plugins istanziati in modo opportuno per la gestione della fusione delle parti, in particolare:

1 – *Multiplexer* – che si occupa di fondere i frames di prefisso e suffisso che gli vengono passati.

2 – *Parser* – che prende i frame provenienti dai DataSource di prefisso e suffisso e li passa alla catena.

3 – *Transcoder* – implementato con una serie di funzioni per la codifica e la transcodifica dei frames.

4 – *Sender e Receiver* – per la recezione e l'invio dei flussi da server e proxy al client.

In dettaglio vediamo che una generica sorgente che deve giungere al client passa prima attraverso un *parser* poi ad un *encoder* e infine arriva al client stesso. Ipotizziamo di avere due strutture identiche (come sotto mostrato) che ricevono la prima, il prefisso, la seconda il suffisso e le indirizzano,frame dopo frame ordinatamente, ad un'apposito componente che ne ricava un solo flusso. Avremo quindi bisogno di uno switch che deve essere posto proprio alla fine consentendo di passare al ricevimento dei frame di suffisso quando tutti quelli di prefisso sono stati salvati e quindi di fare il Merging dei due flussi. Organizziamo quindi 3 blocchi:

1 – *PrefixChainer* per ricevere e passare i frame dal Datasource di

prefisso.

2 – SuffixChainer per ricevere e passare i frame dal Datasource di suffisso.

3 – *MergingManager* per svolgere il merging dei frame ricevuti dai due componenti precedenti.

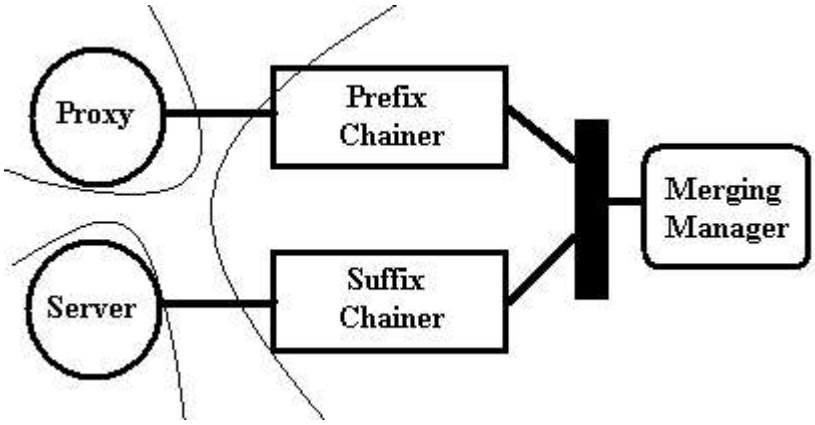

Fig4.4 – Schema progettuale dei componenti per la realizzazzione del merging

Appena è possibile fare il download del prefisso da proxy si istanziano i componenti che accoglieranno i flussi, e si schedula la richiesta di suffisso allo Scheduler.

4.3 Progettazione del *Caching Grain*

Per la realizzazione dell'SBatch necessitiamo che la cache non sia organizzata in una serie di file in modo monolitico, ma che sia divisa in tanti slots che nel capitolo precedente abbiamo chiamato unità di c*aching grain*. In questo modo potremo esprimere tutto ciò che riguarda la cache del proxy (quindi il suo spazio, le dimensioni dei prefissi ...) avendo una unità di misura definita e standard.

Dal punto di vista dell'implementazione la soluzione ottimale è quella di ragionare su due livelli: per ogni presentazione manteniamo un unico file senza operare frammentazioni, ma dal punto di vista logico vediamo la memoria della cache come divisione in slots tutti uguali. Ipotizziamo quindi che la divisione in slots possa essere mantenuta in un metadato come astrazione logica. Infine, per comodità, consideriamo ogni slot espresso in numero di frame: questo è comodo dato che i nostri video sono a bit-rate

costante ed è poi facile passare da frame a byte considerando ogni slot come somma della quantità di byte di ogni frame.

Quando un nuovo file giunge alla cache si occupano gli slot necessari immediatamente però nel caso non ci sia un numero sufficiente di slots liberi allora dapprima viene calcolato il numero di slots che devono essere rimpiazzati e quindi si occupano quelli assegnati alla presentazione. Due parametri guidano la scelta degli slot da liberare: il primo è quello del valore di priority e il secondo riguarda la porzione di frames che sono memorizzati in un dato slot. Si ricercano prima tutti gli slots che hanno il valore di priority più basso poi tra questi si sceglie quello che alloca gli ultimi frame del file, questa convenzione viene stabilita per garantire sempre che, nel caso ci sia un video su proxy, questo sia un prefisso utile.

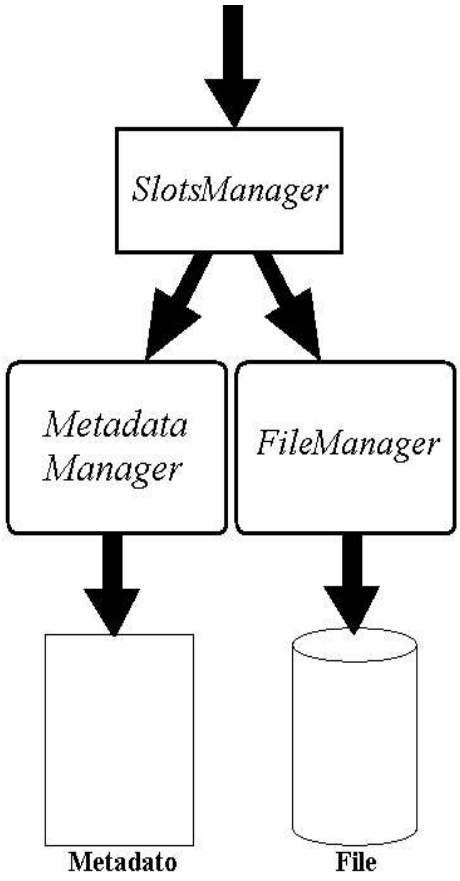

Fig4.5 – Schema progettuale dei componenti per la gestione del Caching Grain

Organizziamo il progetto su due livelli (come mostrato nel disegno precedente), il primo composto da un componente che ha il ruolo di gestire le operazioni ad alto livello, mentre a basso livello

manteniamo i componenti che si occupano delle operazioni su file e su metadato.

Ad ogni azione dello *Slots Manager* corrisponde una reazione sui due componenti sottostanti; ad esempio nel caso sia necessario liberare alcuni slots un comando opportuno partirà dallo *Slots Manager* al *Matadata Manager* e al *File Manager* i quali eseguiranno l'operazione richiesta. In questo caso il *Metadata Manager* cancellerà i nodi relativi agli slots da eliminare e parallelamente il *File Manager* interverrà sul file di prefisso cancellando i frames relativi agli slots occupati.

4.4 Modifiche al *Metadata Browser*

Come è stato appena spiegato i file salvati in locale e su proxy vengono visti attraverso una divisione astratta in *Caching Grain*, quindi è necessario aggiungere un componente *Slot Seeker* che permetta, parallelamente al Metadata Browser di MUMOC, di ottenere le informazioni riguardo alla presenza di prefisso su proxy.

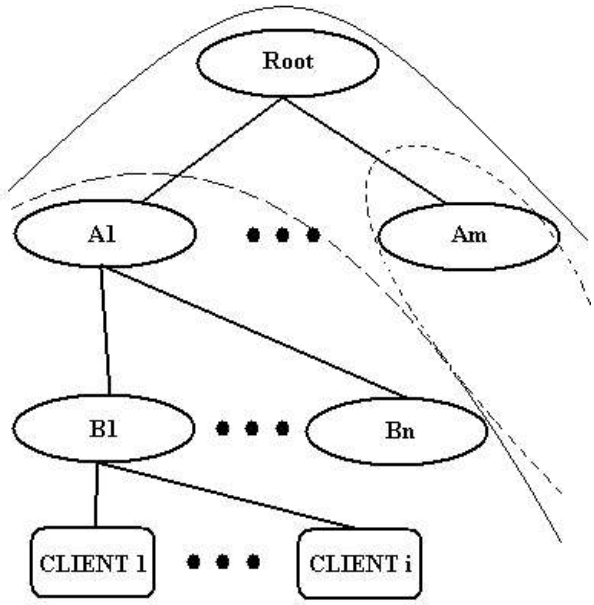

Fig4.6 – Rappresentazione della distribuzione del Metadata Browser di MUMOC

Ogni accesso ad un nodo dell'albero di MUMOC da parte del Metadata Browser viene accompagnato dall'intervento dello *Slot Seeker* quindi da una sua ricerca nel metadato memorizza la partizione logica della cache in slots. Il risultato sarà quello di

riportare l'eventuale presenza di prefisso con le informazioni riguardo i frame quindi al numero di essi memorizzato in cache. Per comodità possiamo ipotizzare che le operazioni di scansione dello *Slot Seeker* si appoggino su quelle già implementate dal *Metadata Manager* e quindi presente su ogni nodo del sistema.

4.5 Gestione delle priorità

Nel progetto della cache in slots abbiamo sottolineato che la priorità sarà assegnata esternamente quindi è necessario un componente che si occupi di questo problema. La soluzione proposta da [1] prevede che la priorità venga calcolata secondo il prodotto tra la grandezza del video e la sua *Request Rate*. Questi valori sono facilmente rintracciabili: il primo in quanto caratteristica intrinseca del video e il secondo grazie ad un monitoraggio sul sistema. A questo punto sarà necessario predisporre dei metodi adeguati sul componente *Metadata Manager* che consentano di modificare dinamicamente la priorità del video a seconda che la *Request Rate* nel tempo si alzi o si abbassi.

Occorre tuttavia considerare un altro aspetto, cioè quello che video molto grandi (quindi con priorità alta) rischiano di occupare stabilmente la cache del proxy anche se poco richiesti; è quindi necessario aggiungere un metodo che assegni la priorità rispetto al tempo, in particolare la abbassi a tutti gli slots alla fine di un intervallo fissato. Infine occorre che il componente modifichi le priorità anche in caso di variazioni di *Request Rate* di un dato video alzando o abbassando la priorità correttamente.

4.6 Analisi del funzionamento completo

Mostriamo ora il funzionamento completo dell'SBatch all'arrivo di una richiesta del client.

Come mostrato nell'esempio soprastante un client fa una richiesta sul proxy che di conseguenza comanda l'avvio del Metadata Browser. Questa operazione consente di riconoscere (grazie allo *SlotsSeeker*) la presenza del prefisso del file richiesto sulla cache del proxy e quindi di iniziare la trasmissione del suddetto prefisso al nodo del client attraverso *PrefixChainer*. Contemporaneamente

viene schedulata la richiesta dei rimanenti frame di suffisso allo scheduler il quale al momento opportuno comanderà lo streaming dal server al client attraverso *SuffixChainer*.

In caso di uno scenario con molti clients e proxy i componenti sono semplicemente ripetuti escluso lo Scheduler e il Metadata Browser che rimangono unici nel intero sistema.

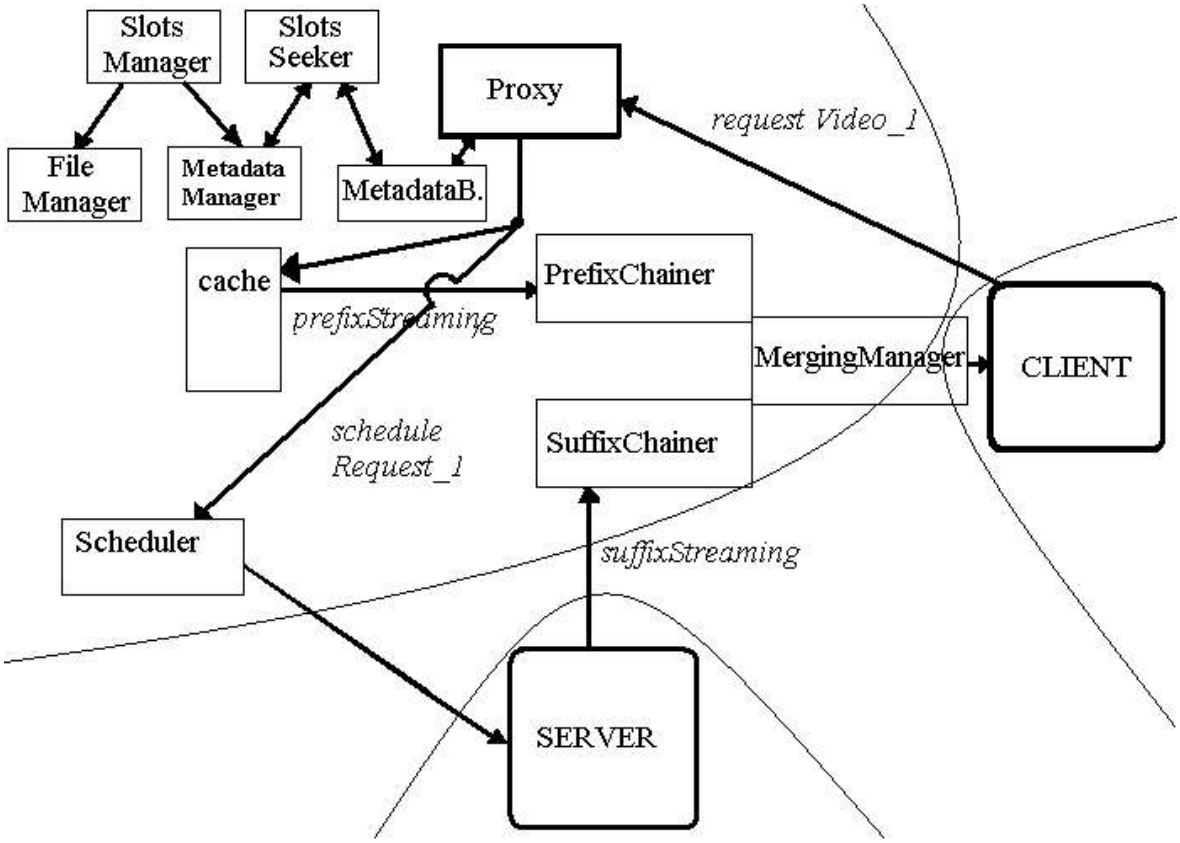

Fig4.7 – Schema progettuale di tutti i componenti che realiazzano l'SBatch

Abbiamo quindi fissato gli schemi progettuali di un SBatch da integrare nella piattaforma MUMOC, nel prossimo capitolo vedremo come vengono implementate queste stesse parti.

- CAPITOLO V -
IMPLEMENTAZONE DI UN SBATCH PER MUMOC

5.1 Implementazione dello *Scheduler*

Abbiamo visto quale sia l'importanza dello scheduler in quanto detiene la responsabilità di gestire l'inizio delle trasmissioni di suffisso da server servendo correttamente le richieste dei clients.

A livello implementativo definiamo una classe `Request` che si occupa di mantenere una generica richiesta da parte di un cliente; essa in particolare consta dei seguenti campi :

`String` `idClient` : rappresenta un semplice identificativo del client al quale bisogna inviare il prefisso.

`Metadata` `movie` : un identificativo della presentazione; vedremo nel prossimo capitolo il significato del metadato in MUMOC

`long` `timeToSleep` : il tempo dopo il quale il client deve ricevere il prefisso.

`long` `timeOfRequest` : memorizza il momento nel quale lo scheduler riceve la richiesta.

`long` `endTime` : valore calcolato dalla classe `Request` al momento dell'inizializzazione di una nuova istanza e rappresenta il momento nel quale è previsto l'invio di streaming di prefisso. Questo valore vine calcolato come somma tra `timeOfRequest` e `timeToSleep`.

Le prime tre costanti vengono definite dal proxy quando comanda la richiesta sullo scheduler le altre due sono calcolate dallo Scheduler stesso.

Ogni richiesta viene fatta su uno *SchedulerManager* il quale si

basa sui seguenti metodi fondamentali :

public void `setRequest(`**String** `c,` **Metadata** `m,` **long** `t)` – Questo metodo accetta una richiesta definita dai primi tre campi sopra esposti ; il metodo salva anche il momento attuale (cioè quello nel quale avviene la richiesta) e crea quindi un nuovo oggetto `Request`. Tutte le `Request` sono memorizzate in un **Vector** in ordine di `endTime`. In particolare quando una `Request` viene creata si controlla se lo scheduler ne sta già soddisfacendo un'altra, in caso positivo si controllano gli `endTime` di entramebe; se la nuova richiesta può essere soddisfatta subito, la si inserisce nel primo posto del **Vector** delle richieste, in caso contrario si invoca il metodo **public void** `insertRequest (`**Request** `req)` che ha lo scopo di inserire nel **Vector** la richiesta in ordine di `endTime`.

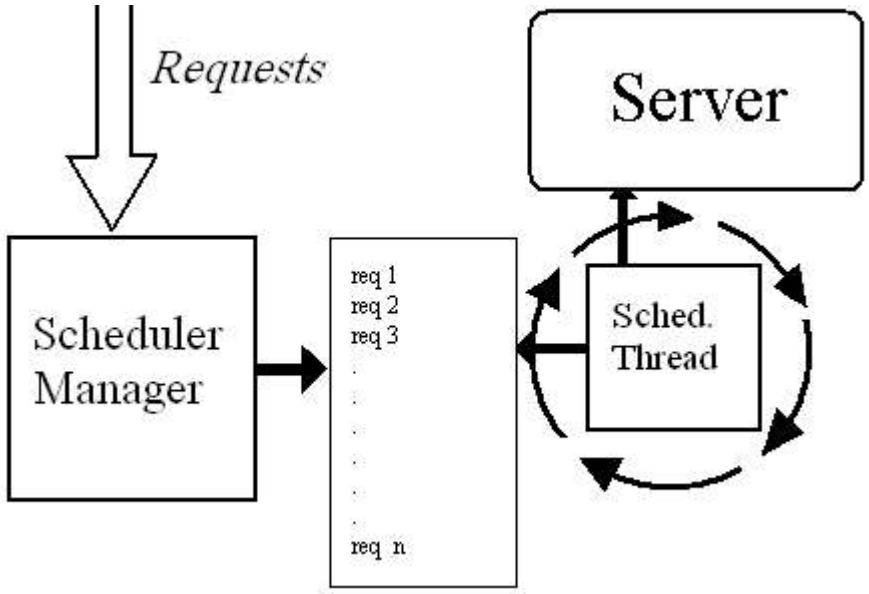

Fig5.1 – Schema implementativo dello Scheduler

A questo punto implementiamo un Thread che continuamente controlla se ci sono richieste nella lista e in caso affermativo prende la prima e accede ad un metodo dello *SchedulerManager* il quale fa dormire il Thread stesso per un tempo definito dalla `timeToSleep` della richiesta considerata. E'quindi chiaro che il Thread si sveglierà solo in due casi :

1 – Ha dormito per un numero di millisecondi definito da `timeToSleep` quindi comanda al server l'invio del prefisso al client che ha fatto la richiesta.

2 – E'giunta una richiesta che può essere considerata prima di quella attuale ; il Thread si sveglia sotto comando `notifyAll()` (comandato dello *SchedulerManager* stesso), reimposta di tempi della richiesta che smette di considerare salvando il numero di millisecondi che ha dormito poi considera la nuova richiesta interamente.

E' importante sottolineare che se la richiesta non deve essere soddisfatta subito il Thread non viene bloccato, comunque si tiene traccia di quanti millisecondi abbia dormito soddisfacendo l'altra richiesta dal momento che la nuova è giunta.

I metodi tengono traccia dinamicamente dell'evoluzione delle richieste in modo che tutte possano essere considerate in modo corretto e nei tempi giusti.

Ogni volta che una richiesta viene soddisfatta, oltre a comandare lo streaming da server, la si cancella dal **`Vector`** e si considera la successiva.

Siccome non è possibile far partire esattamente lo streaming di suffisso al momento definito dal client a causa di piccole latenze (in termini di millisecondi) sia nelle trasmissioni che nei metodi, può risultare utile decrementare leggermente le `timeToSleep` di tutte le richieste di un certo valore per prevenire i sopracitati ritardi. Questo valore può essere facilmente determinato attraverso testing o monitoraggio del sistema.

5.2 Implementazione dei componenti per la realizzazione del Merging

Nel precedente capitolo abbiamo introdotto lo scopo delle catene di plugins per la realizzazione del Merging tra due file remoti e come organizzare questi plugins attraverso *PrefixChainer*, *SuffixChainer* e *MergingManager*; ora vediamo i dettagli implementativi costruiti organizzando componenti realizzati da Ale Falchi.

Innanzitutto interfacciamo i diversi plugins attraverso dei buffer detti *CircularBuffer*: in particolare essi sono formati da una "ruota" di buffer che vengono riempiti da un componente e fatti girare affinchè un'altro possa leggere da essi. Questa soluzione rappresenta una buona soluzione per la gestione di un flusso continuo di frame tra plugins. Alla luce di ciò i nostri plugins saranno organizzati come segue:

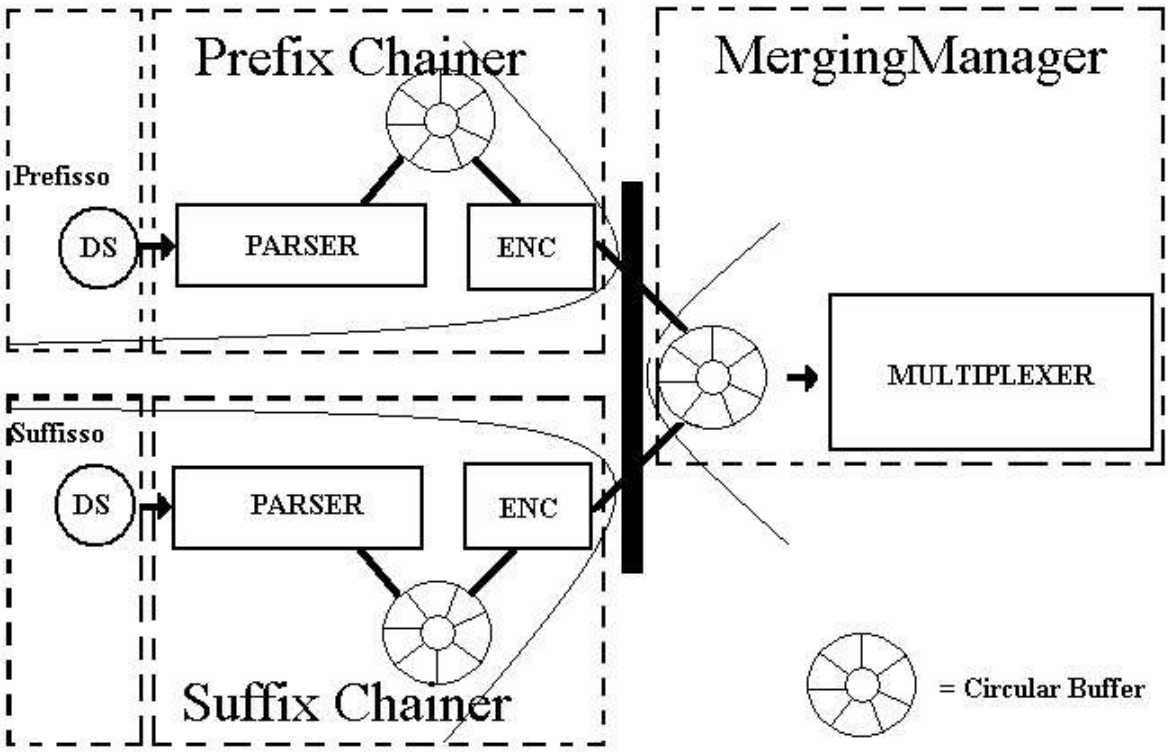

Fig5.2 – Schema implementativo dei componenti che realizzano il Merging

Partiamo con la realizzazione di una classe *DownloadManager* che si occupa del download da remoto dei DataSource di prefisso e suffisso. In particolare essa è composto dalle seguenti applicazioni :

`public class` `ReceivePrefixThread` **`extends`** `Thread` – Un Thread che si occupa di ricevere il DataSource del prefisso dal proxy attraverso un *RTPReceiver* adeguatamente inizializzato.

`public class` `ReceiveSuffixThread` **`extends`** `Thread` – Un Thread che si occupa di ricevere il DataSource del suffisso dal server attraverso un *RTPReceiver* adeguatamente

inizializzato.

I DataSource così ricevuti vengono visti come sequenza di frames e quindi fatti passare attraverso una opportuna catena di plug-ins come spiegato nel precedente capitolo e come verrà ripreso tra breve. Questo trattamento consente di mostrare al client un flusso continuo eliminando qualsiasi distinzione tra prefisso e suffisso, cioè come se sul nodo del client arrivasse apparentemente un DataSource di un solo video intero e non di due parti. Quindi implementiamo i seguenti Thread :

1 - **public class** `PrefixChainerThread` **extends** `Thread` composto dai metodi :

public void `initChainPrefix()` - Si occupa di inizializzare i componenti della catena del prefisso (un *Parser*, un *Circular Buffer* e un *Encoder*)

public `ExtBuffer chainFrame()` - Prende un frame dal DataSource del prefisso e lo passa attraverso la catena restituendolo pronto per il merging.

public void `run()` - Il metodo centrale del Thread : passa tutti i frames del prefisso attraverso la catena di plug-ins.

2 - **public class** `SuffixChainerThread` **extends** `Thread` composto dai metodi :

public void `initChainSuffix()` - Si occupa di inizializzare i componenti della catena del suffisso (un *Parser*, un *Circular Buffer* e un *Encoder*)

public `ExtBuffer chainFrame()` - Prende un frame dal DataSource del suffisso e lo passa attraverso la catena restituendolo pronto per il merging.

public void `jumpToPrefixFrame()` - Siccome il file presente sul server è completo (prefisso + suffisso) questo metodo

salta tutti i frames del prefisso (precisamente legge e non cosidera un numero di frame pari a quello presente nel prefisso) e si prepara ad accogliere, quindi, solo i frame del suffisso.

public void `run()` – Il metodo centrale del Thread : passa tutti i frames del suffisso attraverso la catena di plug-ins.

3 - **public class** `MergingManager` – Una classe con lo scopo di prendere i frames che gli vengono passati dalle catene di plug-ins per poi disporli in un DataSource che risulti la fusione di prefisso e suffisso attraverso un *Circular Buffer* e un Multiplexer. Ovviamente anche questa classe consta di tutti i metodi per l'inizialiazzazione e il merging dei frames.

A gestire le seguenti applicazioni interviene un `CoreManager` che ha lo scopo di inizializzare il tutto e far partire in modo opportuno la sequenza (come vedremo dettagliatamente spiegato tra breve).

Tutte queste classi sono state implementate come Thread per poter sfruttare una certa simultaneità. In particolare al momento della arrivo della richiesta del client viene immediatamente attivato il download e il filtraggio dei frames di prefisso dal proxy, contemporaneamente si si attende il comando di download di suffisso di modo che, finito il ricevimento dal proxy del prefisso si possa fare il merging dei frame di suffisso senza latenze. Come spigato nei precedenti paragrafi, il compito di mandare (attraverso RTPSender) il DataSource di suffisso e di comandare l'inizio effettivo del filtraggio e (quindi del merging), è dello Scheduler.

5.3 Realizzazione dei componenti per la gestione del *caching grain*

Come abbiamo visto nel capitolo precedente manteniamo una partizione logica degli slots in cui abbiamo diviso la cache in un metadato e lo facciamo quindi utilizzando un file XML (un formato già fondamentale per i metadati di MUMOC) che viene salvato nella directory della cache locale assieme ai file di prefisso. Il file XML

LocalCache.xml è formato di un nodo <cache> principale che contiene le informazioni sul numero di slots in cui la cache è divisa e il numero di frame che forma ogni singolo slot; la prima cosa che viene fatta all'attivazione del programma è quella di inizializzare il file con queste due primarie informazioni. In oltre <cache> contiene tanti sotto-nodi quanti il numero di slots da cui è formata. Ogni slots mantiene le informazioni della parte di file che contiene, quindi il numero di frame che memorizza (al massimo della dimensione dello slot) e un valore di priorità determinato da un'apposito componente che vedremo tra breve in dettaglio. Per poter mantenere questo sistema di divisione in slot indipendente dalla nostra progettazione è stato deciso di far calcolare il valore di priority direttamente dal SBatch lasciando quindi una certa indipendenza e riusabilità a questa parte del sistema. Quindi un generico **LocalCache.xml** appare come segue :

```
<cache number="100" size="100">
  <slot priority="22" file="StarWars.mov" framestart="1" framestop="100" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="101" framestop="200" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="201" framestop="300" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="301" framestop="400" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="401" framestop="500" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="501" framestop="600" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="601" framestop="700" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="701" framestop="800" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="801" framestop="900" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="901" framestop="1000" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1001" framestop="1100" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1101" framestop="1200" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1201" framestop="1300" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1301" framestop="1400" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1401" framestop="1500" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1501" framestop="1600" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1601" framestop="1700" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1701" framestop="1800" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1801" framestop="1900" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="1901" framestop="2000" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="2001" framestop="2100" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="2101" framestop="2200" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="2201" framestop="2300" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="2301" framestop="2400" numberofframes="18243" />
  <slot priority="22" file="StarWars.mov" framestart="2401" framestop="2459" numberofframes="18243" />
</cache>
```

Fig5.3 – Esempio di file LocalCache.xml per la memorizzazione degli slots della cache del proxy

A questo punto dovremo implementare i tre componenti fondamentali che sono stati mostrati nel capitolo precedente, in particolare:

SlotsManager – Che gestisce la cache ad alto livello

XmlManager – Che gestisce le operazioni sul metadato XML

FileManager – Che gestisce le operazioni sui file

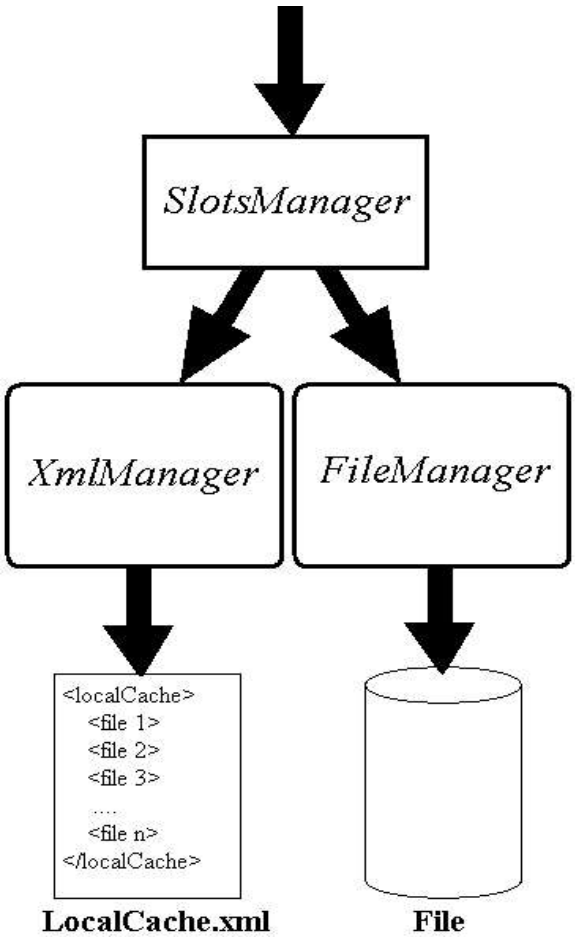

Fig5.4 – Schema implementativo dei componenti che gestiscono la divisione in slots della cache del proxy

Ogni livello consta di una classe formata da diversi metodi:

a. SlotsManager

Organizza ad alto livello la cache grazie ad alcuni metodi dei quali i principali sono :

public static void init(**int** number, **int** size) Inizializza la cache recuperando i dati presenti nella cache locale oppure istanziando una nuova cache definita in termini di numero di slots (number) e dalla grandezza di ogni slot (size) (Queste informazioni sono decise a priori prima del'avvio del sistema).

public static void insertFile(String fileName, **int** frameStart, **int** frameStop, **int** size, **int** requestRate) – Occupa gli slot occorrenti per il video fileName definito nei limiti di frame frameStart – frameStop (solitamente questi due valori rappresentano il primo ed ultimo frame di un video, o di un prefisso). Il valore di priorità, come detto precedentemente viene determinato direttamente dalla politica di SBatch attraverso la requestRate e la grandezza del video da un'apposito componente. Nel caso ci siano sufficienti slot liberi per il video si procede con il normale allocamento ; nel caso contrario si occupano i nodi liberi disponibili e si sostituiscono quelli caratterizzati dai valori di priority più bassi finchè tutto il video non è salvato. L'algoritmo non è attualmente ottimizzato nel senso che non è possibile ancora inserire frame di files diversi su uno stesso slot ; il problema non risulta comunque determinante, infatti gli slot di cache sono solitamente di pochi secondi, quindi anche in caso di un riempimento non ottimale si avrebbero spazi inutilizzati di pochi K. (l'intero modulo è comunque progettato per essere facilmente modificato e migliorato)

b. XmlManager

Gestisce le operazioni di slot direttamente sul document XML. Comprende le funzioni di *CacheManager* implementate ad un livello più basso :

public static void init(**int** number, **int** size) – Se non esiste già crea un nuovo documento **LocalCache.xml** e istanzia il nodo <cache> con attributi number e size (cioè numero di slots e loro grandezza).

public static void setSlot(**int** priority,

`String fileName, int frameStart, int frameStop)` - Occupa uno slot scrivendo un tag <slot> definito con gli attributi che descrivono il file (o frammento di file) salvato (cioè la sua priorità, il suo nome e il numero di frames che lo costituisce).

public static void `setAndReplaceSlots(int priority, String fileName, int frameStart, int frameStop)` – Occupa uno slot scrivendo un tag <slot> definito con gli attributi che descrivono il file (o frammento di file) salvato sostituendo lo slot caratterizzato da priorità più bassa. Questa operazione avviene quando un nuovo file giunge alla cache del proxy.

c. FileManager

Siccome l'organizzazione a slot della cache è definita solo ad alto livello, i metodi sui file risultano pochi :

public static int `determinateMaxFrame(String fileName)`- Determina il numero di frames che compone un determinato video; questa informazione viene mantenuta in memoria nel file XML non tanto per la sua frequente utilità ma solo perchè è una informazione costosa. Infatti per ottenere il numero di frame di una DataSource bisogna scansionare tutta la sorgente e contare dal primo all'ultimo frame. E' quindi chiaro che che se si deve accedere a questa funzione più di una volta è molto meglio ricavare il dato da **LocalCache.xml**.

public static void `Init(String Input, String OutPut, long Start, long Stop)` - Taglia dal file i frame che occupano un certo slot dando come risultato un nuovo file privo dei frame tagliati. Semplicemente la sorgente viene demultiplexata e ogni traccia viene decodificata : la traccia video viene trattata come sequenza di H263 (formato di codifica standard in MUMOC) che viene attraversata tutta e per ogni immagine si decide se salvarla o no nel file finale a seconda se si trovi o no tra `Start` o `Stop`. Siccome ci interessa avere sempre un prefisso

utile il frame di fine taglio `Stop` coincide, nel nostro caso, con l'ultimo frame del video. La classe è stata comunque implementata con la possibilità di tagliare anche anche sezioni di frame interne al video (quindi non solo code, come abbiamo deciso) nel caso di un eventuale riuso in altre architetture o altre applicazioni.

5.4 Implementazione dello *SlotSeeker*

Lo *SlotSeeker* ci permette di determinare la presenza di prefissi allocati nella cache del proxy sfruttando il motore del Metadata Browser. Siccome il componente comanda una lettura sul metadato **LocalCache.xml**, si appoggia alla classe *XmlManager* illustrata poco fa sfruttando i suoi metodi per la scansione del DOMTree del documento Xml.

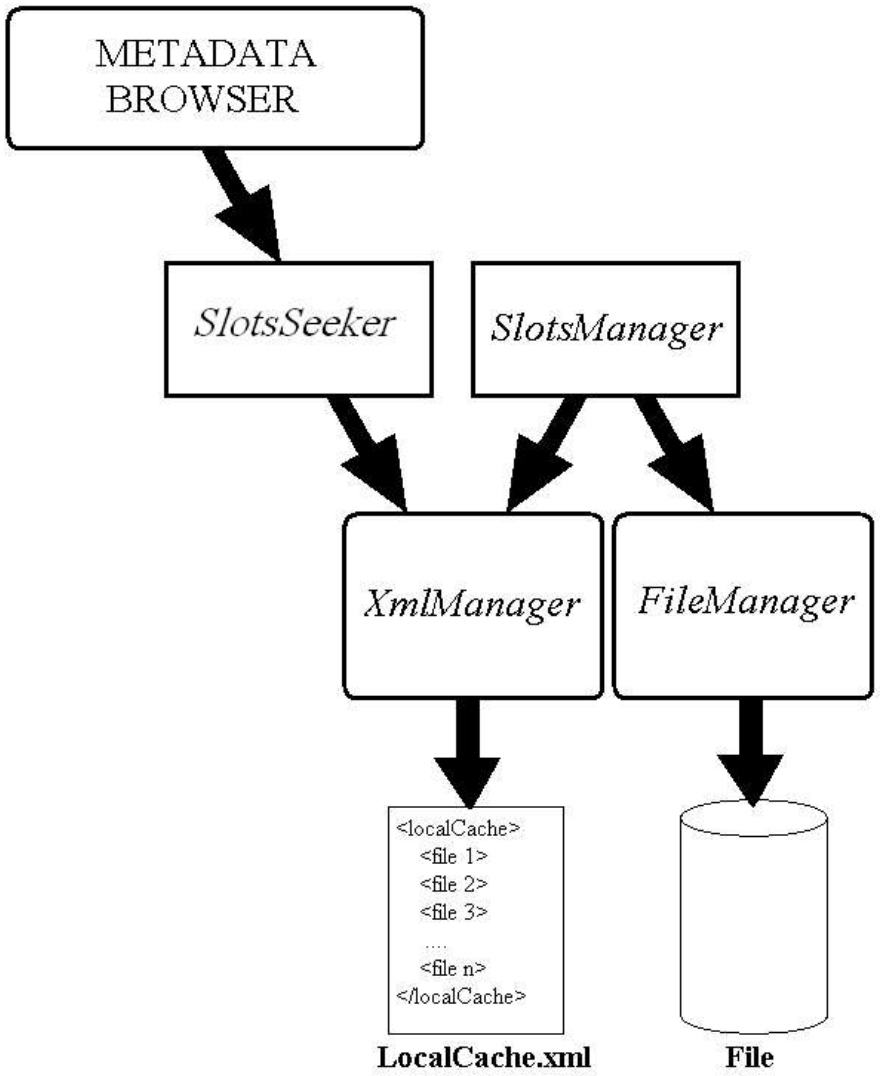

Fig5.5 – Schema implementativo dello SlotsSeeker

Innanzi tutto viene aggiunta alla classe *XmlManager* il metodo :

public static int `getFile(String fileName)`- Dato un certo `fileName` il metodo svolge una ricerca nel metadato **LocalCache.xml** riportando il numero di frame del prefisso richiesto, 0 in caso contrario.

La classe *SlotSeeker* richiama il metodo `getFile(String fileName)` appena illustrato attraverso il metodo

public static int `serchFile(String fileName)`

Il Metadata Browser non fa altro che richiamare il metodo precedente ad ogni ricerca sui nodi dei proxy segnalando quindi la presenza di sots di prefisso richiesti. Nel caso la ricerca abbia avuto successo si abilita il download del prefisso dal proxy immediatamente altrimenti si comanda il download dell'intero video da server (come già implementato in MUMOC).

5.5 Implementazione dei componenti per la gestione delle priorità

Come abbiamo detto nel paragrafo riguardante i componenti per la gestione della *caching grain* è necessario implementare un componente esterno per settare la priorità di ogni file che arriva. Utilizziamo allora un *PriorityManager*, una classe che viene inizializzata dallo *SlotsManager* nel metodo di inizializzazione della cache e implementa i metodi principali per l'assegnamento della priorità :

public static int `detPriority(int requestRate, int size)` – determina la priorità di un file a seconda della sua Request Rate e della sua grandezza; questi valori vengono passati assieme alle altre informazioni al momento di ricezione di un nuovo file su proxy.

public static void `lowPriority(int x)` – abbassa la

priorità di tutti i file di un valore x. Questo metodo viene implementato per evitare, come già spiegato precedentemente, che file molti grandi e con bassa richiesta occupino a lungo la cache del proxy.

public static void `changePriority(`**String** `fileName,` **int** `priority)` – quando si modifica la Request Rate di un file è necessario mutare dinamicamente la sua priorità; questo metodo ricerca un file nel metadato **LocalCache.xml** e modifica i valori di priorità con il valore passato.

Tutti questi metodi si appoggiano, come per lo *slotSeeker*, su metodi implementati nell'*XmlManager* per i medesimi motivi.

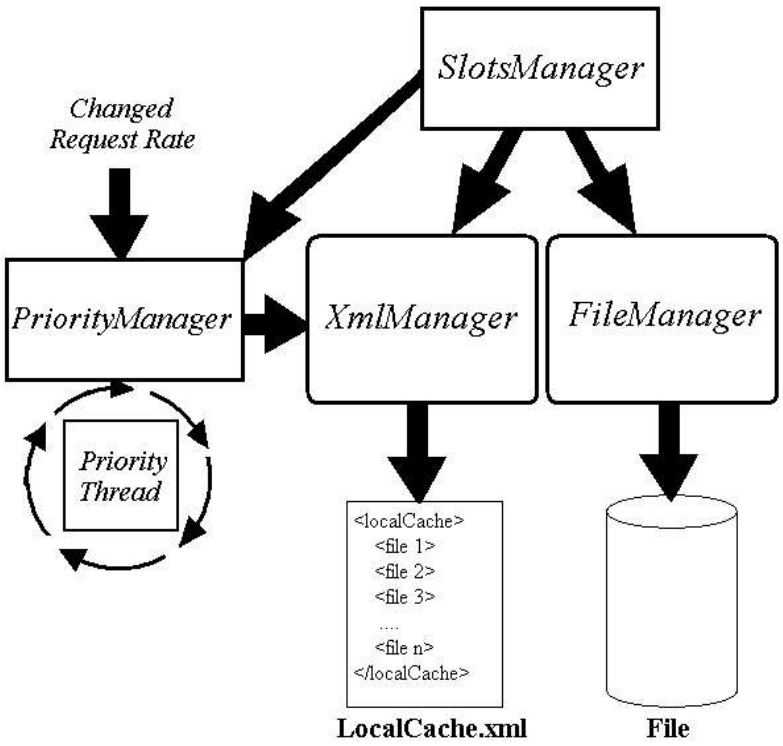

Fig5.6 – Schema implementativo dei componenti per la gestione delle priorità degli slots

A questo punto implementiamo un Thread `priorityThread` che viene inizializzato assieme al *priorityManager* e che ha lo scopo unico di dormire per un tempo definito e di richiamare, al suo risveglio, il metodo `lowPriority` con un valore predefinito `x`. In particolare quest'ultimo viene definito nell'inizializzazione del

Thread stesso. Questo componente quindi permette una corretta gestione delle priorità in modo dinamico e preciso.

- CAPITOLO VI -
INTEGRAZIONE DELL'SBATCH NELL'ARCHITETTURA DI MUMOC

6.1 Download del codice

Dopo aver implementato esternamente le classi che realizzano i componenti appena illustrati passiamo all'integrazione dell'SBatch nell'architettura MUMOC.

Siccome stiamo lavorando su un sistema distribuito dovremo sfruttare le funzioni di MUMOC per il download del codice (*Deploy*) nei vari nodi, in particolare MUMOC stabilisce un apposito piano detto *Plan* nel quale vengono definite le posizioni delle classi all'interno dell'albero. Questo *Plan* viene scansionato da un componente chiamato *PlanVisitorAgent* il quale, all'inizializzazione del sistema, si muove attraverso i vari nodi e comanda il demploy del codice in modo adeguato. Siccome non tutto il codice deve essere scaricato durante l'inizializzazione di MUMOC è presente un *DecisionMaker*, un componente che ha il ruolo di svolgere il download del codice real-time.

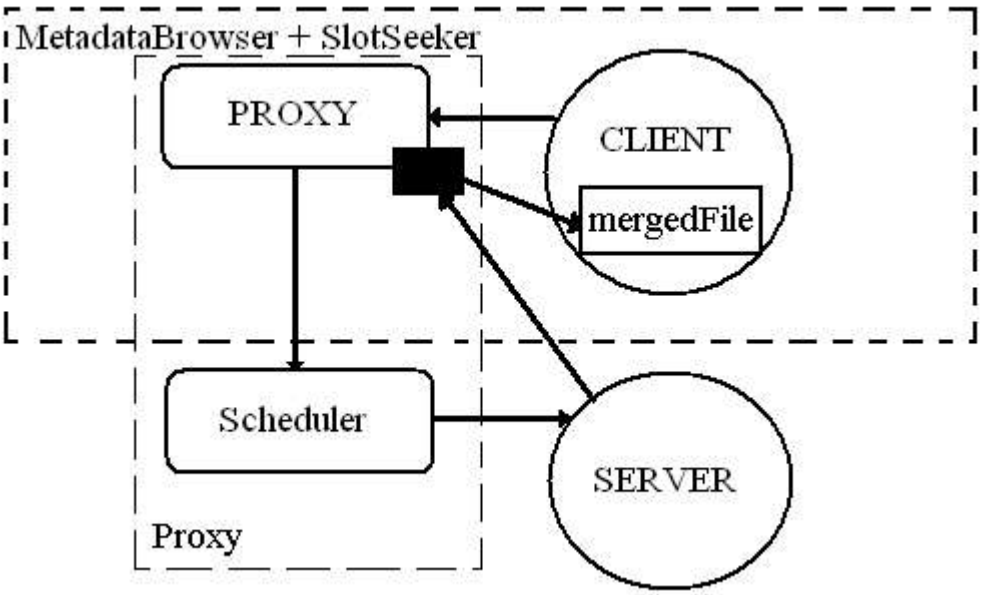

Fig6.1 – Deploy dei componenti nell'architettura Client-Proxy-Server

Per quando riguarda il sistema di SBatch dovremo allocare :

1 – Le classi per la gestione della cache in slots su ogni nodo dell'albero assieme al componente per la gestione delle priorità (attraverso *PlanVisitorAgent*).

2 – La classe *SlotSeeker* assieme alle altre classi del Metadata Browser (attraverso *PlanVisitorAgent*)

3 – Lo Scheduler sul nodo del Proxy (attraverso *PlanVisitorAgent*)

4 – Le classi per la gestione del Merging sul nodo del Proxy in modo real-time affinchè vengano istanziate solo nel caso di un effettivo merging (attraverso *DecisionMaker*).

Infine, siccome MUMOC viene eseguito attraverso il compilatore ***Ant*** il quale compila le classi di inizializzazione che sono elencate in un apposito metadato XML (**build.xml**), sarà quindi necessario inserire nel suddetto file i componenti di SBatch da compilare all'inizializzazione del sistema affinchè le relative classi siano pronte per l'esecuzione al momento opportuno.

6.2 Integrazione delle classi di SBatch

6.2.1 Integrazione dello *Slots Seeker* nel Metadata Browser

Prima di tutto dobbiamo integrare lo *SlotsSeeker* all'interno del Metadata Browser e questo può avvenire facilmente semplicemente utilizzando da tramite i metodi del Metadata Browser. Ad esempio la richiesta di una ricerca di metadato sull'albero corrisponde ad un ricerca del file richiesto attraverso lo *SlotsSeeker* nel metadato degli Slots; stesso discorso per l'inserimento di un nuovo metadato o per la sua cancellazione. Il nostro scopo è quello di sfruttare il motore già implementato del Metadata Browser usando però le funzioni dello *SlotsSeeker*.

Ovviamente sarà necessario stabilire dove si trova il metadato per la divisione della cache in slots (cioè il file **localCache.xml** di cui abbiamo parlato nel capitolo 5 durante l'implementazione di questi

componenti) e questo può essere fatto tranquillamente utilizzando l'interfaccia grafica di MUMOC; in particolare attraverso essa è possibile stabilire il percorso assoluto nel quale è memorizzata la cache di un particolare nodo e questa corrisponderà alla directory nella quale sarà allocata la cache stessa quindi anche il metadato degli slots.

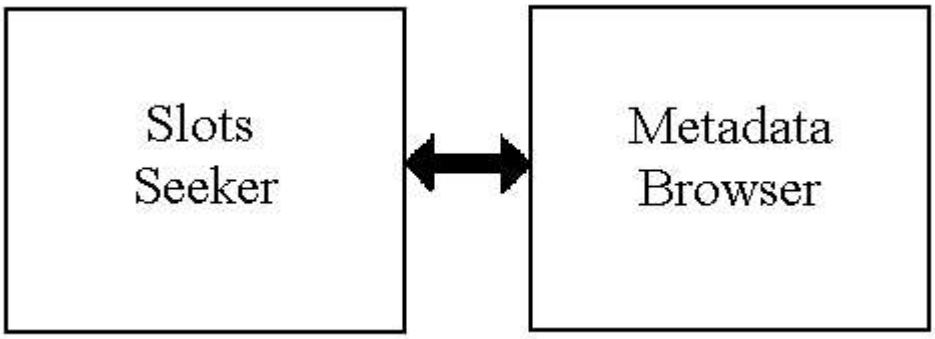

Fig6.1 – Schema di integrazione tra SlotsSeeker e Metadata Browser

Come sappiamo dal funzionamento del Metadata Browser, una ricerca mi da come risultato il metadato del prefisso cercato, ma sappiamo anche che lo *SlotsSeeker* ci permette di ottenere il numero di frames corrispondenti al prefisso considerato; quindi dovremo aggiungere al metadato di MUMOC un campo che conterrà esattamente questo numero e i metodi adeguati di *set* e *get* per settare ed ottenere il valore richiesto.

6.2.2 Integrazione dello Scheduler

L'integrazione dello scheduler nella piattaforma MUMOC risulta immediata in quanto sarà inizializzato assieme a tutti gli altri componenti del sistema (in particolare quando vengono inizializzati i componenti del proxy) e si attiva immediatamente rimanendo pronto per ricevere una richiesta.

Al momento opportuno lo scheduler lancerà un messaggio di **START_STREAMING** al server attraverso socket (sfruttando la *Protocol Unit* di uscita del proxy stesso) secondo le politiche già viste precedentemente; il server quindi invierà il suffisso al proxy (utilizzando i metodi già implementati in MUMOC) il quale svolgerà il merging appropriatamente attraverso i componenti che sono stati appositamente progettati.

6.2.3 Integrazione dei componenti per il Merging

Come abbiamo visto dal capitolo 3 la classe che riceve la richiesta e comanda la ricerca ottenendo il risultato da MetadataBrowser è `SimpleProxyVideoInProtocolUnitNomadicEnabled`. Dopo aver riconosciuto la presenza del prefisso richiesto e quindi la sua allocazione il sistema comanda lo streaming da proxy; prima che questo accada sarà necessario fare richiamare le funzioni per istanziare i componenti implementati per la gestione del *merging* e comandare scheduling del suffisso. Le classi saranno scaricate nel nodo del proxy dal *DecisionMaker* e verrano richiamate dalla `InProtocolUnit` subito dopo la ricezione del risultato della ricerca. Lo scheduling del prefisso è automatico mentre lo scheduler è già avviato dall'inizio del sistema (come introdotto nel paragrafo precedente).

6.3 Analisi del funzionamento generale

Ora possiamo concentrarci sulla panoramica generale vedendo il funzionamento del sistema: come già detto, all'inizializzazione di MUMOC viene scaricato il codice dai vari nodi e avviene l'inizializzazione dei diversi componenti. Tra questi dovremo comandare l'inizializzazione dei componenti per la gestione della cache in slots e dello scheduler in particolare il primo assieme all'inizializzazione del Metadata Browser mentre il secondo assieme ai componenti del proxy (come introdotto poco fa).

A questo punto il sistema è pronto. Ipotizziamo che un client faccia una richiesta (1); un metadato contenente il nome del file parte diretto verso il server. Il proxy più vicino al client e sul tragitto tra client e server intercetta la richiesta e comanda immediatamente l'avvio del Metadata Browser (2). Esso ricerca nei nodi la presentazione richiesta e attraverso lo *SlotSeeker,* nei metadati degli slots, il prefisso corretto. In caso negativo MUMOC schedula una richiesta da server come normalmente succede, in caso positivo si istanziano i componenti per la realizzazione del merging e si schedula la richiesta di suffisso(3). Il DataSource di prefisso viene, intanto, fatto passare attraverso il *PrefixChainer* che, come ricordiamo, prende un frame alla volta della presentazione attraverso

un *parser,* poi lo decodifica e lo passa a un *multiplexer* per il caching. Contemporaneamente lo scheduler sta aspettando per un tempo ipotizzato dal protocollo nel istante dello scheduling della richiesta al server; al momento opportuno (4) avverrà il comando al server di fare lo streaming sfruttando le funzioni già implementate nella stessa piattaforma di MUMOC. In particolare esso invia al proxy il DataSource di suffisso (5) il quale viene preso, come il prefisso, frame per frame e passato al *multiplexer*. Quest'ultimo deve, ovviamente, aver già finito di leggere il prefisso e deve essere pronto a salvare i frame da *SuffixChainer*.

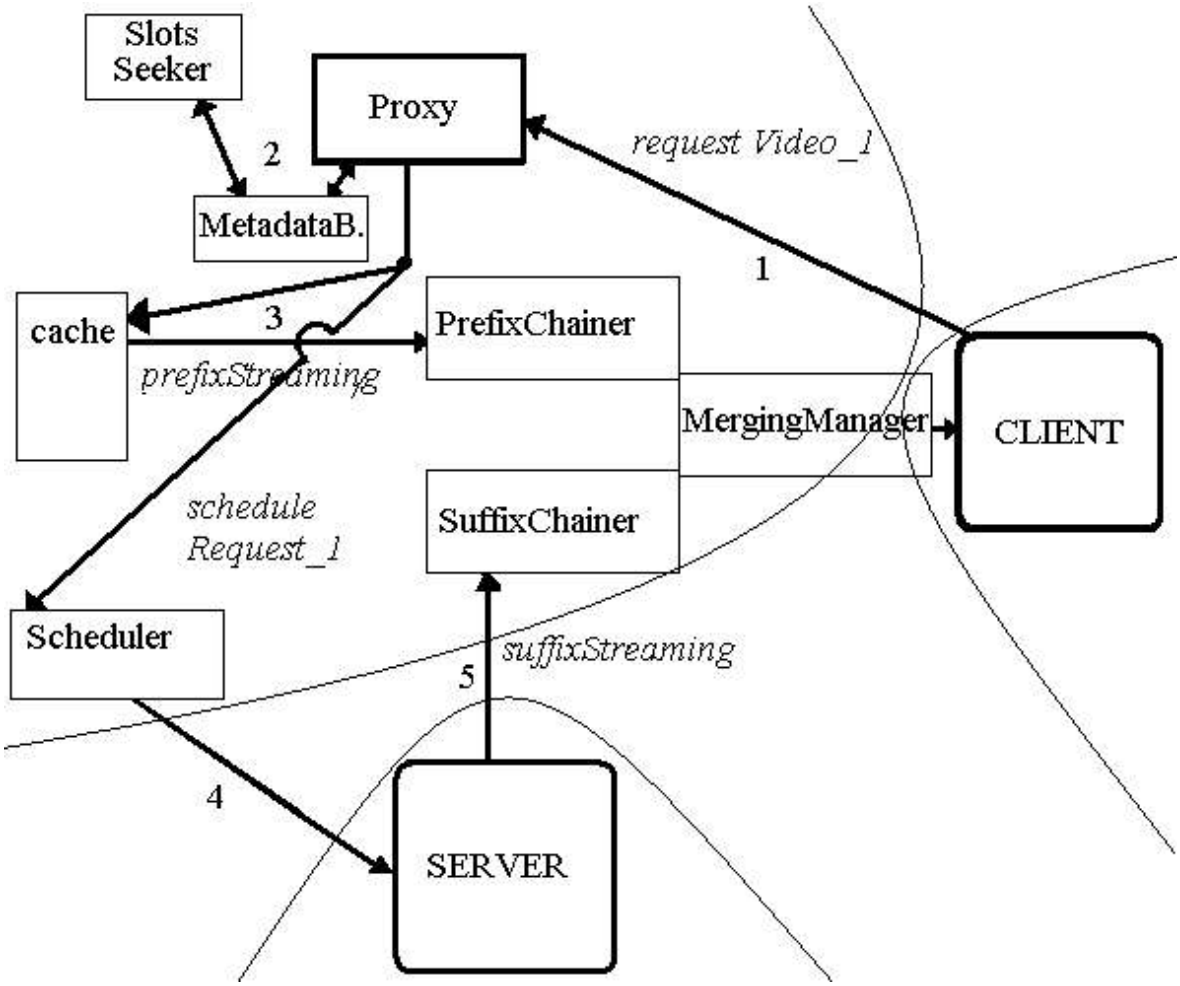

Fig6.3 – Schema di integrazione di tutti i componenti che realizzano l'SBatch

Terminato lo streaming, quindi il merging, il client avrà nella sua cache un video completo come aveva richiesto. Il sistema è quindi pronto per un'altra richiesta.

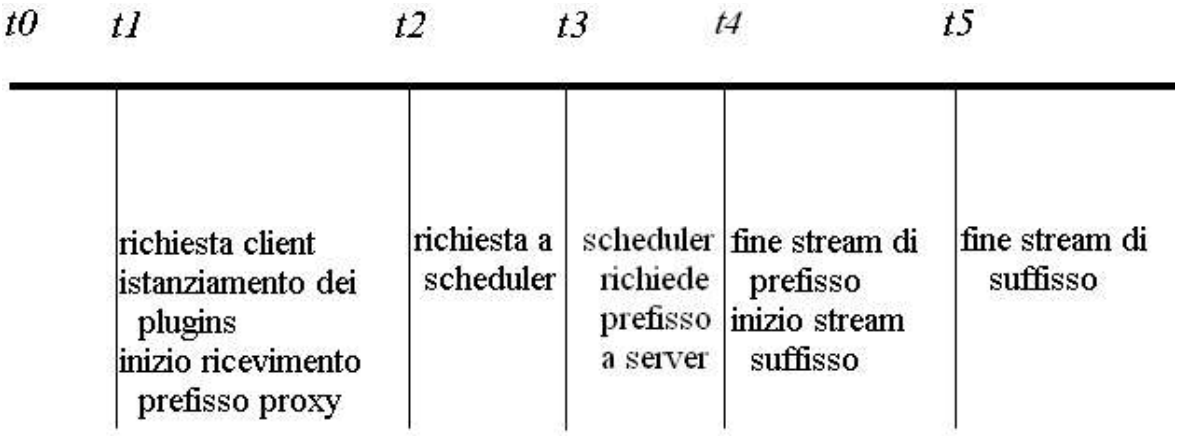

Fig6.4 – Schema dei tempi di richiesta e streaming nell'SBatch

Nel caso multi-client il funzionamento risulta poco diverso; ogni richiesta produce sempre una ricerca di prefisso, quindi una richiesta di scheduling che comanderà lo streaming del suffisso da server adeguatamente secondo le politiche che sono state implementate nei precedenti capitoli.

6.4 Risultati Sperimentali

In questo paragrafo vedremo i risultati ottenuti testando alcuni componenti dell'SBatch integrati all'interno della piattaforma MUMOC. Per il testing sono stati utilizzati 2 client, 1 proxy e 1 server come mostrato in **Fig6.5**.

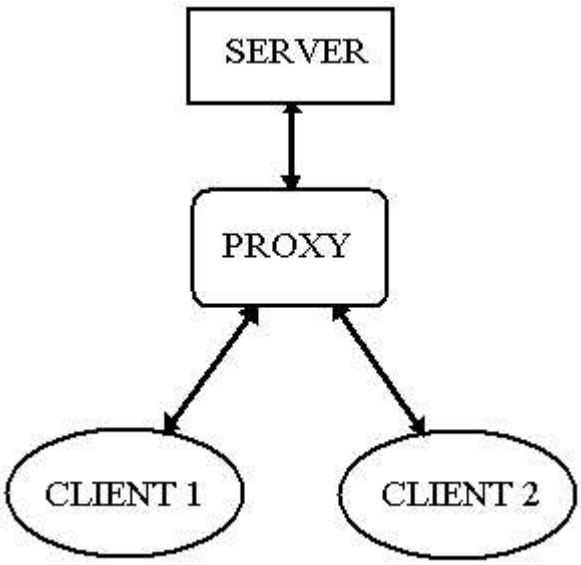

Fig6.5 – Topologia utilizzata per il testing dei componenti dell'SBatch

Le macchine utilizzate sono illustrate nella seguente tabella:

NODO	*SPECIFICHE HARDWARE*
SEVER	ASUS 1.2GHz – 1024 Mb RAM
PROXY	ASUS 1.2GHz – 1024 Mb RAM
CLIENT (x 2)	AMD Duron 900MHz – 256 Mb RAM

Tab6.1 – Componenti Hardware delle macchine utilizzate per il testing

In oltre su ogni macchina erano presenti i seguenti supporti software :

SOFTWARE	*VERSIONE*
WINDOWS	XP Professional
Java Virtual Machine (JVM)	1.4.2-b28
Java Media Framework (JMF)	2.1.1e

Tab6.2 – Componenti Software installati sulle macchine utilizzate per il testing

Infine tutti i nodi della rete risultavano connessi attraverso una LAN Ethernet da 100Mbps.

Grazie alla configurazione appena illustrata mostriamo i risultati ottenuti testando i tempi e le latenze di alcuni componenti di SBatch integrati nella piattaforma MUMOC.

6.4.1 – Testing dei componenti per la gestione della cache in slots

L'inizializzazione dei componenti per la gestione della cache del proxy in slots vengono inizializzati assieme alla *Protocol Unit* del proxy stesso come abbiamo spiegato nei capitoli precedenti. Abbiamo inizializzato la cache con una divisione in 200 slots da 100 frame ciascuno e abbiamo misurato i tempi richiesti per inserire, sostituire e cancellare sia slots che file.

La tabella seguente mostra i risultati ottenuti:

OPERAZIONE	*TEMPO RICHIESTO (msec)*
Inizializzazione dei componenti per la gestione della cache del proxy in 200 slots da 100 frame ciascuno	40
Inserimento di un file che occupi 1 slot	600
Inserimento di un file che occupi 10 slot	811
Inserimento di un file che occupi 100 slot	3054
Inserimento di un file che sostituisca 1 slot	810
Inserimento di un file che sostituisca 10 slot	1101
Inserimento di un file che sostituisca 100 slot	3326

Tab6.3 – Risultati ottenuti dopo il testing sui componenti per la gestione della cache in slots

Vediamo chiaramente che le latenze aumentano all'aumentare del numero di operazioni fatte sul documento XML mentre la sostituzione (quindi il taglio dei frame del file corrispondenti a quelli cancellati sul documento XML) risulta un'ulteriore latenza di circa 200 msec dovuta all'accesso al file video e al taglio dei frame corretti. Il processo di taglio risulta comunque contribuire ad un ritardo di 200 msec sia per un taglio di 1 slot che per il taglio di 100 slot. Questo è chiaramente compresnibile in quanto le operazioni sul metadato risultano la somma di diversi metodi per la ricerca, la scrittura e la riscittura di diverse parti del documento mentre il taglio consiste nella semplice eliminazione di una coda del file video (come abbiamo spiegato nel capitolo 5).

6.4.2 – Testing dello *Slots Seeker*

Abbiamo poi operato alcune ricerche di un dato file su proxy attraverso *Slots Seeker*; la tabella seguente mostra i risultati ottenuti.

Da essi percepiamo che la ricerca risulta essere rapida sia nel caso di video che occupano un numero bassissimo di slot che nel caso occupino un numero di slot molto più consistente; questo appunto perchè l'operazione consiste nella scansione del file intero per ottenere il numero di slot che un dato video occupa.

OPERAZIONE	*TEMPO RICHIESTO (msec)*
Inizializzazione	70
Ricerca di un file che occupi 1 slot	19
Ricerca di un file che occupi 10 slot	20
Ricerca di un file che occupi 100 slot	22

Tab6.4 - Risultati ottenuti dopo il testing sullo *Slots Seeker*

6.4.3 – Testing dello Scheduler

Anche lo scheduler viene inizializzato assieme alla *Protocol Unit* del proxy; abbiamo quindi operato una serie di richieste sperimentando le casistiche viste nei capitoli precedenti misurando quanto ritardo viene aggiunto dalle operazioni dello scheduler.

OPERAZIONE	*RITARDO AGGIUNTIVO (msec)*
wait() di una richiesta	10
wait() di una richiesta contemporanea soddisfatta dopo quella attualmente considerata	17
wait() di una richiesta contemporanea soddisfatta prima di quella attualmente considerata	27

Tab6.5 – Risultati ottenuti dopo il testing dello *Scheduler*

I ritardi sopra considerato sono calcolati nel caso di due richieste contemporanee; nel caso di più richieste contemporanee è necessario aggiungere 10 secondi di ritardo circa per ogni operazione di *wait()* e *notify()*.

6.5 Considerazioni Finali

Per quanto riguarda sincronizzazioni e ritardi abbiamo quindi capito che gli unici problemi possono esserci solamente nel

intervallo di tempo trasmesso allo scheduler per il soddisfacimento della richiesta. Tra questi ritardi si considerano soprattutto quelli dovuti alla trasmissione dei messaggi, allo streaming da server e ai piccoli ritardi causati da `wait()` e `notify()` nel considerare molte operazioni di scheduling. Attraverso un monitoraggio del sistema è possibile stabilire queste latenze medie (come abbiamo fatto poco fa) e quindi decrementare il suddetto intervallo di scheduling in modo corretto; è comunque consigliabile mantenerlo anche un poco più basso del previsto in modo da essere sicuri che al momento di fine streaming del prefisso, il suffisso sia già pronto per essere salvato.

Proseguendo notiamo che la ricerca del metadato sui diversi nodi dell'albero del sistema risulta efficiente in quanto sfrutta il motore del *Metadata Browser* e anche lo streaming non subisce modifiche da quanto introdotto nel capitolo 3 in quanto l'SBatch continua ad usufruire dei servizi di streaming dei DataSource già implementati in MUMOC.

Il client riceve il flusso normalmente da un solo DataSource come se stesse svolgendo un download da una sola fonte e soprattutto accede al servizio in modo leggero anche nel caso di tante richieste contemporanee per le quali si otterrebbe un eccessivo carico del server senza una politica come quella implementata.

In generale notiamo quindi che il server sarà più leggero mentre il peso delle gestioni e delle trasmissioni è meglio gestito su tutto il percorso; otteniamo quindi il risultato voluto, cioè la migliore gestione di un servizio multi-client.

- CONCLUSIONE -

Nel corso di questa tesi abbiamo studiato come sia possibile risolvere alcuni problemi attinenti ai servizi di VoD attraverso politiche progettate per la gestione di Proxy Caching, in particolare con lo scopo di rendere più rapido l'accesso alle risorse condivise.

Siamo poi passati ad analizzare la piattaforma MUMOC, sviluppata per lo streaming di servizi multimediali attraverso proxy, e per essa abbiamo realizzato uno specifico algoritmo di batching: SBatch. Esso in particolare consente il download di video attraverso lo streaming di una parte (detta prefisso) da proxy e di un'altra (detta suffisso) da server.

Il risultato ottenuto è quello di un sistema che alleggerisce il server portando ad una migliore distribuzione del carico su proxy, quindi permettendo una migliore gestione delle richieste che si presentano in finestre temporali limitate.

Abbiamo perciò capito che in un ambiente distribuito i compiti devono essere gestiti e spartiti al meglio tra i diversi nodi dell'albero in modo da poter ottenere accessi alle risorse sempre più rapidi, quindi migliorando la qualità del servizio per i client che ne usufruiscono.

In futuro potrebbe essere possibile migliorare la politica implementata eliminando gli effetti causati delle latenze mostrate nel capitolo precedente: ad esempio decrementando dinamicamente tutti i tempi di attesa comunicati allo scheduler ogni volta che il Thread viene avviato o fermato, oppure velocizzando le operazioni svolte sul metadato XML che mantiene la divisione logica della cache in slots. Infine è possibile partire dai risultati già ottenuti per implementare una politica più efficiente in un ambiente diverso come spiegato nel capitolo 2, allargandoci verso più complessi scenari multicast.

Ringraziamenti

Si ringraziano tutti gli studenti e i professori della facoltà di Ingegneria di Bologna che hanno lavorato ai progetti MUMOC, MUM e SOMA in questi anni.

Si ringrazia inoltre Alessandro Falchi per i plugins realizzati per il merging di video utilizzati per la progettazione della relativa parte della politica.

Un ringraziamento particolare all'Ing. Luca Foschini, correlatore della tesi, per la continua disponibilità ed attenzione.

Infine un ringraziamento particolare alla mia famiglia per avermi sostenuto durante questi mesi.

Dedico questo lavoro alla memoria di mia mamma Maria.

Bibliografia e riferimenti bibliografici

[1] Adler M., Sen S., Towsley D., Wang B., 2004, *Optimal Proxy Cache Allocation for Efficent Streaming Media Distribution*, [IEEE 2004]

[2] Barlow M., Ostuni J., Terrazas A., *Java Media APIs : Cross-Platform Imaging, Media and Visualization*, Sams Publishing

[3] Bellavista P., Corradi A., Foschini L., 2004, *MUMOC : an Infrastructure for Open Video Caching*

[4] Chung K., Hong H., Lim E., Park S., 2001, *A Proxy Caching Scheme for Continuous Media Streams on the Interner*, [IEEE 2001]

[5] Cornell G., Horstmann S., 2001, *Java 2 : I Fondamenti*, Mc Graw Hill

[6] Fahmi H., Ghafoor A., Hsu L., Latif M., Liu P., Sedigh-Ali S., 2001, *Proxy Servers for Scalable Interactive Video Support*, [IEEE 2001]

[7] Fellow W., Yu P., Wu K. 2004, *Segmentation of Multimedia Streams for Proxy Caching*, [IEEE 2004]

[8] Rejaie R. 1999, *Multimedia Proxy Caching*, Doctoral Thesis of Southern California

[9] Tanenbaum A., 1997, *Computer Networks*, UTET Libreria Srl