Università degli studi di Bologna

FACOLTA' DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica
Reti di Calcolatori L-A

INFRASTRUTTURA PER IL BILANCIAMENTO NELL'USO DELLE RISORSE PER L'EROGAZIONE DI SERVIZI MULTIMEDIALI

Tesi di laurea di:
Marco Passerini

Relatore:
Chiar.mo Prof. Ing. Antonio Corradi

Correlatore:
Ing. Luca Foschini

ANNO ACCADEMICO 2003-2004

Indicazione delle parole chiave:
Stream
Broadcast
Proxy
Servizio
Load Balancing
Multimediale
Agenti Mobili
Video
Cpu

INDICE

INTRODUZIONE

Il grande sviluppo tecnologico avvenuto negli ultimi anni nella produzione di calcolatori e nello sviluppo di infrastrutture di rete informatiche ha permesso la nascita di nuove soluzioni software in grado di fornire servizi un tempo impensabili. Il campo multimediale è stato uno di quelli che hanno ricevuto una spinta maggiore da questo sviluppo: è sempre più alla portata di tutti infatti la possibilità di collegarsi ad Internet e poter ascoltare trasmissioni radiofoniche, vedere presentazioni video, parlare al telefono o partecipare a video-conferenze, tutto questo attraverso la rete.

Il motivo per cui queste soluzioni non sono state realizzate già in passato è legato al fatto che esse richiedono un ingente consumo di risorse computazionali, di memoria, e di banda, caratteristiche presenti in maniera non sufficiente sui calcolatori presenti nelle case qualche anno fa.

La logica architetturale di questi servizi è legata a quella dei normali servizi di rete. E' infatti possibile ricondursi al modello cliente-servitore, differenziando gli utenti finali, le risorse e i passaggi intermedi necessari a portare quest'ultima ai richiedenti del servizio.

Questo è il contesto in cui si colloca questa tesi: con essa ci si pone l'intento di realizzare un servizio di broadcast video adottando un meccanismo di bilanciamento del carico sui nodi di tipo proxy. Si intende fornire un servizio multi-piattaforma, in grado di adattarsi alle caratteristiche delle architetture di rete sui cui viene eseguito. Sarà necessario prevedere una funzione di gestione dei flussi e del loro corretto recapito ai singoli utenti, seguendo le loro richieste per quanto riguarda la qualità di servizio. Per la gestione del bilanciamento dei proxy bisognerà analizzare il loro comportamento qualora si trovino nel caso di dover gestire un elevato numero di richieste e studiare la possibilità di attribuire parte del loro carico ad altri proxy meno sollecitati. Tutto questo dovrà avvenire nel modo più trasparente possibile dal lato dell'utente.

La tesi sarà organizzata come segue. Nel capitolo 1 vengono presentate alcune nozioni generali riguardo al contesto in cui si inserisce il progetto, dando alcune definizioni e presentando i modelli utilizzati per lo sviluppo di soluzioni multimediali. Nel capitolo 2 si entra più nello specifico mostrando le soluzioni software già esistenti che sono servite come punto di partenza per lo sviluppo: la piattaforma per lo sviluppo di soluzioni ad agenti "Secure and Open Mobile Agent" (SOMA), il middleware di supporto alla creazione di servizi multimediali "Mobile agent based Ubiquitous multimedia Middleware" (MUM), e il "Java Media Framework" (JMF), vale a dire il supporto offerto dalla Sun all'uso di Java per lo sviluppo multimediale. Nel capitolo 3 viene effettuata l'analisi di progetto, inquadrando il problema e analizzando le scelte implementative. Nel capitolo 4 viene spiegata la fase di progettazione del sistema, mentre nel capitolo 5 vengono mostrati alcuni dettagli implementativi. Il capitolo 6 mostra i risultati dei test effettuati sul sistema e le considerazioni che da essi sono nate.

CAPITOLO 1

CONTESTO DI PROGETTO

1.1 STREAMING MULTIMEDIALE

Col passare degli anni l'incremento dei fruitori della rete Internet ha portato ad un incredibile aumento degli investimenti volti al suo miglioramento, creando nuove strutture e servizi un tempo impensabili. Inizialmente il suo utilizzo era legato strettamente ai contenuti ipertestuali, ma oggi vediamo come stiano proliferando i servizi di tipo multimediale, grazie all'allargamento della banda trasmissiva e al potenziamento dei calcolatori: web radio che trasmettono programmi in diretta, videoconferenze, servizi di telefonia realizzata con tecnologia Voice over IP.

1.1.1 Risorse

Introducendo il concetto di streaming multimediale è necessario prima di tutto fornire la spiegazione di alcuni termini utilizzati.
Un oggetto multimediale è un particolare dato che varia con continuità e soggetto a vincoli di tempo. Potrebbe essere una risorsa audio, video o un generico flusso di dati.

I flussi sono l'identità minima identificabile per la trasmissione di oggetti multimediali, possono essere di tipo unicast (da uno ad uno) o multicast (da uno a molti).
Con fruizione di uno stream multimediale si intende il recapito e rendering dell'oggetto multimediale stesso

In questo contesto sono state sviluppate svariate applicazioni, le cui architetture sono solitamente riconducibili a un insieme di quattro componenti: la risorsa (che può essere un filmato, un file audio, le immagini riprese da una telecamera), un calcolatore che agisce da server, la rete di trasmissione dei dati, e uno o più client che ricevono e rielaborano i dati ricevuti rendendoli accessibili agli utilizzatori finali.

Per quanto riguarda le risorse, generalmente si tratta di risorse di tipo video o audio.
Una risorsa di tipo video è caratterizzata dai seguenti parametri:

- Risoluzione orizzontale: espressa in pixel, corrisponde al numero di punti con cui viene discretizzata, sull'asse delle ascisse, un'immagine. Maggiore è la risoluzione più risulta definita l'immagine
- Risoluzione verticale: come sopra, ma rispetto all'asse delle ordinate.
- Frame per secondo (fps): rappresenta il numero di immagini visualizzate ogni unità di tempo.
- Formato dell'immagine: sono presenti numerosi formati, legati al tipo di rappresentazione che viene data utilizzando il codice binario.

Una risorsa di tipo audio è espressa da una successione tempo discreta codificata in numeri binari corrispondente a un segnale tempo continuo.

- Frequenza di campionamento: è il corrispondente dei frame per secondo nel campo video ed è espressa in Hz. Per ottenere la rappresentazione di tutte le frequenze del segnale originario è necessario che la frequenza di campionamento sia almeno pari al doppio della massima frequenza presente nel segnale. Se la risorsa è la voce umana si potrà utilizzare una frequenza di campionamento relativamente bassa, per una buona qualità con la musica è invece necessaria una frequenza molto più alta, in genere 44.100Hz
- Risoluzione: numero di bit per campione, solitamente 16 bit per una buona qualità, 8 per una qualità più scadente. Essa è legata al "Signal to Quantization Noise Ratio" (SQNR), cioè il rapporto fra il segnale e il rumore.
- Numero di canali: solitamente si utilizza un canale per segnali monofonici, due per quelli stereofonici, ma possono essere anche di più, come ad esempio avviene nel caso del Dolby surround.

La necessità di far coesistere due flussi di tipo di verso sullo stesso stream fa nascere il concetto di interleaving. Detto anche Time Division Multiplexing (TDM), consiste nella suddivisione dei flussi audio e video in pacchetti di piccole dimensioni, compressi e inseriti nello stream in maniera alternata.

Le risorse vengono processate e compresse, operazioni che richiedono un impegno notevole di CPU e memoria nei trasmettitori e ricevitori, ma i vantaggi legati alla diminuzione della banda occupata dalla trasmissione inseguito alla compressione sono molto più importanti.

1.1.2 Trasmissione

Solitamente in caso di streaming multimediale uno a molti, come può essere una trasmissione radiofonica sulla rete, il servitore è già attivo quando arriva la richiesta di partecipazione da parte di cliente, il quale si collega tramite socket internet ad una sua porta lasciata in ascolto e richiede l'avvio della partecipazione al flusso. A questo punto possono avvenire controlli sulle caratteristiche del cliente, la negoziazione della qualità di servizio richiesta e solo successivamente incomincerà il flusso dei dati.

Il fatto che un oggetto multimediale sia inteso come dato che varia in modo tempo continuo, è un fattore legato alla percezione dell'utente. Il concetto della percezione è importante, poiché i contenuti provenienti dall'elaborazione dei calcolatori in realtà sono dati che variano in modo tempo discreto. Se le variazioni avvengono in modo molto veloce l'essere umano però non se ne rende conto.

I flussi multimediali sono detti isocroni poiché se un dato arriva in tempo utile per il suo utilizzo, allora sarà consumato, altrimenti sarà scartato. Ciò avviene poiché è preferibile avere una momentanea riduzione di qualità dei dati ricevuti rispetto a un loro ritardo. Questo è importante, poiché implica l'utilizzo di protocolli con caratteristiche differenti rispetto a quelli normalmente utilizzati per le trasmissioni di dati, per i quali è necessaria una perfetta coerenza fra dati inviati e dati ricevuti, a discapito della tempistica di arrivo.

Come noto le trasmissioni sulla rete richiedono una certa "larghezza di banda". Essa rappresenta la portata del canale di trasmissione, vale a dire il numero di dati che possono transitarvi per unità di tempo. Quando la banda occupata diventa eccessiva per

le possibilità del mezzo trasmissivo, l'invio del flusso multimediale diventa problematico, mettendo a rischio le caratteristiche previste per la corretta fruizione dello stesso.
I parametri che incidono sulle trasmissioni sono i seguenti:

- La latenza è il ritardo temporale con cui un dato discretizzato arriva dalla sorgente al destinatario. In caso di necessità di interazione fra i nodi è importante che sia mantenuta a livelli bassi.
- Il jitter è indicato come varianza della latenza. E' una variabile aleatoria, e per compensarla è necessario creare buffer di memoria sul ricevitore in modo da registrare una certa quantità di dati prima di fornirla all'utente in modo regolare.
- Il loss rate è la percentuale di dati che possono andare persi nel flusso.
- Lo skew caratterizza le trasmissioni di presentazioni multimediali, costituite da più flussi che vengono trasmessi contemporaneamente. Identifica il loro lo sfasamento di arrivo al ricevitore.

1.1.3 Protocolli

Internet è caratterizzato dal supporto per il protocollo di rete IP, sopra al quale agiscono i protocolli di trasporto come TCP e UDP.
IP supporta tre modi di utilizzo, vale a dire unicast, broadcast, e multicast:

- Unicast, conosciuto anche come punto a punto, descrive la trasmissione dei pacchetti da una sorgente a un singolo indirizzo; questo è l'utilizzo di IP oggi più frequente.
- Broadcast è un modo di utilizzo più sofisticato, che prevede la trasmissione di pacchetti a più host di una subnet. Questo permette un maggior risparmio di banda (non essendo presenti trasmissioni duplicate) ma il fatto di essere limitato ad una subnet rende il suo utilizzo più ristretto.
- Multicast è il metodo di indirizzamento più complesso e versatile: il trasmettitore invia i dati ad un singolo indirizzo,

quello corrispondente alla sessione di multicast, e l'infrastruttura di rete è responsabile di smistare i dati a tutti i ricevitori. I ricevitori devono identificarsi come tali registrandosi presso di essa, richiedendo la particolare sessione desiderata. L'Internet Assignment Numbers Authority (IANA) ha assegnato un preciso range di IP alle trasmissioni multicast, quelli che vanno da 224.0.0.0 a 239.255.255.255. Ulteriori divisioni in questo intervallo sono state specificate per utilizzi particolari , quali le video conferenze.

TCP è un protocollo con connessione, che offre garanzie spesso non strettamente necessarie per le trasmissioni multimediali: in questi casi vengono più spesso preferite soluzioni basate su UDP, un protocollo più vicino al sottostante IP.

UDP, a differenza di TCP, non garantisce l'arrivo a destinazione dei pacchetti, né che arrivino ordinati. E' un protocollo a basso overhead.

I sistemi di streaming multimediali sono spesso di tipo "best-effort": attribuiscono cioè maggior importanza alla sincronia dei dati rispetto alla loro correttezza. Questa scelta è quella che generalmente motiva la decisione di utilizzare UDP per questo tipo di trasmissioni, in quanto non rallenta né blocca la trasmissione in caso di errori sui dati ma continua a trasmetterli lasciando a livelli superiori eventuali controlli sull'integrità.

1.2 PROBLEMI NELLO DI SVILUPPO DI APPLICAZIONI MULTIMEDIALI

Il grande sviluppo di questo tipo di servizi ha però messo in luce quelli che sono i limiti e i problemi più comuni che si riscontrano sviluppando architetture per applicazioni multimediali.

Il primo grande limite si è dimostrato essere la carenza fisica di risorse. Dati di tipo audio o video impiegano un carico sui calcolatori molto maggiore rispetto a semplici contenuti testuali, andiamo ora a elencare le aree più critiche:

- CPU: per i contenuti multimediali questa è la risorsa più contesa, poiché la codifica, la compressione dei dati, l'impacchettamento, lo spacchettamento, la decompressione, la decodifica e il rendering sono operazioni che impegnano fortemente la CPU.
- Banda trasmissiva: l'apertura di molte connessioni contemporaneamente provoca una suddivisione della banda disponibile e, poiché si parla di trasmissioni multimediali, ogni trasmissione è molto onerosa, in quanto trema video e audio richiedono un flusso sostenuto di dati. Sono per questo utilizzate compressioni dei dati in trasmissione e decompressioni in ricezione. Queste operazioni impegnano la CPU e, in caso di codifiche con perdita (lossy), portano alla trasmissione di presentazioni di qualità ridotta rispetto a quella originale.
- Risorse di memoria: esse sono importanti poiché nei sistemi multimediali si è soliti effettuare l'operazione buffering, cioè la memorizzazione delle parti di flusso più recenti in memoria. Questo è utile per evitare le variazioni temporali dei flussi e renderli alla vista dell'utente più regolari.

Per questi motivi nasce l'esigenza di progettare sistemi di divisione dei compiti, inserendo intermediari che si fanno carico di gestire parti del lavoro che altrimenti sarebbero svolte da un unico calcolatore. Sarà questo l'argomento trattato da questa tesi.

Un altro problema è quello legato alla carenza di supporto da parte delle piattaforme esistenti. Non esistono infatti metodologie standard per quanto riguarda la gestione della qualità di servizio, né a livello di protocolli (TCP ad esempio), né a livello di sistemi operativi. Poiché la modifica dei sistemi operativi per offrire questo supporti è ancora lontana, è quindi necessario sviluppare opportuni applicativi che si fanno carico di monitorare le condizioni di corretta fruizione dei servizi offerti.

Il sistema discussi in questa tesi sarà sviluppato come supporto di tipo middleware. I middleware consistono in specifici software che agiscono da intermediari fra differenti componenti di applicazioni. Sono usati soprattutto per supportare applicazioni distribuite complesse. Forniscono inoltre astrazioni e servizi di alto

livello per le applicazioni, per facilitare la programmazione, l'integrazione fra applicativi e la gestione dei sistemi.

La soluzione presentata è inoltre più adattabile a contesti multi-piattaforma, e si può adattare facilmente alle API offerte dai livelli sottostanti, ma comporta costi maggiori in termini di overhead.

L'eterogeneità delle piattaforme condiziona lo sviluppo delle architetture costringendo gli sviluppatori a tenere conto delle varie casistiche di utilizzo dei servizi. Possiamo ad esempio avere utenti che ricevono strema video su moderni calcolatori oppure su dispositivi palmari o addirittura su telefoni cellulari. Queste condizioni portano a creare strutture il più possibili modulari, utilizzando entità facilmente rimpiazzabili, con funzioni di interfaccia specifiche per le relazioni i terminali, che non influenzano l'architettura del sistema e si legano ad esso tutte allo stesso modo assumendo il ruolo di "adattatori". Queste entità vengono solitamente identificate con i server proxy.

Un proxy è un entità di rete che svolge il compito sia di client che di server. Esso permette al client di avere una connessione indiretta con altri servizi di rete. Solitamente i client si connettono ad un proxy e richiedono una risorsa; a questo punto il proxy decide se richiedere questa risorsa ad un altro server o se fornire quella che potrebbe contenere all'interno della propria cache.

1.3 QUALITA' DI SERVIZIO

Le problematiche di frequente scarsità di banda, l'alto numero di fruitori o altri fattori più complessi come ad esempio la mobilità dei terminali hanno fatto nascere la necessità di un controllo sulle caratteristiche qualitative dei flussi di informazione inviati attraverso le reti. A questo si è aggiunta l'eterogeneità dei supporti utilizzati dagli utenti, che possono andare ad esempio da potenti calcolatori a semplici telefoni cellulari o palmari.

Si è perciò deciso di coniare il termine Qualità di servizio, per indicare i requisiti che i sistemi multimediali progettati devono

garantire per consentire una fruizione migliore possibile dei contenuti da parte di chi inoltra le richieste dei servizi.

I servizi forniti da questi sistemi possono essere molteplici, dalla telefonia, alle video conferenze, allo streaming di programmi video e audio, e i requisiti legati ai tipi di media trattati possono essere differenti.

Un'attenta analisi va dedicata al contenuto delle trasmissioni: testo, musica, voce, filmati, in real-time o meno. L'essere umano presenta diversi livelli di tolleranza quando si tratta di recepire queste informazioni, il caso più classico è quello delle trasmissioni telefoniche. In questo tipo di trasmissioni è fondamentale la comprensibilità di ciò che si riceve ma non la sua qualità; è considerata fastidiosa infatti una trasmissione audio che arriva agli utenti ritardata di qualche secondo o magari a scatti, mentre è più accettabile udire una banda di frequenze più ristretta che si limiti a solo quella della voce.

I parametri della Qualità di servizio possono dividersi in soggettivi ed oggettivi.

- I parametri oggettivi, sono quelli che si possono rilevare dal lato di chi riceve il servizo, come ad esempio ritardo di trasmissione e banda trasmissiva.
- I parametri soggettivi invece sono quelli che dipendono dal contesto in cui viene fruita la presentazione. Potremmo infatti avere la ricezione su un mezzo non adeguato, ad esempio uno schermo piccolo o degli altoparlanti scadenti, oppure fattori legati alla soggettività dell'utente.

I parametri di tipo oggettivo possono ridurre la qualità di servizio in maniera differente a seconda della loro natura. In caso di streaming di filmati video o di programmi radiofonici, latenze dell'ordine di grandezza di qualche secondo non sono recepite come un grave disturbo, sicuramente sono preferibili a jitter elevati che compromettono la fluidità della visione/audizione. In caso di servizi ad alta interazione invece, come possono essere telefonate via internet o video conferenze, una latenza maggiore a un valore di circa mezzo secondo risulta fastidiosa, poiché porta a dovere attendere un certo tempo prima di avere una risposta ad una

domanda posta, e potrebbero esserci problemi di sincronia sui discorsi. Nelle video conferenze potrebbe invece essere meno fastidiosa la presenza di sfasamento fra immagine e audio, in quanto il parlato sarebbe comunque intelligibile.

Quando un programmatore progetta sistemi multimediali dovrebbe sempre tenere conto della disponibilità di risorse fisiche in relazione al tipo di servizio da affrontare, studiando soluzioni opportune per cercare di ottenere trasmissioni di qualità il più soddisfacente possibile.

Normalmente i sistemi prevedono la negoziazione della qualità di servizio prima dell'inizio della trasmissione, in modo da permettere al servitore di scegliere quali devono essere le caratteristiche del servizio da offrire, quali privilegiare e quali considerare meno importanti. Generalmente l'accordo avviene in questo modo: alla richiesta della presentazione il cliente indica oltre al suo identificativo le scelte riguardo i parametri relativi alle sue preferenze (ad esempio la risoluzione, o il framerate che si desiderano mantenere in un filmato); il sistema allora si configura e prenota la risorse necessarie, viene effettuata la negoziazione della qualità di servizio e, terminata questa fase, può iniziare l'invio dei dati richiesti. Nei sistemi più complessi le negoziazioni possono essere rieffettuate in un secondo tempo, poiché le condizioni di utilizzo possono variare, ad esempio in caso di congestioni di rete o di mobilità dei terminali. Le regolazioni possono essere fatte avvenire dinamicamente o, nel caso in cui le condizioni siano talmente critiche da non permettere più un servizio soddisfacente, la trasmissione può essere interrotta.

1.4 BROADCAST

Questa tesi si inserisce in un contesto particolare dello streaming multimediale, vale a dire quello del broadcast. Con sistema di broadcast si intende un sistema che prevede l'invio di flussi da un server a uno o più client. La sua particolarità è legata al fatto che questi flussi contengono tutti le stesse informazioni considerando lo

stesso istante. Si può quindi dire che i flussi siano tutti la duplicazione di un primo flusso.

Una tipica trasmissione di tipo broadcast via rete è quella fornita dalle web-radio, con le quali un utente fornito di un semplice computer può trasmettere dati audio a un vasto numero di utenti in qualsiasi regione del mondo, similmente a come avviene con le radio via etere. Come nelle radio ci si mette infatti in ascolto di un flusso di dati che è già attivo, gli utenti non hanno possibilità di comandarlo (a differenza delle possibilità offerte ad esempio da un registratore) e possono riceverlo con una qualità più o meno buona (si pensi ad esempio alla scarsa qualità del segnale in montagna).

Una situazione analoga si presenta per i flussi di questo tipo attraverso la rete. Tutto infatti ricevono gli stessi dati, nessuno può comandare la trasmissione e può variare la qualità con cui la si riceve.

Trattandosi solitamente di applicativi destinati alla rete Internet e non specificatamente alle reti locali, non è possibile l'utilizzo di tecnologie di multicast, e quindi è necessario stabilire trasmissioni di tipo unicast dirette con tutti gli utenti del servizio. Queste trasmissioni possono essere effettutate da un server oppure, con architetture più complesse, possono essere gestite da dei proxy, i quali si prendono la responsabilità di ricevere un solo flusso dal server e di duplicarlo ed inviarlo a tutti i clienti. Questo viene fatto poiché, come accennato in precedenza, la gestione dei flussi è onerosa dal punto di vista computazionale, e distribuirla su più calcolatori permette servizi più efficienti e stabili.

Sono ancora possibili controlli sulla qualità di servizio per stabilire le necessità di ogni utente e fornirgli un flusso adeguato alle caratteristiche richieste.

1.5 CONCLUSIONI

In questo primo capitolo sono stati introdotti i concetti generali riguardanti le caratteristiche dei servizi di streaming attraverso la

rete, presentando prima la spiegazione dei termini legati alle entità più comuni in questo tipo di applicazioni e in seguito analizzando le principali problematiche che sorgono quando si analizzano questi sistemi. Si è poi introdotto il concetto di Qualità di Servizio e si è presentato quello di broadcast attraverso le reti di calcolatori.

CAPITOLO 2

TECNOLOGIE UTILIZZATE

Per lo sviluppo del sistema proposto da questa tesi si è deciso di utilizzare delle tecnologie già esistenti per fornire un valido supporto alla programmazione. Si è quindi deciso di basare lo sviluppo su un modello ad agenti. Per questo motivo si è scelta la piattaforma “Secure and Open Mobile Agent” (SOMA), la quale fornisce un ambiente di lavoro completo per la creazione di agenti mobili su topologie che sono astrazioni di reti di calcolatori. Un altro supporto viene dato dal middleware “Mobile agent based Ubiquitous multimedia Middleware” (MUM), che offre un framework per lo sviluppo di applicazioni multimediali fornendo entità server, proxy, client per la gestione dei contenuti multimediali, della qualità di servizio, funzioni di mobilità e di caching locale. Il linguaggio di programmazione scelto è Java, per motivi di compatibilità con i due sistemi appena citati e per la sua caratteristica di essere un linguaggio ad oggetti multi piattaforma. Il solo Java però non è sufficiente alla creazione di sistemi multimediali, per cui si è deciso di includere il Java Media Framework; esso fornisce il supporto ottimizzato a applicazioni multimediali, fornendo delle API complete ed efficienti.

2.1 SOMA

SOMA (acronimo di Secure and Open Mobile Agent) è un ambiente ad agenti mobili sviluppato presso il Dipartimento di Elettronica, Informatica e Sistemistica presso la facoltà di Ingegneria, appartenente all’Università di Bologna.

SOMA si inserisce nel contesto dei modelli basati sulla mobilità del codice, dei quali ora descriviamo le principali caratteristiche.

2.1.1 Modelli sulla mobilità del codice

I modelli basati sulla mobilità del codice sono modelli applicati alla programmazione per permettere lo spostamento su nodi di una rete di parti di codice in grado di essere eseguiti, come se fossero dei

normali thread. Questo metodologia di lavoro permette il risparmio di elevati quantitativi di banda trasmissiva, poiché il codice può essere eseguito direttamente dove si trova la risorsa da gestire, e in questo modo essa non deve venire trasmessa attraverso la rete.

Questo modello è applicato dove siano necessari lo spostamento di processi per operare bilanciamenti di carico, gli aggiornamenti software di apparati che devono rimanere sempre in funzione, l'esecuzione di task di controllo e monitor su reti remote.

Queste operazioni richiedono una serie di circostanze esecutive particolari e spesso non realizzate nei sistemi oggi comuni. Prima di tutti è la necessità di dotare il software di un ambiente di esecuzione uniforme. Questo è raramente possibile, nel caso analizzato da questa tesi ciò è possibile poiché si è scelto di utilizzare un linguaggio multi piattaforma come Java. Un altro limite rilevante è il problema della sicurezza. E' infatti pericoloso attribuire agli agenti la possibilità di agire senza regole e di accedere alle risorse senza un opportuno controllo. E' quindi opportuno che l'ambiente di esecuzione sia precisamente calibrato per prevedere ogni possibile attacco a informazioni e risorse sensibili, e deve poter effettuare un monitoraggio sugli agenti in esecuzione.

Questo modello purtroppo non è ancora stato precisamente definito e non esistono regole riguardo la sua implementazione; ogni programmatore che desidera farne uso quindi deve studiare soluzioni originali cercando di renderle il più possibile robuste e sicure per evitare gli inconvenienti precedentemente spiegati.

La mobilità del codice impiegata nel modello a Remote Evaluation prevede che, se su un nodo è presente il codice di elaborazione per una risorsa ma non è presente quella risorsa, allora il codice di elaborazione viene spostato sul nodo in cui essa è presente. Su quel nodo viene allora effettuata la computazione, senza aver trasmesso una grande mole di dati sulla rete, ma solo quella relativa al codice di esecuzione, poiché in è genere essa è molto minore rispetto ai dati da trattare.

Il modello a Code on Demand è quello che viene utilizzato solitamente per il download degli aggiornamenti del codice. Infatti, esso prevede che se un entità non possiede la capacità per gestire le risorse disponibili, può richiedere un'altra entità il codice necessario per la computazione e quindi eseguirlo in locale.

Il paradigma ad agenti mobili si è affermato negli ultimi tempi per la sua flessibilità e per le sue caratteristiche di mobilità, particolarmente adeguate alla programmazione di sistemi paralleli e distribuiti.

Gli agenti mobili sono componenti software, simili ai thread, che sono in grado di migrare da un computer all'altro autonomamente, e continuare la propria esecuzione sul calcolatore di destinazione. Questo avviene in contrasto con i paradigmi di Remote Evaluation e di Code On Demand, poiché gli agenti mobili sono particolari forme di codice realmente attive che possono scegliere di spostare la propria esecuzione su un secondo computer in qualsiasi momento durante la loro esecuzione.

Nei paradigmi di mobilità del codice esistono due tipi di mobilità:

- Mobilità forte, in cui lo stato degli agenti viene salvato. Vengono perciò spostati sia il codice della "Execution Unit" che il loro stato; arrivato a destinazione l'agente inviato sarà eseguito a partire dal punto in cui era stato fermato.
- Mobilità debole, in cui lo stato non è salvato e l'agente instanziato sul calcolatore di destinazione riparte dall'inizio del suo codice. Si è però soliti in questi casi creare degli stati "manualmente", a livello applicativo.

Nel paradigma ad agenti mobili ci si trova nel caso di mobilità forte, in cui le risorse si muovono assieme alla Execution Unit e allo stato di esecuzione.

Di fondamentale importanza sono il problema dell'identificazione e della comunicazione fra agenti. Quest'ultima può essere a scambio di messaggi oppure a memoria condivisa, caso particolarmente interessante per la comunicazione fra molti.

I sistemi ad agenti mobili non sono stati molto utilizzati per problemi legati alla sicurezza. Esiste infatti il pericolo che agenti creati appositamente possano essere eseguiti maliziosamente all'interno delle reti studiare ed eseguire codice pericoloso, che potrebbe bloccare il funzionamento del sistema, appropriarsi delle risorse o provocare danni. Per questo motivo i sistemi a mobilità del codice devono garantire che gli stati degli agenti non siano osservabili quando attraversano la rete, arrivino integri a destinazione, che tutti i codici siano autenticati prima dell'esecuzione, e devono essere previsti controlli degli accessi per le risorse sensibili.

1.5.4 Caratteristiche di SOMA

SOMA è stato realizzato utilizzando il linguaggio Java, che risulta particolarmente adatto allo sviluppo di sistemi di questo genere. Le sue caratteristiche più interessanti per questo campo sono:

- Portabilità: essendo un linguaggio interpretato può essere eseguito su calcolatori di qualsiasi architettura, a patto che abbiano installata una Java Virtual Machine (JVM). Essa è in grado di trasformare il bytecode delle classi in istruzioni macchina specifiche a seconda del tipo di calcolatore su cui è eseguita.
- Linguaggio Object-oriented: l'astrazione fornita dai linguaggi ad oggetti permette lo sviluppo di software modulari, grazie alla possibilità di estendere le classi e sfruttare l'ereditarietà. E' quindi più semplice rispetto ai linguaggi non ad oggetti la scrittura di agenti a partire dalle classi messe a disposizione dal sistema.
- Caricamento dinamico: fornisce funzioni di caricamento e bind dinamico delle classi da sorgenti remote.
- Sicurezza: favorisce lo sviluppo di applicazioni sicure grazie a soluzioni built-in quali domini di protezione, controlli di accesso e permessi da associare al codice remoto e a quello locale.

- Serializzazione: offre la possibilità di salvare gli oggetti, codificarli in stream di byte e trasferirli attraverso la rete.

La scelta di un linguaggio come questo però implica una serie di limitazioni alle possibilità di sviluppo. Il limite più evidente risulta essere il fatto che una parte dello stato di esecuzione rimane gestito dall'interprete, ed è impossibile reperirlo se non apportando modifiche all'interprete stesso. L'utilizzo di un interprete è però necessario per permettere la portabilità dell'applicazione, fatto di grande importanza nello sviluppo di sistemi ad agenti mobili.
Queste condizioni rendono SOMA della classe dei sistemi a mobilità debole.

SOMA fornisce un'astrazione di località per l'esecuzione degli agenti che si ispira a sistemi di rete complessi come internet. Si tratta di una struttura ad albero gerarchica, che prevede per ogni località caratteristiche specifiche, dal punto di vista dell'amministrazione della sicurezza e della gestione.

- E' importante spiegare il significato di Domini e Place in questo contesto. Un Place corrisponde al contesto di esecuzione di un agente, e corrisponde solitamente al concetto di nodo. Esso può trovarsi su una macchina specifica, da solo o con altri Place. Al suo interno gli agenti possono interagire con le risorse locali e cooperare con altri agenti presenti.
- Un Dominio è costituito da un gruppo di Place accomunati da qualche caratteristica (di tipo fisico o logico). Può ad esempio essere associato al concetto di LAN.

I Place di uno stesso dominio sono strettamente legati l'un l'altro, si conoscono fra di loro e possono comunicare senza intermediari.
E' definito un place principale, detto Default Place, che svolge il ruolo di Place "primario" in una località.
La struttura gerarchica permette l'identificazione dei nodi (Place e Default Place) indicandoli con il percorso fra la radice e il nodo stesso.

L'ambiente di esecuzione degli agenti è coincidente con i Place, in esso sono presenti i moduli di supporto che forniscono i servizi utili per la loro funzione. Questi moduli permettono la comunicazione fra i Place, mantenendo i canali di comunicazione

necessari, gestiscono l'esecuzione, l'ingresso e l'uscita dalle località, e mantengono le informazioni sui Place appartenenti a un dominio.

I Place e i Domini sono identificati grazie a un sistema di naming che attribuisce ad ognuno di esse un PlaceID, vale a dire la concatenazione del nome del dominio e il nome del Place (nullo per i default Place).

Ogni agente è identificato grazie all'AgentID. Esso è formato da una coppia di valori, vale a dire l'identificatore del Place su cui nasce l'agente e un numero intero progressivo. Questo è possibile poiché sussiste l'ipotesi di base che i Place abbiano nomi diversi fra di loro.

La comunicazione fra agenti in SOMA è del tipo a scambio di messaggi, soluzione che risulta essere molto flessibile.

E' stato studiato inoltre un meccanismo di sicurezza che non permette agli agenti di accedere in modo incontrollato alle risorse e ai servizi messi a disposizione. Inoltre è possibile definire politiche di sicurezza sia a livello di dominio che a livello di Place.

2.2 MUM

Il progetto di questa tesi si svolgerà principalmente sul middleware MUM, sviluppato al Dipartimento di Elettronica, Informatica e Sistemistica della facoltà di Ingegneria presso l'Università di Bologna. Il nome è l'acronimo di Multimedia agent based Ubiquitous Multimedia middleware, e gia da esso è lecito intuire la natura di questo progetto: si tratta di un sistema ad agenti mobili per la fruizione di presentazioni multimediali di varia natura e con diversa qualità, su rete fissa oppure mobile. MUM prevede l'esistenza su alcuni nodi di risorse multimediali, gli utilizzatori possono richiedere il delivery di questi contenuti, comandare la presentazione e chiedere lo spostamento della sessione su un altro nodo. Le trasmissioni avvengono solo in maniera unicast.

Le risorse multimediali possono essere di vario tipo, audio, video o presentazioni formate da vari media. Viene gestita

automaticamente la prenotazione di questi dati ed essi possono essere inviati nel formato più adatto a seconda del esigenze legate al tipo di dispositivo utilizzato dagli utenti. I profili degli utenti contengono infatti indicazioni sul tipo di piattaforma da loro utilizzata.

Gli utenti dispongono di un interfaccia grafica per il comando delle richieste simile a quella di un videoregistratore, e se usufruiscono del servizio utilizzando dispositivi mobili, che ad esempio si collegano al sistema grazie alla rete wi-fi, la loro sessione viene automaticamente spostata al nodo più vicino alla loro posizione.

Appoggiandosi sul sistema SOMA, MUM ha un efficiente sistema di download e inizializzazione dinamica dei componenti software.

2.2.1 Fruizione del materiale multimediale

MUM, essendo un sistema basato su SOMA, sfrutta pienamente le possibilità da esso offerte di creare agenti mobili in grado di comunicare fra loro, collocati su topologie ad albero. Al paradigma ad agenti si somma il modello Client Server , infatti sono presenti le seguenti entità fondamentali: Client, Server, Proxy, ClientAgent e ProxyAgent. Le prime tre sono fisse, mentre le ultime due sono veri e propri agenti mobili. Fra il cliente e il servitore si viene a creare un percorso, chiamato Service path, sul quale possono venire instanziati dei proxy. Questa catena che unisce il cliente al servitore viene utilizzata per l'instanziamento delle entità lungo il percorso e permettere quindi l'inizio delle comunicazioni. Questo percorso è riconfigurabile, e può variare in caso di mobilità di terminale o nel caso analizzato da questa tesi, in cui un proxy attribuisca la responsabilità di gestire i propri flussi a dei proxy secondari.

- Client: il client è un'entità fissa, incaricata di ricevere i flussi multimediali richiesti dall'utente. Il codice di questa entità, sebbene si tratti di un entità fissa, può essere scaricato dinamicamente, seguendo il paradigma di Code on Demand.

- Server: l'entità server è anch'essa fissa, e anche il suo codice può essere caricato dinamicamente seguendo il paradigma di Code on Demand. Il server carica la presentazione multimediale richiesta e inizia il flusso lungo il service path. Esso, una volta attivato, rimane attivo in modo da poter servire anche le prossime richieste di dati sul nodo su cui si trova.
- Proxy: il proxy segue la struttura delle entità appena descritte, svolge in un certo senso funzioni sia di cliente che di servitore. Esso riceve lo stream dal server (o da un altro proxy) e lo ridirige al prossimo nodo, cioè un proxy o un client, gestisce le eventuali disconnessioni che potrebbero avvenire e salva in locale il flusso ricevuto, facendo quindi del caching dei dati. Incapsula la logica di trasformazione dei dati multimediali, per adattarli a clienti che potrebbero essere collegati con dispositivi mobili dotati di scarsa capacità computazionale e svolge il compito di monitorare costantemente se la Qualità di Servizio è sufficiente: in caso di problemi può avviare la riconfigurazione della sessione.
- Client Agent: questa entità si propone come interfaccia fra l'utilizzatore e il sistema. Inizializza la sessione e riceve i comandi dell'utente attraverso un'apposita interfaccia grafica. E' stato studiato come agente mobile per favorire la gestione della mobilità.
- Proxy Agent: è l'entità che assume maggior importanza nello sviluppo di questa tesi. E' stata progettata come agente mobile per sfruttare la mobilità dei terminali, e si lega all'entità proxy per la gestione dei flussi.

2.2.2 Un modello a tre livelli

Il sistema MUM è composto da tre livelli costituenti:

- Middleware Mechanism Layer: in cui vengono implementati i servizi basilari dell'ambiente. Qui vengono gestiti i contenuti multimediali basandosi su un'architettura di tipo client-server. Ogni nodo richiede al nodo padre il riferimento al database

contenente le informazioni riguardo alle presentazioni multimediali, vale a dire il server, e mette a disposizione localmente uno stub per la sua interrogazione.

- Middleware Facilities Layer: che utilizza i servizi di base forniti dal Middleware Mechanism Layer e fornisce servizi più avanzati agli sviluppatori.
- Application Layer: contiene le entità costituenti del sistema e le interazioni con esso.

2.2.3 Sistema di streaming video

In seguito alla richiesta di inizio dello streaming da parte del cliente, vengono automaticamente instanziati i proxy sui nodi del percorso fra esso ed il servitore. Ognuna di queste entità è dotata di un gestore ed un protocollo che si occupano della sessione della sessione, uno o più componenti chiamati Streaming Agent per gestire i flussi, e un manager per eseguire controlli sulla Qualità di Servizio. Gli Streaming Agent gestiscono specificatamente i flussi video, mentre le Protocol Unit si occupano di incapsulare i protocolli utilizzati. Ovviamente il client gestirà solo i flussi in ingresso, il server solo quelli in uscita, e i proxy entrambe.

I flussi sono divisi in due livelli, uno riguardante i comandi, e uno che incapsula gli stream. Gli stream partono dal server, passano attraverso il proxy e raggiungono il client. I comandi seguono invece il senso contrario, partendo dal client e arrivando al server.

Descriviamo ora in modo riassuntivo quello che avviene nell'attuale implementazione di MUM:
Il primo client quando richiede la presentazione fa scattare un meccanismo di inizializzazione che prevede la localizzazione della risorsa, la creazione di un plan speciale contenente i componenti necessari per tutte le entità da creare e il percorso per arrivare alla risorsa. Questo plan viene passato a un agente chiamato *PlanVisitorAgent*, che si reca nodo per nodo attraverso tutto il percorso presente nel plan ad instanziare i componenti necessari su ognuno di essi. Terminata l'inizializzazione viene inviato

all’indietro, dopo dopo nodo, le *ComponentInfo* (vale a dire indirizzo IP e porta in ascolto) di ogni nodo successivo. In questo modo, supponendo di trovarci nella situazione della topologia creata per lo svolgimento di questa tesi, il client riceve l’indirizzo del proxy principale, e il proxy riceve quello del server. Terminata questa fase inizia lo stream del flusso di dati richiesto.
L’utente può a questo punto comandare la presentazione per mezzo di un interfaccia grafica con funzioni simili a quelle di un semplice videoregistratore.

Nella seguente immagine si mostra come la nuova connessione da parte di un utente, in MUM comporti la creazione di un nuovo flusso lungo il percorso.

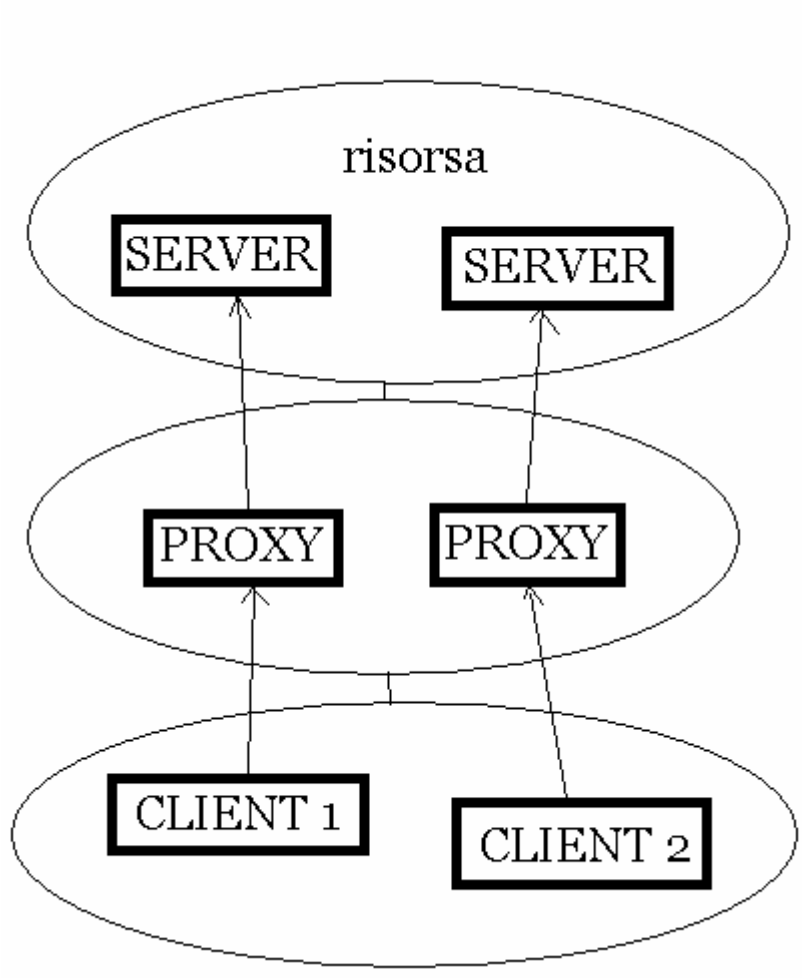

Fig. 2.1 – Rappresentazione della inizializzazione in MUM

Il *PlanVisitorAgent* segue, come si può capire dal nome, il pattern visitor, il cui utilizzo è largamente consigliato dai testi di ingegneria del software. La forza di questo pattern sta nel fatto che

permette di dividere gli algoritmi dalla struttura degli oggetti. Esso consiste nel creare una struttura di classi che possiedono un metodo che accetta come argomento l'oggetto visitor. Visitor è un interfaccia che ha un metodo visit per ognuna di queste classi, questi metodi sono richiamati dal metodo accept delle classi. Possono essere allora scritte varie implementazioni dell'interfaccia Visitor, e queste classi svolgeranno le particolari operazioni da noi desiderate.

L'entità proxy è stata realizzata come una via di mezzo fra l'entità server e l'entità client. Questa entità riceve dall'entità a valle i comandi, e li inoltra all'entità posizionata a monte, lungo lo topologia. Riceve invece dall'entità a monte i flussi e li inoltra all'entità posizionata a valle.

Nella prossima figura viene mostrata la struttura UML di questa entità.

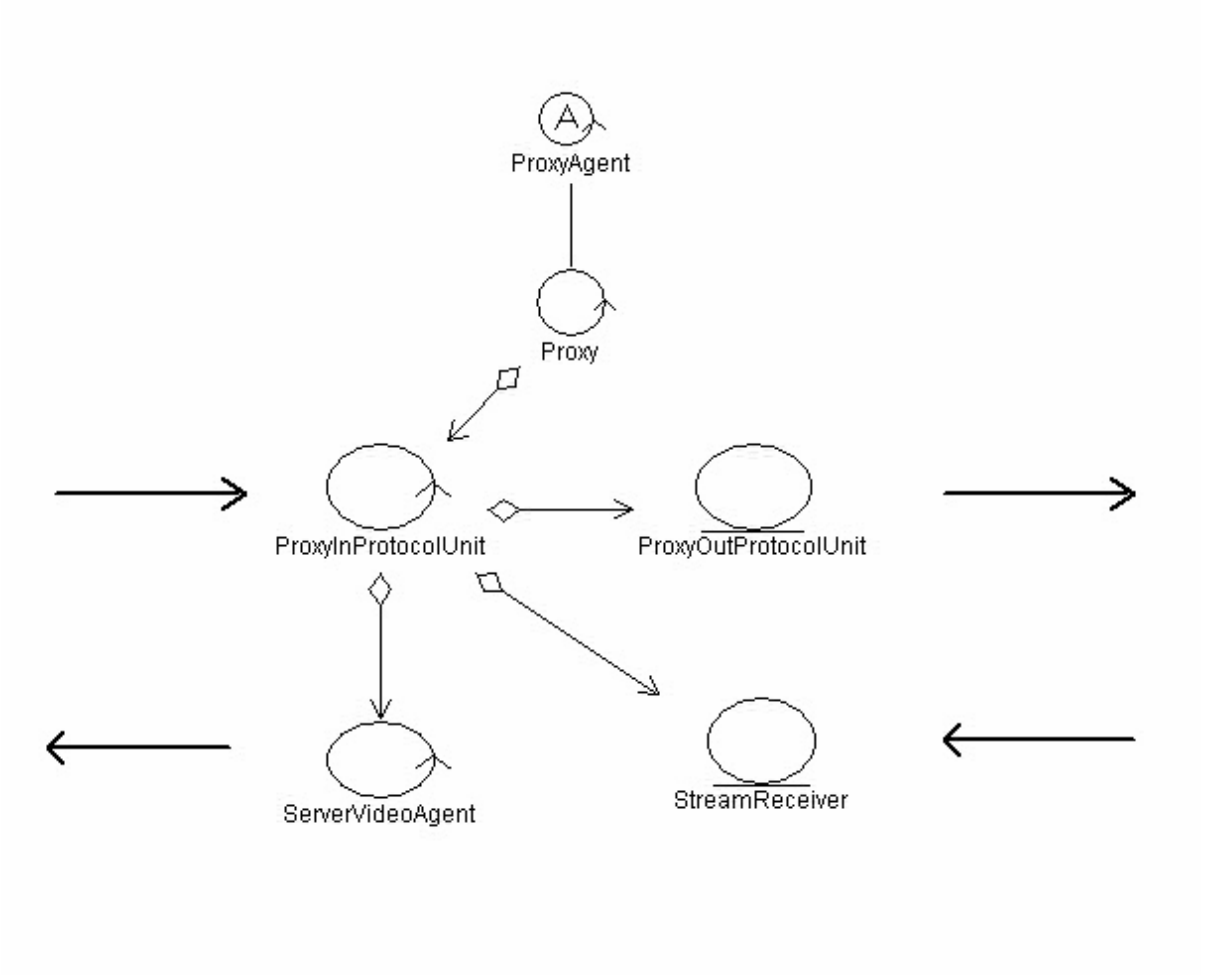

Fig. 2.2 – Rappresentazione delle entità costituenti del proxy in MUM

Si può subito evidenziare come siano state realizzate due classi in grado di incapsulare i flussi in ingresso e quelli in uscita. L'agente *ProxyAgent* è quello che coordina tutta l'entità, contiene il riferimento a un *Proxy* che gestisce la sessione attuale; possono essere evidenziati tue livelli di comunicazione, uno legato allo scambio di comandi e uno per la gestione degli streaming.

- Livello dei comandi: le classi impiegate per questo livello sono *ProxyInProtocolUnit* e *ProxyOutProtocolUnit*; *ProxyInProtocolUnit* svolge anche la funzione di coordinare tutte le altre classi di gestione di flussi. *ProxyInProtocolUnit* è la classe che riceve i domandi mentre *ProxyOutProtocolUnit* è quella che li invia al server.
- Livello di flusso: il flusso è gestito dalle classi *StreamReceiver* e *ServerVideoAgent*. *StreamReceiver* riceve il flusso multimediale dal server e *ServerVideoAgent* permette al proxy di comportarsi come un server nei confronti del client, inviandogli i dati ricevuti in ingresso.

L'attuale implementazione prevede il supporto al caching locale dei dati sui proxy, alla gestione della qualità di servizio e alla mobilità dei terminali. E' in fatti possibile ricevere lo stream video su un terminale mobile e spostarsi appoggiandosi a differenti nodi di rete mantenendo sempre la sessione corrente.

2.3 JMF

Fondamentalmente, Java Media Framework (JMF) è un'estensione di Java per gestire l'audio e il video. Più rigorosamente, le JMF API (Java Media Framework Application Programming Interface) sono una delle API opzionali di Java che estendono le funzionalità del core Java Platform. Esistono altre API opzionali, liberamente disponibili presso il sito della Sun, quali Java 3D e Java Advanced Imaging (JAI).

JMF, come si può intuire dal nome, è una collezione di classi che permette l'elaborazione di oggetti media. La documentazione di JMF 2.1.1 fornita ai programmatori dalla Sun presenta JMF dicendo che fornisce un'architettura unificata e un protocollo di messaggi per la gestione dell'acquisizione, dell'elaborazione e del recapito di dati multimediali che evolvono temporalmente. JMF permette di aggiungere audio video e altri media timebased alle applicazioni e alle applet. Esso permette di registrare, eseguire, trasmettere come flusso e convertire molti formati; essendo legato a Java dà la possibilità agli sviluppatori di creare applicazioni scalabili e multi piattaforma.

I formati supportati sono molteplici, ad esempio sono previsti i supporti per WAV, AIFF, MIDI, MPEG, QuickTime, AU, GSM, RMF.

Le principali caratteristiche delle API sono:

- Basandosi su Java sono multipiattaforma, in grado cioè di essere eseguite su qualsiasi calcolatore che supporti la Java Virtual Machine.
- Gestione integrata e uniforme di oggetti Audio e Video.
- Supporto per un grande numero di formati e codec audio e video.
- Permette l'esecuzione di questi media.
- Permette il salvataggio su file dei media.
- Si possono registrare media da microfoni e videocamere.
- Si possono ricevere e inviare flussi attraverso le reti.
- Possibilità di eseguire multiplexing e demultiplexing di dati.
- Possibilità di conversione fra formati.
- E' possibile effettuare modifiche sui media, come ad esempio applicare effetti.
- Si possono aggiungere nuovi plugin e aggiungere supporti a nuovi formati.
- Perfetta integrazione con le API di Java.

Le operazioni su dati multimediali prevedono un grosso utilizzo di risorse fisiche, memoria per il loro salvataggio e velocità computazionale per la loro esecuzione. Sono infatti necessarie varie

fasi di elaborazione prima della loro esecuzione, quali la decompressione delle immagini e il rendering, per non parlare di altre operazioni molto onerose come potrebbero essere l'applicazione di effetti in real time.

Il grande vantaggio della portabilità di Java è legato al fatto che le classi sono eseguite su una Virtual Machine che interpreta durante la fase di esecuzione il byte-code generato dal compilatore. Questo meccanismo però penalizza fortemente la velocità di esecuzione che, anche utilizzando ottimizzazioni, rimane troppo bassa per poter utilizzare applicazioni multimediali con buone prestazioni su dispositivi di capacità computazionale limitata. Un modo per ovviare a questo problema è scrivere codice di basso livello, ricorrendo al codice nativo della piattaforma che sta utilizzando. Questo modo di lavorare però elimina la possibilità di scrivere codice portabile e comunque prevede che il programmatore abbia un livello elevato di conoscenza dell'architettura del proprio calcolatore.

JMF è stato studiato per permettere ai programmatori di scrivere codice di alto livello, senza doversi curare di questi problemi. Chiunque abbia una buona conoscenza del linguaggio Java può quindi cimentarsi con questo framework, avendo a disposizione una collezione di classi completa per lo sviluppo di applicazioni multimediali e ottenere comunque risultati soddisfacenti in termini di prestazioni senza curarsi di dover scrivere istruzioni di basso livello. Esistono implementazioni del JMF sia pure java, quindi multipiattaforma, sia di tipo nativo, specifiche per certi tipi di sistemi operativi (ad esempio Windows, Linux, Solaris).

2.3.1 Il componente *Player*

Il *Player* è il componente centrale delle API fornite da JMF per la riproduzione dei media. Può essere paragonato ad un videoregistratore, dotato perciò di funzioni di controllo del tutto simili alle sue e della possibilità di essere collegato ad altre entità.

Utilizzarlo non obbliga il programmatore ad operare a livello di codice nativo, e a effettuare controlli sul rilascio e la prenotazione delle risorse. E' quindi totale la trasparenza fra il codice java e le routine di sistema.

Nella seguente figura viene mostrato come le risorse multimediali vengano identificate con l'elemento *DataSource*, mentre le connessioni con le periferiche audio e video si trovano dall'altra parte del *Player*.

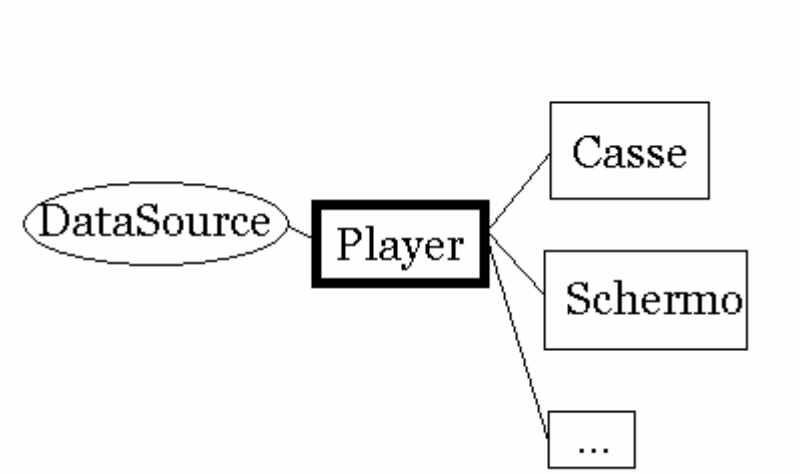

Fig. 2.3 – Rappresentazione del componente player

Per il *Player* sono previsti sei stati di esecuzione, cinque dei quali indicano una situazione di non riproduzione. La transizione fra essi è dovuta un qualche particolare evento.
Gli stati sono:

- *Unrealized*
- *Realizing*
- *Realized*
- *Prefetching*
- *Prefetched*
- *Started*

2.3.2 Il componente *Processor*

Esiste un'entità, denominata *Processor*, che estende il *player* offrendo metodi per la gestione e la trasformazione dei flussi multimediali. Esso è utilizzato per l'invio e la ricezione dei flussi multimediali, ed è utilizzato in questa tesi per la possibilità offerta di ottenere un *DataSource*. Questa funzionalità è essenziale poiché il *DataSource* in uscita dal P*rocessor* può essere passato in ingresso all'*RTPManager*, e quindi rediretto sotto forma di flusso verso la rete.
E' quindi comprensibile che il *Processor* sia stato utilizzato nei componenti Server e Proxy di MUM, proprio per le sue funzionalità di gestione degli stream.

Nella seguente figura mostriamo come avvenga la lettura dei dati da disco rigido o in diretta da ad esempio un microfono, il processing dei dati e la successiva possibilità di salvare in locale lo strema o inviarlo via rete:

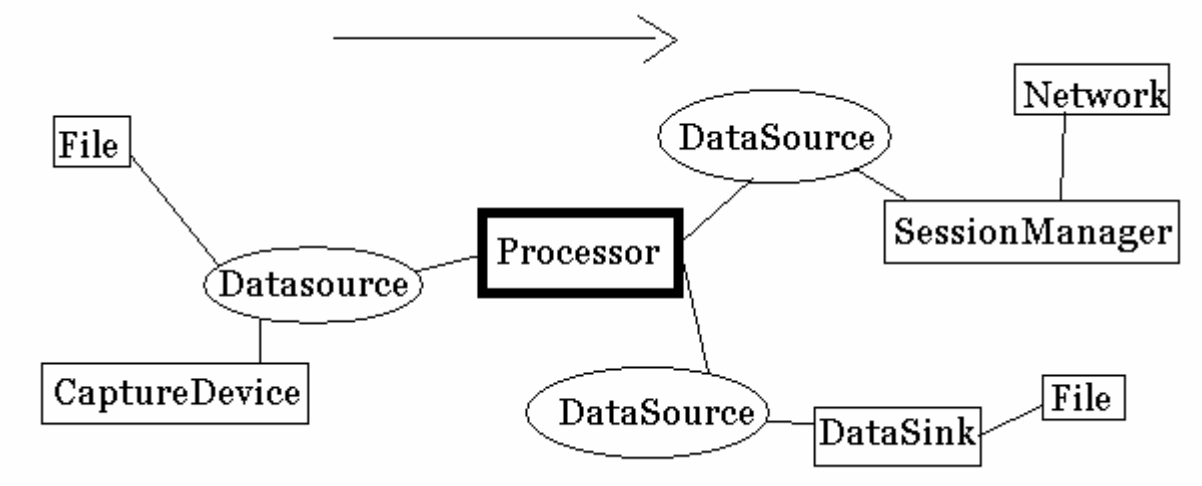

Fig. 2.4 – Rappresentazione del componente processor

2.3.3 Il componente *DataSource*

Come si può capire, l'oggetto *DataSource* assume un ruolo rilevante in questa tesi, poiché rappresenta l'astrazione della risorsa fisica, il dato cioè che deve essere gestito dalle entità realizzate. Al suo interno sono contenute le informazioni riguardo alla posizione

dell'oggetto multimediale a cui si riferisce, riguardo ai protocolli e al software interessato.

Nel nostro progetto di tesi, come vedremo, è stato impiegato un particolare DataSource nella sua versione che viene detta "clonable". Questa versione permette la creazione di cloni del *DataSource* originario, che possono essere controllati direttamente dal *DataSource* utilizzato per crearli.

2.3.4 Il componente *DataSink*

DataSink è il componente che viene utilizzato per redirigere il flusso multimediale da un *DataSource* verso un'altra direzione. Solitamente viene utilizzato per salvare su file uno stream o per inviare il flusso attraverso la rete. Esso offre la possibilità di iniziare l'invio dei dati, interromperlo o reagire in caso di errori o comunque eventi degni di essere gestiti.

2.4 PROTOCOLLO RTP

Come già è stato accennato, nelle applicazioni che garantiscono servizi in tempo reale e soprattutto in quelle a carattere multimediale una perdita dei dati ricevuti non è considerata grave quanto un ritardo troppo elevato nella ricezione di essi. Questo avviene in contrasto con le applicazioni di tipo tradizionale, che presuppongono una trasmissione e ricezioni di dati obbligatoriamente integra, a discapito del fattore temporale.
Per questo motivo sono stati studiati protocolli specifici per questo tipo di applicazioni, come ad esempio quello che ci si accinge ad esporre: il protocollo RTP.

RTP è l'acronimo di Real-time Transport Protocol, ed è uno standard Internet per quello che riguarda il trasporto di dati Real-time, quali possono essere le trasmissioni di tipo audio o video. Esso è definito dall'Audio-Video Transport Working Group dell'Internet Engineering Task Force (IETF). Informazioni sull'IETF sono

reperibili al sito http://www.ietf.org; questa task force fa parte della più vasta Internet Society (ISOC), una società professionale che tratta tematiche quali l'evoluzione ed il futuro di Internet. L'IETF descrive il protocollo RTP con un Request for Comments, siglato RFC1889 (reperibile su http://www.ietf.org/rfc/rfc1889.txt). E' già dall'inizio del 1996 che esso è stato definito come standard e, poiché è stato largamente usato fino ai giorni nostri senza rilevare particolare errori, può essere certamente considerato stabile.

Dall' IETF viene descritto come segue: RTP provvede funzioni di trasporto end-to-end adatte per applicazioni che trasmettono dati in real-time, come audio, video, dati simulativi, su servizi di rete multicast o unicast. Non tratta la prenotazione delle risorse e non garantisce la qualità di servizio (QOS) per i servizi real-time. Questi aspetti vanno quindi trattati a livello applicativo.

E' stato definito un protocollo apposito per il trasporto delle informazioni di controllo, chiamato RTCP. Esso è stato creato per permettere il monitoraggio della consegna dei dati in maniera scalabile per grandi reti di multicast e provvedere controllo e funzioni di identificazione minimali.

Questi due protocolli sono stati progettati per essere indipendenti dai livelli inferiori di trasporto e rete.

Tipicamente le applicazioni eseguono RTP utilizzando come livello di trasporto il protocollo UDP, preferibilmente al TCP, poiché garantisce le caratteristiche di tipo best-effort richieste per il tipo di servizio da effettuare.

Si ricorda che UDP costituisce una semplice interfaccia fra il livello di rete e il livello applicativo; non fornisce garanzie sull'effettivo arrivo dei messaggi, né se essi verranno recapitati in maniera ordinata, ma permette il loro arrivo in una maniera temporalmente più rigorosa rispetto a quello che avviene con TCP. Oltre a ciò esso è anche un protocollo a basso overhead, grazie al limitato numero di funzioni offerte.

I servizi previsti da RTP includono l'identificazione del tipo di contenuto (vale a dire il tipo di formato della presentazione trasmessa) di un singolo pacchetto, la numerazione dei pacchetti, il loro timestamping, e l'abilità di sincronizzare stream provenienti da

diverse sorgenti. Mancano però servizi di livello più alto quali la negoziazione della connessione o garanzie della qualità di servizio. La numerazione dei pacchetti permette al ricevente l'identificazione dell'ordinamento corretto dei dati, e quindi un riordino in locale, ed inoltre permette di poter identificare un pacchetto preciso all'interno di un flusso, senza dover per forza decodificarlo.

Queste caratteristiche evidenziano il fatto che RTP è stato progettato per essere snello e occupare la minor banda possibile, per lasciare quindi spazio al contenuto della presentazione che si sta trasmettendo. Questo protocollo è comunemente impiegato nelle applicazioni di video conferenza, ma anche in quelle che prevedono la memorizzazione di flussi continui, nelle simulazioni interattive distribuite, nelle applicazioni di misurazione e di controllo.

I pacchetti RTP sono composti da un header e da un payload. L'header include il tipo di media, il numero di sequenza, il time stamp, la sorgente di sincronizzazione e l'indicazione riguardo a dove è stato originata la presentazione.
L'header può variare da una dimensione di 12 byte fino a quella di 72 byte, a seconda del numero di sorgenti del dato.
I pacchetti RTCP possono essere invece classificati in 5 tipi: Sender Report, Receiver's Report, Source Description, Bye e Application specific. Essi sono utilizzati per monitorare la qualità di servizio e fornire informazioni sui partecipanti di una sessione in atto. Non sono sufficienti però per la gestione complessa di gruppi né per un monitoraggio molto efficace o frequente.

2.5 CONCLUSIONI

In questo capitolo sono stati mostrate le principali caratteristiche delle soluzioni già esistenti che sono state utilizzate per lo sviluppo di questo progetto. SOMA, come valido supporto alla creazione di sistemi basati su paradigmi ad agenti; MUM come middleware in grado di fornire entità già ben implementate per lo sviluppo di sistemi multimediali, le soluzioni che offre dovranno essere modificate ed estese per lo sviluppo di un sistema di broadcast; JMF

come estensione di java in grado di fornire performance accettabili per applicazioni multimediali che richiedono un elevato consumo di risorse, ed interfacce per la gestione completa delle presentazioni e dei flussi. Per informazioni più dettagliate si consiglia di reperire le rispettive documentazioni.

CAPITOLO 3

ANALISI PROGETTUALE

In questo capitolo affronteremo le problematiche relative alla realizzazione di un meccanismo di load balancing dei proxy all'interno del sistema già esistente di MUM, opportunamente modificato per poterlo utilizzare come un servizio di broadcast via rete, simile a quello offerto dalle radio o dalle emittenti televisive.
Nei precedenti capitoli sono stati introdotti in termini generali le condizioni e le soluzioni esistenti allo stato dell'arte da utilizzare per implementare un servizio come quello che si vuole creare. Innanzitutto si è definito il contesto delle applicazioni multimediali, con particolare riferimento al caso dello streaming, poi si è entrati più nello specifico descrivendo il protocollo RTP e concetti importanti come la Qualità di Servizio e il paradigma degli agenti mobili. Dopo di che è stata data una descrizione non troppo dettagliata delle tecnologie utilizzate per lo sviluppo di questa tesi, vale a dire SOMA, MUM e le "Application Programming Interfaces" (API) di Java Media Framework. Nelle prossime sezioni sarà fatta un'analisi del problema affrontato, per meglio comprendere le sue implicazioni e per poter più facilmente risolvere le problematiche che nasceranno durante il lavoro.

3.1 INQUADRAMENTO DEL PROBLEMA

Il problema da affrontare può essere analizzato come se fosse composto da due fasi: in una prima fase si vuole realizzare un servizio di broadcast multimediale via rete, mentre nella seconda fase verrà implementato un meccanismo di bilanciamento del carico dei proxy attivi nella località in cui sono presenti. L'obbiettivo è quello di sviluppare i componenti necessari sfruttando le possibilità offerte dal middleware MUM e dall'ambiente di gestione di agenti mobili SOMA.

Abbiamo analizzato come MUM sia stato progettato per reperire da un nodo, per mezzo di un meccanismo di streaming monocanale, risorse multimediali presenti su un altro nodo. La richiesta viene effettuata da un utente attraverso un'apposita interfaccia grafica, e in seguito ad esso scatta una procedura di

inizializzazione del sistema che prevede l'instanziamento di un server sul nodo su cui è situata la risorsa e di una serie di proxy sul percorso che porta dall'utente al server. Questa struttura va modificata per ottenere un servizio di tipo broadcast, sarà perciò necessario presupporre un solo server e una sola presentazione disponibile, e molti utenti che ricevono lo stesso flusso di dati. Questa struttura implica la rimozione delle possibilità di comandare il flusso da parte degli utenti, le loro capacità saranno ridotte solo al potersi unire alla ricezione o abbandonarla. La seconda fase consisterà nell'agire sull'architettura dei componenti proxy già esistenti, dotandoli di un sistema di riconoscimento dello stato del proprio carico computazionale. Come spiegato in precedenza infatti le operazioni di gestione dei dati di tipo audio e video sono molto onerose e ai proxy è attribuito un ingente carico computazionale quando si tratta di gestire più connessioni contemporaneamente. Prima del raggiungimento del limite verrà quindi lanciato il processo di inizializzazione di un nuovo proxy, che si prenderà l'onere di gestire le prossime connessioni in arrivo degli utenti. Questo meccanismo di bilanciamento del carico potrà essere realizzato in modo ricorsivo, per cui i proxy instanziati saranno molteplici. Infine sarà necessario implementare un protocollo per il mantenimento dello stato dei vari proxy, per una gestione corretta del carico su ognuno. Tutto questo dovrà prevedere un sistema di gestione degli errori come ad esempio l'eventualità di una caduta delle entità senza preavviso.

Queste modifiche sono volte a migliorare le prestazioni del sistema, permettendo un servizio in grado di fornire un numero di connessioni praticamente illimitato (a patto di avere un gran numero di calcolatori su cui eseguire i proxy). I loro effetti saranno analizzati nella fase di test, che verrà presentata come ultimo capitolo di questa tesi.

3.2 ANALISI DEI REQUISITI

I requisiti del progetto prevedono l'ideazione di un sistema di broadcast multimediale, creato estendendo il middleware MUM e l'ambiente di gestione degli agenti SOMA. E' necessario modificare adeguatamente il sistema MUM poiché attualmente non è prevista una struttura di questo tipo.

I proxy presenti in questa soluzione dovranno prevedere un meccanismo di bilanciamento del carico computazionale, per evitare situazioni critiche in cui il servizio fornito al cliente non sia adeguato. Esso deve mantenersi infatti al livello migliore possibile.
Non dev'essere inoltre percepibile dall'esterno la modifica avvenuta all'interno del sistema, che dovrà quindi essere più trasparente possibile.

Il sistema progettato deve essere modulare e poter essere facilmente estendibile poiché saranno possibili future implementazioni per servizi più complessi.

3.3 PROBLEMATICHE AFFRONTATE

Durante lo sviluppo del sistema si sono presentate diverse problematiche riguardo alle scelte implementative da effettuare. Le possibilità di soluzione offerte sono state analizzate e le scelte sono state effettuate seguendo precise motivazioni. Nonostante ciò le soluzioni non sono univoche, si sarebbero potute effettuare altre scelte che avrebbero privilegiato altri aspetti del sistema. In seguito sottolineeremo i pro e i contro di ognuna di esse.

3.3.1 Prima inizializzazione del servizio di broadcast

La prima problematica affrontata è stata la scelta della modalità con cui inizializzare il sistema in seguito alla prima richiesta di attivazione.

Il sistema MUM prevede attualmente che in seguito alla richiesta di una risorsa, venga instanziato un server sul nodo su cui essa è presente e di seguito una catena di proxy sui nodi fino al cliente.

Un sistema di broadcast multimediale potrebbe prevedere due casi di implementazione:

- Un server è presente staticamente sul nodo contenente la risorsa e viene inizializzato manualmente, con un'apposita interfaccia per far partire il servizio utilizzabile dai gestori. Potrebbe essere presente anche un proxy anch'esso statico, su un nodo situato fra il nodo con il server e quello a cui si collegheranno i client. I client quando attivati conoscono staticamente la posizione di questi elementi e quindi si collegano direttamente al proxy esistente e iniziano la ricezione.
- Più similmente al sistema presente attualmente in MUM, quando un client richiede una certa presentazione viene lanciato un meccanismo automatico di inizializzazione, per cui viene lanciato un server sul nodo contentente la risorsa e i proxy lungo il percorso.

La prima soluzione è più statica, mentre la seconda è più dinamica e prevede la possibilità di non conoscere dove realmente è situata una presentazione nel sistema. In questa tesi è stata scelta la seconda via, sia per motivi di omogeneità con il sistema esistente, sia perché così è più facile fare il deploy del servizio qualora ce ne sia bisogno, sia perché questo modo di agire è ritenuto più dinamico ed estendibile perché non prevede la conoscenza statica dell'ubicazione dei servizi e la topologia della rete. Questa scelta è un po' più limitata per quanto riguarda la configurabilità da parte dei gestori, ma si potrebbe intendere il primo client instanziato come quello utilizzato dai gestori per inizializzare il sistema.

3.3.2 Connessione dei client successivi al primo

La procedura di configurazione del sistema in seguito alla richiesta del servizio da parte di utenti successivi al primo presenta una problematica non troppo differente da quella appena presentata per la connessione del primo client e quindi della prima configurazione del servizio.
Potrebbero infatti presentarsi due alternative che verranno ora spiegate, le quali però sono sostanzialmente equivalenti nel caso di topologie semplici formate da una località per il server , una per il proxy e una per i client, ma si rivelano abbastanza differenti in caso di possibili implementazioni di topologie più complesse.

- I nuovi client richiedono al sistema dove si trovi il proxy più vicino al client che ha inizializzato il servizio per la presentazione da loro richiesta. Ricevuto l'indirizzo con la risposta vi si connettono, e ricevono da esso il flusso già attivo. Nella seguente figura viene mostrato graficamente questo caso utilizzando una topologia più complessa di quella semplice che sarà utilizzata per lo sviluppo di questa tesi.

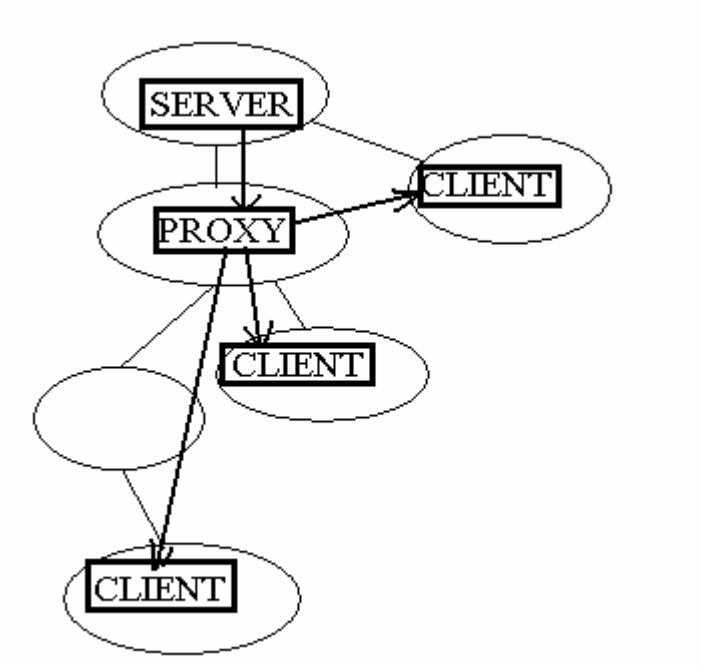

Fig 3.1 Soluzione in cui tutti i client si connettano allo stesso proxy.

- I nuovi client si comportano come se fossero i primi a connettersi. Avviene quindi la ricerca automatica del nodo su cui è presente la presentazione e viene lanciata la procedura di inizializzazione dei componenti sul percorso. A questo punto però è presente la sostanziale differenza rispetto alla prima inizializzazione: se sul percorso vengono rilevati proxy già attivi per quella presentazione, viene salvato il loro endpoint e quindi verranno inseriti nel proprio percorso senza doverli nuovamente istanziare. Nella seguente figura viene mostrato questo caso.

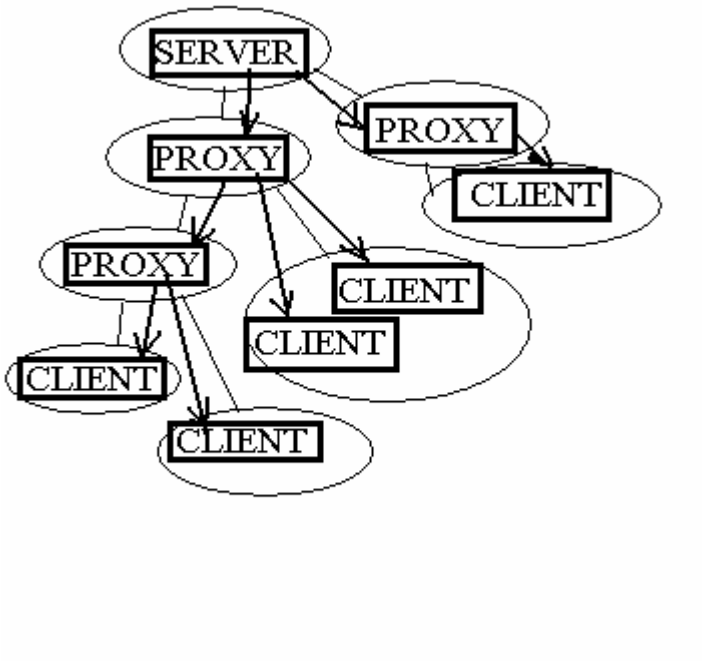

Fig 3.2 Caso in cui ogni client inizializza un percorso.

La prima soluzione è poco dinamica e relativamente simile al primo caso analizzato nel paragrafo precedente.
E' stata considerata preferibile la seconda soluzione, poiché permette l'instanziamento automatico dei proxy in modo ragionato, sul percorso che collega i client al server. Se un client si trova sullo stesso nodo di un altro client che sta già ricevendo la stessa presentazione non dovrà far altro che collegarsi al primo proxy che trova sul percorso, lo stesso che è già stato inizializzato. Questa soluzione prevede che possano formarsi delle vere e proprie reti di proxy, che si diramano ad albero a partire dal server principale. Quindi, maggiore sarà il numero di clienti su nodi diversi della rete

esistente, maggiore sarà il numero dei proxy presenti e minore sarà il tempo di inizializzazione della connessione.
Questa soluzione è stata ritenuta la migliore poiché prevede un sistema più dinamico e con una distribuzione dei carichi già più efficiente rispetto alla prima soluzione, che prevede tutto il carico su un solo proxy. In entrambi i casi è comunque possibile effettuare un load balancing in locale, generando nuovi proxy nella stessa località per la gestione delle nuove connessioni.

3.3.3 Parametro per il riconoscimento della threshold

Un altro problema affrontato è stato quello della scelta del parametro da mantenere in osservazione per il controllo del carico sui nodi che ospitano proxy attivi. Come è stato visto in precedenza l'operazione di impacchettamento e spacchettamento dei dati, e eventuali codifiche, decodifiche e renderizzazioni sono operazioni molto onerose in termini di impegno dei processori delle macchine su cui vengono effettuate. Il dover mantenere molte connessioni aperte con molti clienti implica anche un ingente impiego di banda trasmissiva. L'architettura prevista infatti prevede una realizzazione su rete di tipo Internet, non solo locale. Questo fatto implica l'impossibilità di utilizzare meccanismi di multicast, ed è quindi necessario aprire connessioni end-to-end da un proxy a tutti i client che vi si collegano. Il contesto presentato per questa tesi implica la scelta di un modello di rete ben preciso, che valorizza le scelte effettuate. Si può presupporre infatti di avere client, proxy e servitori posizionati su domini diversi, mentre i proxy sono invece tutti posizionati all'interno dello stesso dominio. Un dominio può essere inteso come una rete locale di calcolatori, mentre il livello più ampio, quello che contiene tutti i domini può essere inteso come la rete internet. E' possibile quindi fare la considerazione di avere le connessioni all'interno della rete locale di tipo molto veloce, mentre quelle fra i domini di tipo più limitato in banda. Considerazioni sulla banda però sono meno rilevanti rispetto a quelle del calcolo computazionale, in quanto comunque è possibile supporre la banda

disponibile ai server proxy come molto ampia, e comunque migliorabile nel tempo. La verifica di questi livelli di carico verranno affrontati in modo sperimentale alla fine di questa tesi, e probabilmente potranno essere sviluppati in modo più approfondito in progetti successivi.

Nel caso analizzato si è deciso di scegliere come parametro di riconoscimento del livello di soglia il numero di connessioni correntemente attive, scegliendo un valore non reale da utilizzare durante la progettazione. Nella fase di test verranno eseguiti controlli sul consumo effettivo della CPU dei calcolatori utilizzati e sarà quindi possibile fare una stima del numero di connessioni contemporaneamente attive sopportabili da diversi tipi di computer dedicati.

Sono lasciate a future implementazioni controlli automatici sullo stato della CPU, in modo da permettere una soluzione più dinamica e indipendente dal tipo di macchina su cui si è deciso di implementare il servizio. Un servizio di questo tipo potrebbe avvenire per mezzo di chiamate ad apposite primitive di monitor di sistema, come ad esempio controlli sulla percentuale di CPU e sulla banda effettivamente impegnata.

3.3.4 Differenziazione fra tipi di proxy esistenti

E’ necessaria una scelta riguardo al fatto di realizzare entità proxy tutte appartenenti allo stesso tipo oppure differenziarle a seconda delle loro funzioni. Per la natura delle scelte precedentemente effettuate ci si trova nella condizione in cui, in seguito alla richiesta della stessa presentazione, tutti i client che hanno in comune lo stesso percorso si collegano ad uno stesso proxy, sul nodo del path più vicino a loro. Anche in seguito all’instanziamento di nuovi proxy all’interno della stessa località del primo, i client continueranno a collegarsi sempre allo stesso proxy, e solo in una seconda fase saranno rediretti su proxy più scarichi. E’ chiaro quindi come il tipo di servizio presentato presenti un proxy dotato di un’importanza maggiore rispetto agli

altri, in quanto il primo proxy attivato è quello che riceve tutte le nuove connessioni dei client e quindi è colui che smista i flussi ai nuovi proxy.

Le alternative di progettazione in questo caso potrebbero quindi essere due:

- Creare proxy tutti dello stesso tipo. Questo implica un sistema più dinamico, ma intrinsecamente più complesso. In questo caso si ha comunque un proxy centrale che riceve le connessioni e le smista ai proxy secondari, ma la principale differenza sta nel fatto che utilizzando proxy tutti uguali è necessario mantenere in ognuno di essi una tabella contenente gli stati di tutti gli altri, e un sistema di coordinamento, basato su multicast locali a tutta la rete del dominio in seguito ad ogni nuova connessione e disconnessione, e la gestione degli errori in caso di caduta di un nodo sarebbe complicata.
- Creare un proxy principale differenziato rispetto ai proxy secondari implicherebbe una maggiore responsabilità di gestione attribuita al suo nodo. La tabella con le caratteristiche dei proxy attivati sarebbe mantenuta solo su questo nodo principale, e tutti i proxy secondari manterrebbero un rapporto diretto con esso, senza curarsi degli altri nodi. Tutti i messaggi di stato ed errore verrebbero inviati in questa direzione. Questa soluzione da un lato prevede minori problemi in caso di caduta di un proxy secondario, ma problemi molto piu grossi nel caso in cui a cadere sia un proxy principale, poiché tutti gli altri ne risentirebbero e il servizio subirebbe un interruzione più complessa da gestire rispetto a quella che potrebbe avvenire nel primo caso.

In questa tesi si è scelto di adottare la seconda soluzione, poiché prevede una soluzione più chiara e sistematica al problema del load balancing. Non è però esclusa una possibile implementazione futura di un sistema di gestione dei proxy fra pari , intrinsecamente più complessa, ma sicuramente più affidabile.

3.3.5 Protocollo di comunicazione

In seguito al superamento del livello di soglia del carico della CPU del proxy principale il sistema studiato implementa un meccanismo di instanziamento automatico di un proxy secondario. Qualora anche il proxy secondario raggiunga il livello critico di carico viene nuovamente lanciato il sistema di riconfigurazione del sistema per l'instaziamento di un nuovo proxy. Questo meccanisco avviene in modo ricorsivo per tutti i proxy attivi, finché sono disponibili nodi con calcolatori liberi. Nel caso cessino tutte le connessioni su un proxy allora potrà essere implementato un servizio di di riconfigurazione per rimuovere i proxy inutilizzati.

Da quanto visto nei paragrafi precedenti si è deciso che ogni proxy si trova a dover mantenere uno stato contenente il numero di connessioni effettivamente attive, per poter permettere il riconoscimento della soglia critica di carico. Questo stato deve essere incrementato di uno ad ogni nuova connessione effettuata e decrementato di uno ad ogni connessione chiusa in modo corretto o comunque persa in seguito alla caduta di un client o a errori di connessione dovuti a problemi di rete. E' quindi necessario mantenere una conoscenza a livello generale dello stato di tutti i proxy, per fare in modo che quando uno di essi è troppo carico non abbia bisogno necessariamente di instanziarne uno di nuovo, ma ridiriga le nuove connessioni in arrivo sul proxy presente più scarico al momento attuale. Per far questo si è deciso di implementare un protocollo di comunicazione fra i nodi che svolgono il compito di proxy.

Si è presentata a questo punto la scelta del tipo di protocollo da utilizzare per un servizio di comunicazione di questo tipo. I protocolli a disposizione potrebbero essere TCP e UDP.
Analizzando le funzionalità e il tipo di servizi forniti da entrambi i protocolli si è scelto di optare per il protocollo UDP.
Il protocollo UDP offre una semplice interfaccia fra il protocollo di rete IP e il livello applicativo. Si tratta di un protocollo molto leggero in termini di banda, che non presenta meccanismi di

controllo sulla ricezione dei messaggi. Essi possono arrivare in maniera non ordinata o addirittura non arrivare.
Per sua natura è un tipo di protocollo che non prevede di stabilire una connessione e un flusso di dati, ma un semplice scambio di pacchetti. Questa caratteristica è quella che ha fatto optare per la scelta di questo protocollo, poiché è molto vicina al tipo di comunicazione necessario per l'implementazione di questo tipo di servizio. L'eventuale perdita di qualche messaggio potrà essere gestita a livello applicativo con la gestione opportuna di eccezioni e messaggi in grado di riportare il corretto stato del sistema.

Un protocollo di questo tipo, oltre alla gestione degli stati può essere utilizzato anche per la gestione degli errori e degli inconvenienti che potrebbero nascere all'interno del dominio delle entità proxy. Potrebbe infatti accadere che un proxy, per qualche problema tecnico (ad esempio un problema nel calcolatore su cui è eseguito), cessi di essere attivo, e quindi non sia in grado di gestire le connessioni dei client. E' quindi necessario implementare un sistema di controllo, con modo di funzionamento simile al "ping", che preveda un costante controllo dell'effettivo funzionamento dei proxy nella rete.

La scelta affrontata nel precedente paragrafo di differenziare i tipi di proxy si addice al modello prescelto, in quanto tutti i proxy secondari invierebbero i messaggi in modo end-to-end al proxy principale, il quale manterrebbe lo stato di tutti gli altri in una tabella apposita, mentre la soluzione con proxy paritari potrebbe prevedere un meccanismo di scambio di messaggi di tipo multicast, poiché tutti i proxy dovrebbero essere alla conoscenza dello stato di tutti gli altri.

3.3.6 Connessione dei client al secondo proxy

Ultimo problema affrontato è quello di come far sapere al cliente che si collega dopo il raggiungimento della soglia critica che deve ricevere il flusso da un proxy secondario e non dal proxy principale. Questo avviene quando un client richiede una

presentazione, e il proxy a cui si collega decide di redirigerlo su un proxy secondario perché ha raggiunto il limite massimo di connessioni disponibili.
Le alternative che si sono presentate sono le seguenti:

- La responsabilità della gestione rimane a lato proxy. Una volta connesso al primo proxy, il client riceve da esso un messaggio indicante che non può più accettare connessioni, allora si mette in ascolto di una connessione in arrivo. Il primo proxy comunica ad un proxy secondario che deve iniziare lo stream al client appena arrivato e quindi questo richiede una connessione verso il client. Dopo questa fase può iniziare lo stream.
- La responsabilità della gestione è affidata al client. In questo caso invece, appena il client si connette al proxy riceve da esso l'avviso che non può ricevere più connessioni perché è troppo carico. Questo avviso si concretizza con l'invio dell'endpoint del nuovo proxy instanziato. Il client allora si connette al nuovo, e intraprende con esso una nuova fase di inizializzazione della sessione.

La prima soluzione prevede di affidare al secondo proxy il compito di connettersi al client, mentre la seconda prevede che sia il client a connettersi al nuovo proxy.
Non sono state riscontrate grosse differenze nelle due implementazioni, poiché entrambi prevedono la modifica del codice del client. La seconda prevede un impegno minore da parte di un proxy già carico, e quindi porterebbe a sollevarlo da un compito in più. La prima invece prevede che il client non conosca l'endpoint del nuovo proxy. E' però da notare che la prima scelta risulta più naturale nel per la scelta di avere un proxy principale e proxy secondari, mentre la seconda sarebbe invece migliore se distribuissimo il protocollo di load balancing facendo proxy tutti uguali.

E' stato quindi deciso di scegliere per questo progetto di tesi la seconda opzione, ma non è esclusa che una futura implementazione del codice possa adottare il primo modo, causa possibili motivi di scelte architetturali differenti.

CAPITOLO 4

PROGETTAZIONE DEL SISTEMA

In questo capitolo e nei seguenti ci accingiamo a illustrare, seguendo le scelte implementative illustrate nei precedenti paragrafi, la fase di progettazione del sistema di load balancing.
Il sistema da realizzare sarà un sistema di broadcast multimediale per il quale sarà studiato un apposito meccanismo di load balancing dei proxy. Lo scopo del meccanismo di load balancing studiato è quello di impedire l'aumento del carico su proxy che hanno già raggiunto il limite massimo di connessioni disponibili e trasferire tale compito su un altro proxy, già esistente o ancora da creare.

Come già evidenziato la progettazione sarà suddivisa per chiarezza in fasi successive, tutte legate fra loro.
La spiegazione quindi sarà suddivisa nelle seguenti parti:

- Creazione di un sistema di broadcast multimediale, agendo sul sistema già esistente di MUM.
- Implementazione di un servizio per il un controllo sul livello di carico del proxy.
- Implementazione di un sistema di creazione di nuovi proxy e gestione delle procedure di inizializzazione della sessione all'arrivo di nuove connessioni.
- Implementazione di un sistema di comunicazione fra i proxy creati per mantenere un livello di carico ideale fra i nodi esistenti.

4.1 TOPOLOGIA DI SVILUPPO

Si è deciso innanzitutto di creare, sfruttando le funzioni fornite dall'ambiente SOMA, una topologia semplificata adatta al tipo di servizio di broadcast che si vuole realizzare.
La topologia creata prevede la presenza di tre domini collegati in cascata.

- Il primo dominio è quello che contiene il dato multimediale di cui si desidera eseguire lo stream. Questo è il luogo in cui di conseguenza viene instanziato l'unico server previsto, che invia un flusso di dati continuo verso il proxy.

- Sul secondo dominio, figlio del dominio radice contenente la presentazione, è presente un numero di place qualsiasi maggiore di uno, nella figura si è deciso di mostrarne tre. Questo dominio può essere considerato l'equivalente di una rete locale, in cui su ogni nodo è presente un server proxy. Questa rete locale è collegata all'esterno per mezzo di un gateway, che corrisponde alla posizione del default place. Sul default place viene instanziato il proxy principale, sui place normali sono invece lanciati i proxy secondari al momento del bisogno.
- Il terzo ed ultimo dominio, figlio del dominio contenente i proxy, è quello su cui si suppone siano presenti i client degli utilizzatori del servizio.

Per facilità di memorizzazione si è deciso di chiamare "root" il nodo su cui è presente il server, "europe" il dominio contenente i proxy (rispettivamente, il proxy principale è chiamato europe, i secondari europe2 ed europe3), "italy" il dominio a cui si collegano gli utenti.

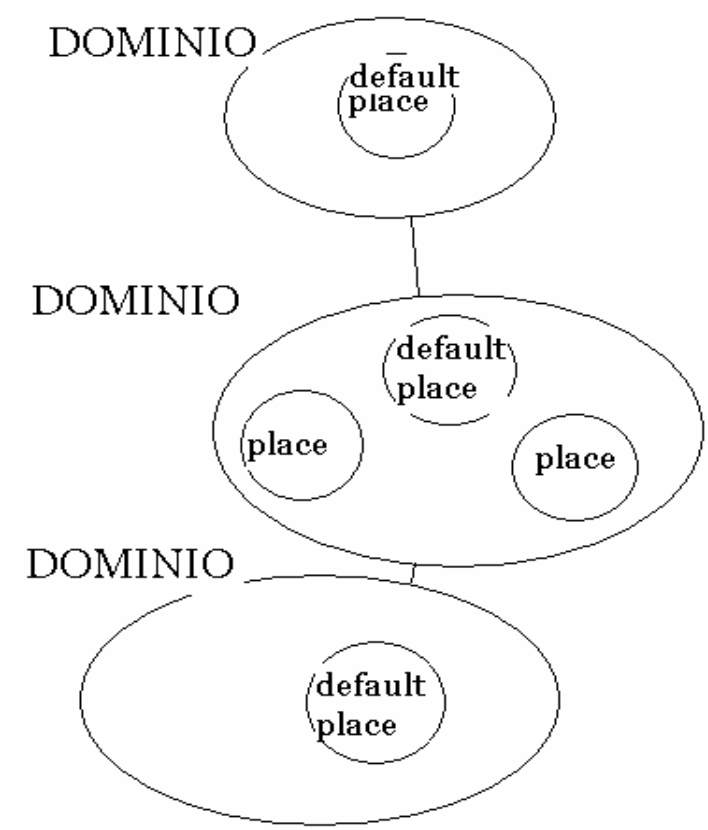

Fig. 4.1 – Rappresentazione della topologia di esempio

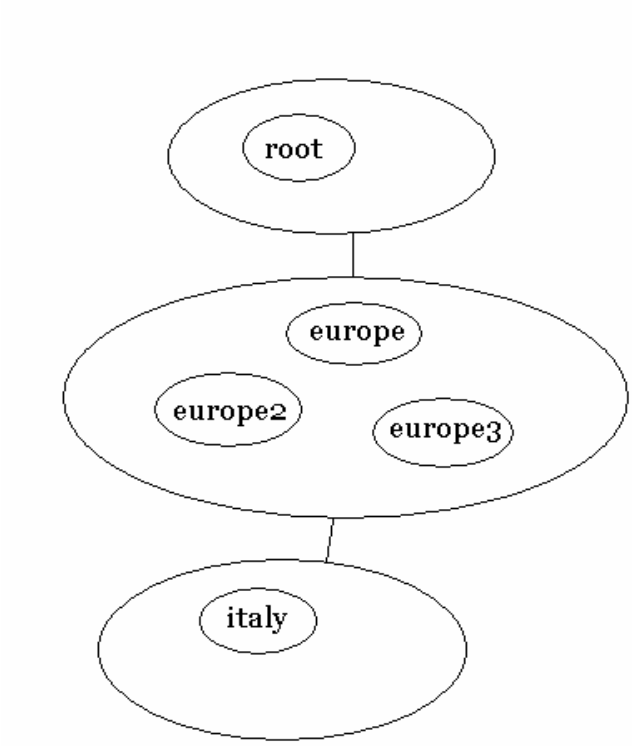

Fig. 4.2 – Altra rappresentazione della località di esempio

4.2 SISTEMA DI BROADCAST

Per creare un sistema di broadcast, come già detto in precedenza, è necessario modificare la struttura di inizializzazione delle entità già presente in MUM.

Nell'analisi delle problematiche si è spiegato come sia stato deciso di realizzare un sistema che prevede una prima inizializzazione del sistema a seguito della prima richiesta da parte di un client e quindi un'inizializzazione parziale da parte che dei client successivi, che prevede la possibilità di collegarsi a path già esistenti.

La soluzione creata sarà creata a partire dal servizio già esistente di streaming video end-to-end presente nell'implementazione attuale di MUM.

Vediamo ora come sia possibile suddividere questa sezione progetto in fasi successive; le illustriamo brevemente:

- Implementazione di un metodo di inizializzazione del sistema che permetta di riconoscere i proxy già attivi sul percorso.
- Creazione di proxy in grado di gestire più connessioni contemporaneamente.
- Disabilitazione dei comandi di controllo del flusso da parte degli utenti.

4.2.1 Inizializzazione

Detto questo possiamo entrare nel vivo della progettazione, partendo dalla creazione del sistema di inizializzazione.

Abbiamo visto come la situazione già esistente in MUM possa essere utilizzata anche per la prima inizializzazione del sistema. Il sistema però non permette il broadcast a più, utenti poiché ad ogni richiesta vengono instanziati nuovi proxy e nuovi servitori lungo il percorso, ognuno che funziona in modo parallelo a quello già esistente. E' quindi necessario agire sul funzionamento dell'agente *PlanVisitorAgent.*

La modifica da effettuare per permettere l'unione dei percorsi fra server e client in presenza di nodi comuni interessa il codice del metodo *visitSinglePlan()*. Questo metodo presenta una serie di casistiche. Queste casistiche consistono nel valore dell'attuale *PlanEntry*, vale a dire il tipo di entità da attivare sul nodo corrente. E' quindi presente la fase di inizializzazione del client, quella dei proxy e quella del server. E' necessario andare a modificare quindi la fase di inizializzazione del proxy, in quanto non è sempre necessario inizializzarne uno nuovo, ma se è presente già un proxy sul nodo corrente allora bisogna collegarsi ad esso e ricevere il flusso che ne esce ed è già attivo per il primo utente che ha richiesto il servizio. E' quindi opportuno inserire un controllo delle entità presenti sul nodo attuale, e se è già presente un proxy che sta eseguendo lo stream della presentazione corrente, eseguire una istruzione nuova, altrimenti creare come avveniva in MUM una nuova entità proxy. Nel caso in cui l'entità proxy sia già esistente è allora necessario rilevarne l'end point (*ComponentInfo*), e salvarla

per poi inviarla all'indietro lungo il percorso, all'entità presente sul nodo precedente. Seguendo la topologia studiata l'endpoint arriverà quindi al primo client.

Nella seguente figura viene mostrato come i client che richiedono l'inizio del flusso dopo la prima inizializzazione uniscano il proprio path a quello già esistente, senza dover istanziare nuove entità.

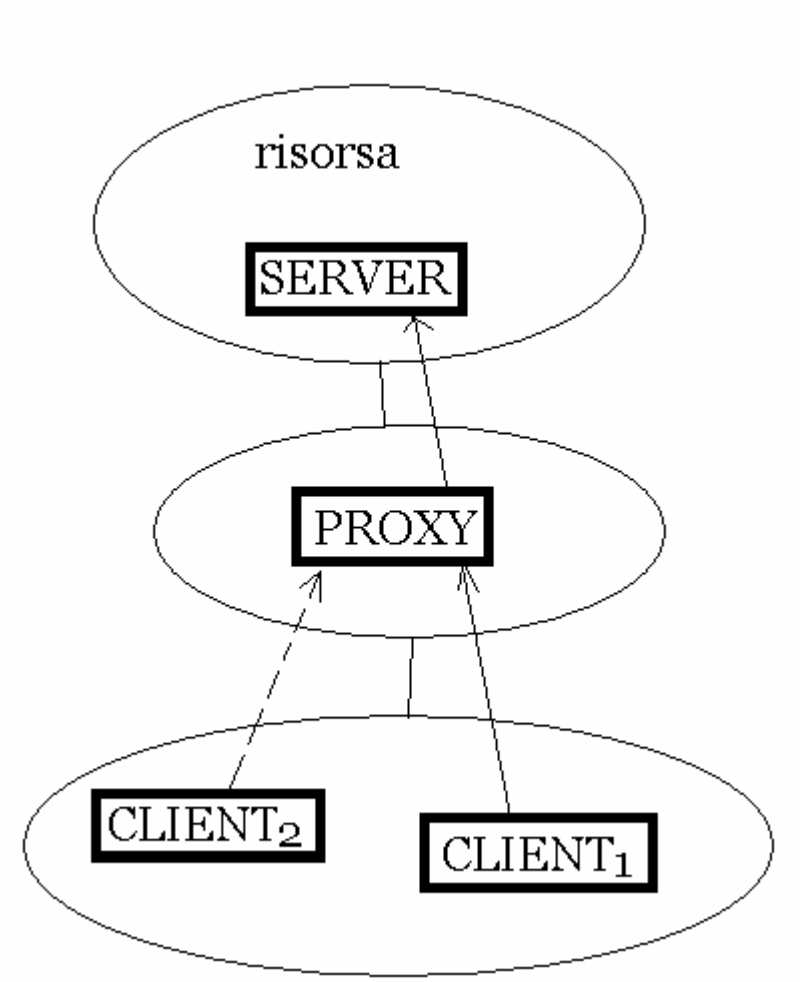

fig. 4.3 – Rappresentazione della inizializzazione in progettata

4.2.2 Supporto a più clienti sul proxy

A questo punto abbiamo permesso la creazione di path con le parti terminali dei percorso in comune a più clienti contemporaneamente. Questo è fondamentale per la creazione del sistema di broadcast.

Ora però si presenta un problema: il proxy attualmente studiato non è in grado ci gestire più di una connessione alla volta.

La classe interessata per il controllo del numero di connessioni in ingresso sul proxy è la classe di grande importanza chiamata *ProxyInProtocolUnit*. Questa classe esegue una *accept()* sospensiva, in attesa di ricevere la connessione da parte di un cliente. Questo metodo è quello fornito dalle *ServerSocket* di Java, e permette di bloccare l'esecuzione della classe che lo chiama fino a che non arriva una connessione in ingresso.

In seguito allo stabilimento della connessione il proxy attende di ricevere un comando da parte del client, che può essere la richiesta di inizio della trasmissione, della sua terminazione, della sua pausa ecc.. E solo in seguito esegue il metodo corrispondente all'azione richiesta.

Questa struttura deve essere cambiata per permettere la gestione di più connessioni contemporaneamente.

E' quindi necessario inserire un ciclo sul metodo di ascolto in ingresso. In questo modo in seguito ad ogni connessione avvenuta il proxy sarà subito disponibile in ascolto per l'arrivo di una nuova connessione. Questo è possibile perché le funzioni di streaming sono state create in modo da essere divise da quelle di ricezione dei comandi. Questa scelta è stata effettuata in MUM per avere una maggior chiarezza nell'esecuzione, e si presenta molto utile nel caso analizzato. E' infatti necessario spiegare che il ciclo sulle *accept()* è possibile poiché il flusso multimediale è gestito da un'altra classe che ha un funzionamento di tipo Thread. Se non fosse così una volta avvenuta la connessione non sarebbe possibile affrontarne una nuova fino a che la prima non sia terminata. Si ha quindi un proxy che gestisce le connessioni in modo concorrente, per cui le nuove connessioni non devono rimanere in attesa del termine di quelle correntemente attive ma possono avvenire in parallelo ad esse.

Nella seguente figura viene mostrato come effettivamente il livello di accettazione delle nuove connessioni rimanga separato da quello dell'invio dei flussi.

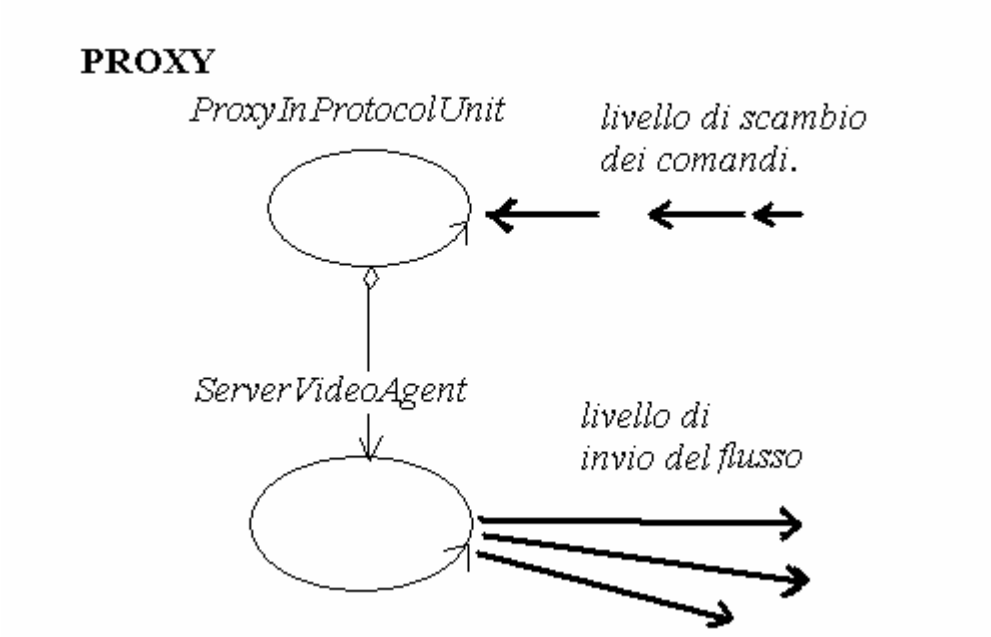

Fig 4.4 Differenziazione fra livello di comandi e di flusso

E’ poi opportuno eliminare la possibilità di ricevere comandi da parte del client diversi da quello per unirsi alla connessione.
Le eventuali disconnessioni devono essere rilevate utilizzando le funzioni di java per il rilevamento delle disconnessioni sulle socket.
Disabilitati i comandi e attivata la possibilità di avere connessioni multiple è ora necessario analizzare il metodo *beginStreaming().*
Questo metodo è quello che viene eseguito in seguito al ricevimento del messaggio di *JoinStreaming* inviato dal client.
Seguendo le scelte implementative spiegate in precedenza è evidente che questo messaggio indica il desiderio di unirsi alla trasmissione corrente, oppure di ricevere la prima trasmissione nel caso in cui questa non sia ancora stata inizializzata.

In MUM, in questo metodo è già presente la funzione di join su un flusso che non è ancora stato inizializzato. Questo viene eseguito leggendo il flusso di ingresso che il proxy riceve dal server, e redirigendolo verso il client.
Viene inoltre creato un clone del *DataSource* (che abbiamo già descritto nei capitoli precedenti) in ingresso e viene effettuata la scrittura su un disco locale dei dati in ingresso. Questo passaggio è stato inserito in una precedente tesi per implementare il meccanismo di caching locale.

La soluzione per la creazione di un nuovo flusso in uscita può essere trovata sfruttando il concetto di *DataSource* clonabili.
Infatti si può innanzitutto inserire un controllo sul fatto che sia già presente o meno un *DataSource* in uscita attivo.
Se il *DataSource* non è ancora attivo allora si prosegue seguendo il procedimento già presente in MUM.
Qualora invece esso sià presente e già attivo allora è necessario implementare il codice per la gestione di un nuovo flusso.
E' quindi necessario creare un clone del *DataSource* già esistente e quindi redirigerlo in uscita. Questa soluzione viene applicata automaticamente ad ogni nuova connessione, in quanto ogni client riceve i dati da un nuovo DataSource.

4.2.3 Disabilitazione dei comandi

La disabilitazione dei comandi di gestione del flusso, come è già stato detto, è necessaria poiché in un sistema di broadcast multimediale non deve essere possibile per gli utenti comandare la presentazione, poiché questo comporterebbe il fatto che gli effetti dei comandi verrebbero viste da tutti gli utenti. E' quindi necessario permettere agli utenti la sola possibilità di unirsi al flusso di ricezione dello stream e quella di abbandonarlo.
In parte questo è già stato fatto nel precedente paragrafo, disabilitando la ricezione dei comandi sul server.
Sul lato client a questo punto la disabilitazione è ancora più semplice, in quanto basta modificare l'interfaccia grafica disponibile eliminando le funzioni di pausa, rewind ecc.

4.2.4 Conclusione

Le modifiche apportate al sistema MUM hanno reso quindi possibile la creazione di un servizio di tipo broadcast. Ora gli utenti possono richiedere una presentazione e riceverla automaticamente. Se essa non è ancora stata richiesta viene lanciato il meccanismo di

inizializzazione del sistema e la presentazione parte dall'inizio, altrimenti viene effettuata la ricezione di quello dello stream che è attualmente attivo.

Questo sistema però non prevede un controllo sul carico effettivo sui nodi proxy né sul server. Il server si può considerare come non troppo sollecitato, poiché da esso parte un unico stream che va verso il proxy. Non risente per questo del numero di clienti correntemente collegati. Il vero problema si pone sul nodo del proxy, in quanto il processo di duplicazione dei *DataSource* è computazionalmente impegnativo e quindi un elevato numero di connessioni attive potrebbe portare il sistema a saturare e a non essere in grado di fornire correttamente i flussi multimediali agli utenti. E' quindi ora necessario entrare nel vivo della tematica del load balancing, progettando un sistema per ridurre al minimo i problemi di sovraccarico dei proxy.

4.3 CONTROLLO SUL LIVELLO DEL CARICO

Come si è accennato nell'analisi delle problematiche si è deciso di inserire il controllo del livello del carico come controllo sul numero di connessioni effettivamente attive. E' lasciato a implementazioni future un possibile controllo sull'impegno effettivo della CPU.

Il numero di connessioni attive sopportabili da un calcolatore sarà ricavato con test sperimentali nella fase finale di questa tesi, in cui verrà eseguito il sistema studiato su varie macchine controllando il livello di impegno del processore.

Il controllo consiste in un gestore che deve essere sempre a conoscenza del numero di connessioni attive e, se esse sono in numero maggiore o uguale a un livello stabilito, lanciare l'apposita procedura di gestione. E' quindi da inserire una funzione di notifica dell'arrivo di ogni nuova connessione in ingresso, e una per ogni disconnessione avvenuta.

La notifica è implementata seguendo le regole della programmazione ad eventi, sfruttando le possibilità offerte dal linguaggio Java.

La programmazione ad eventi è un paradigma di programmazione che permette a livello concettuale di allontanarsi dal solito flusso sequenziale di istruzioni utilizzato dalla programmazione classica.

Questo paradigma prevede la presenza di una serie di gestori di eventi, che sono chiamati a seguito della realizzazione di un particolare evento per il quale si sono "messi in ascolto". Le chiamate dei gestori degli eventi sono solitamente effettuate da un "dispatcher". Questo tipo di programmazione permette di slegarsi dal classico flusso di esecuzione permettendo una gestione "asincrona". Tipicamente viene impiegata nella realizzazione di interfacce grafiche.

In Java un evento deve essere lanciato in una certa classe a seguito del verificarsi di una certa condizione ed esso viene gestito dal metodo di un'altra classe, che solitamente viene chiamata "*listener*".

Seguendo questa linea guida si è creata una classe che rappresenta l'evento dell'arrivo di una nuova connessione. Questa classe è stata chiamata *ConnectionToProxyEvent*, e contiene al suo interno le informazioni riguardo alle *ComponentInfo* del client che ha richiesto la connessione e le e quelle che riguardano la presentazione richiesta intesa come *IMetadata*.

E' stata creata un interfaccia che identifica il listener per quel tipo di evento. Questa Interfaccia è stata chiamata *IConnectionToProxyListener* . Questa contiene un metodo chiamato *connectionToProxy()* che nell'implementazione di questa classe conterrà il codice da eseguire a seguito dell'arrivo di una nuova connessione.

Detto questo analizziamo l'implementazione effettiva di questo meccanismo.

La classe che implementa l'interfaccia *IConnectionToProxyListener* è l'agente ProxyAgent. Questa sarà infatti la classe che gestirà l'evento di arrivo della nuova

connessione. Questa classe nella fase di inizializzazione passa un riferimento a sé stessa, in qualità appunto di *IConnectionToProxyListener* al componente *Proxy* il quale invia questo riferimento alla classe *ProxyInProtocolUnit*.
In questo modo la classe ProxyInProtocolUnit contiene il riferimento alla classe ProxyAgent e, in seguto all'arrivo di una nuova connessione, vale a dire dopo che è stata effettuata la *accept()* sulla *ServerSocket*, può lanciare l'evento *ConnectionToProxyEvent.*
L'evento viene allora gestito dall'handler presente sull'entità *ProxyAgent*, vale a dire dal metodo implementato *connectionToProxy().*

Un procedimento analogo può ora essere implementato per la gestione della disconnessione di un client. Vale a dire, creando un' opportuna classe *DisconnectionFromProxyEvent* e facendo gestire questo evento sempre dalla classe ProxyAgent. L'evento deve essere lanciato a seguito della disconnessione di un client, vale a dire ricevendo la disconnessione attraverso la ServerSocket.

La esecuzione del metodo *connectionToProxy()* su *ProxyAgent* prevede a questo punto l'incremento di uno contatore, corrispondente al numero di connessioni correntemente attive. Il metodo corrispondente per la disconnessione invece decrementa questo stato.

A questo punto è stata creata un'espressione condizionale, che fa il controllo sul contatore del numero delle connessioni. Se il contatore ha raggiunto il livello di threshold (quello che sarà verificato sperimentalmente nella fase di test) il meccanismo di load balancing sarà attivato. Altrimenti non succede niente, e si attente l'arrivo della prossima connessione.

4.4 CREAZIONE DI UN NUOVO PROXY

Come è stato analizzato nella sezione riguardante le problematiche di progetto, il raggiungimento della threshold prevede:

- Creazione di un nuovo proxy su un place appartenente allo stesso dominio del proxy principale.
- Passaggio delle sue *ComponentInfo* al client e inizio del flusso della presentazione richiesta.

Mostriamo graficamente i passaggi che avvengono nel caso attualmente analizzato.

Nella seguente figura viene mostrato come il proxy principale sia già attivamente impegnato nell'invio di flussi verso dei client. A questo punto arriva la richiesta di inizio del flusso da parte di un nuovo client.

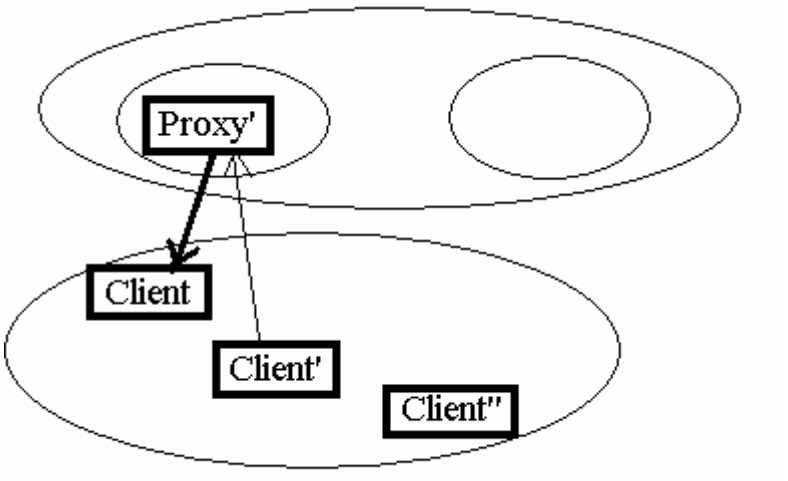

Fig. 4.5 – Il proxy primario è attivo e riceve la richiesta da un nuovo client.

Il proxy principale è in grado di soddisfare la richiesta arrivata e quindi inizia il flusso. A questo punto però scatta il superamento della soglia di carico, per cui viene inizializzato un proxy secondario che gestirà le connessioni future. Terminata l'inizializzazione il proxy principale riceve l'endpoint del nuovo proxy.

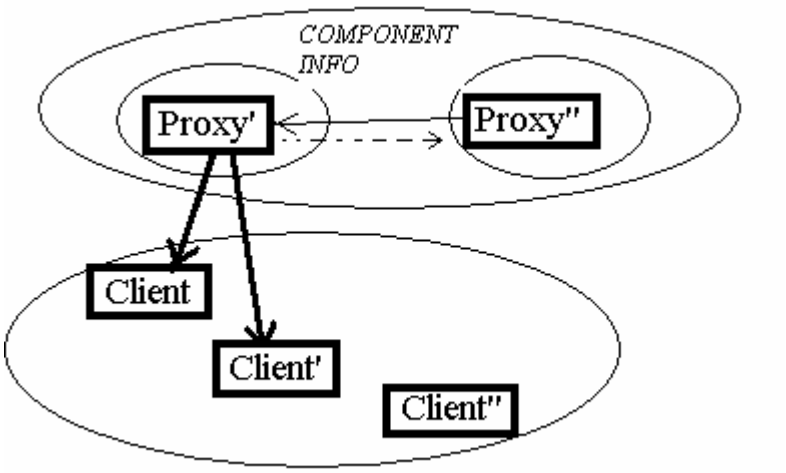

Fig. 4.6 – Inizia il flusso verso il client e viene istanziato un nuovo proxy.

E' ora possibile iniziare lo stream del flusso video verso la nuova entità creata. A questo punto, se un nuovo client richiede l'inizio della presentazione il proxy principale gli invia l'endpoint di quello secondario.

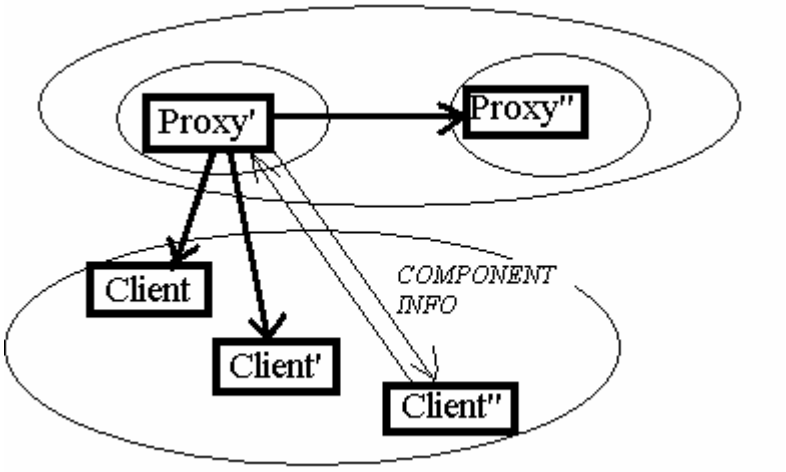

Fig. 4.7 – Inizia il flusso verso il proxy secondario e viene inviato il suo endpoint al nuovo client.

Ricevute queste informazioni il client allora effettua la disconnessione dall'entità principale e richiede l'inizio del flusso a quella secondaria, la quale può soddisfare la richiesta essendo al di sotto del proprio livello di soglia.

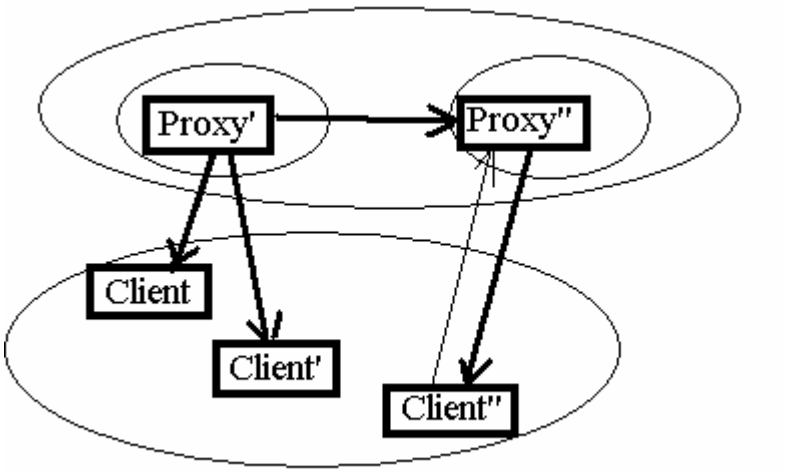

fig. 4.8 – Viene rieffettuata la negoziazione fra il nuovo client e il proxy secondario.

Iniziamo con il progettare la prima parte, quella dell'inizializzazione del proxy secondario.

Sono stati realizzati due agenti specifici per questo compito: l'agente *SecondaryProxyInstantiatorAgent* e l'agente *ProxyMobileAgent.*

A seguito del rilevamento del raggiungimento della threshold sul numero di connessioni l'agente *ProxyAgent* svolge innanzitutto una ricerca per verificare che esistano proxy secondari già attivi che non abbiano ancora raggiunto il livello di soglia di carico, oppure che siano presenti Place liberi all'interno del dominio in cui è in esecuzione. Se non vengono rilevato proxy secondari disponibili è necessario crearne uno nuovo.

Nelle seguenti figure viene mostrata la fase i inizializzazione di un nuovo proxy, che ci accingiamo a spiegare.

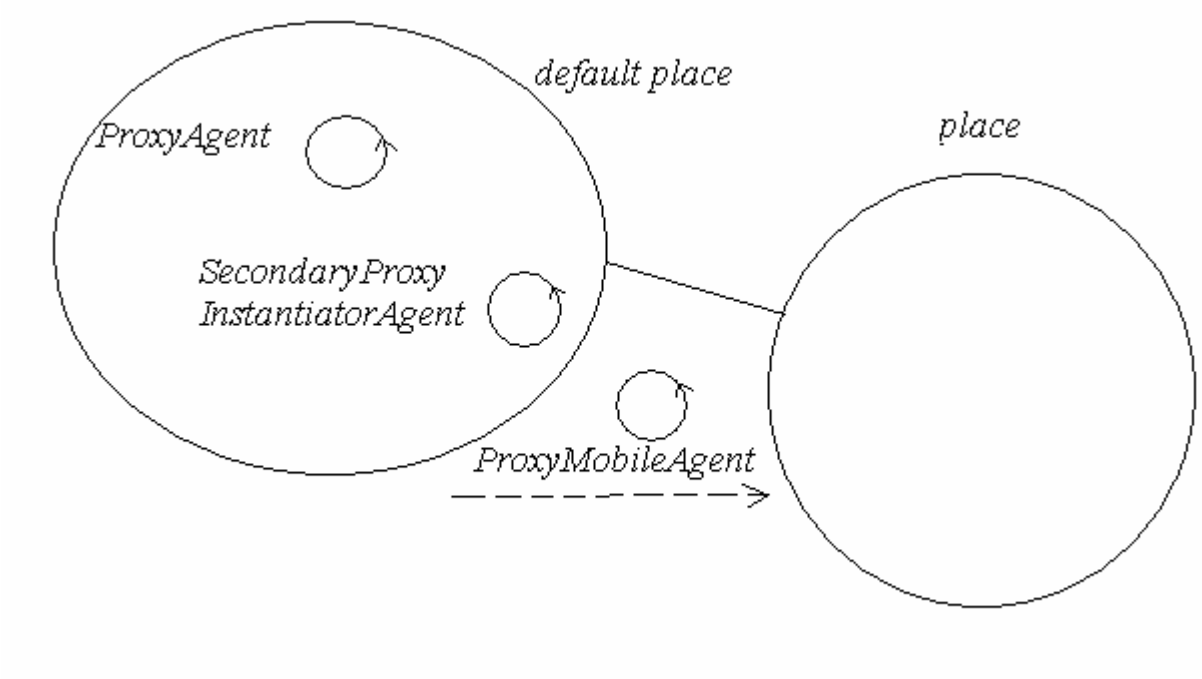

Fig 4.9 - Creazione di un agente mobile che si reca sul nuovo place e di uno che rimane in ascolto.

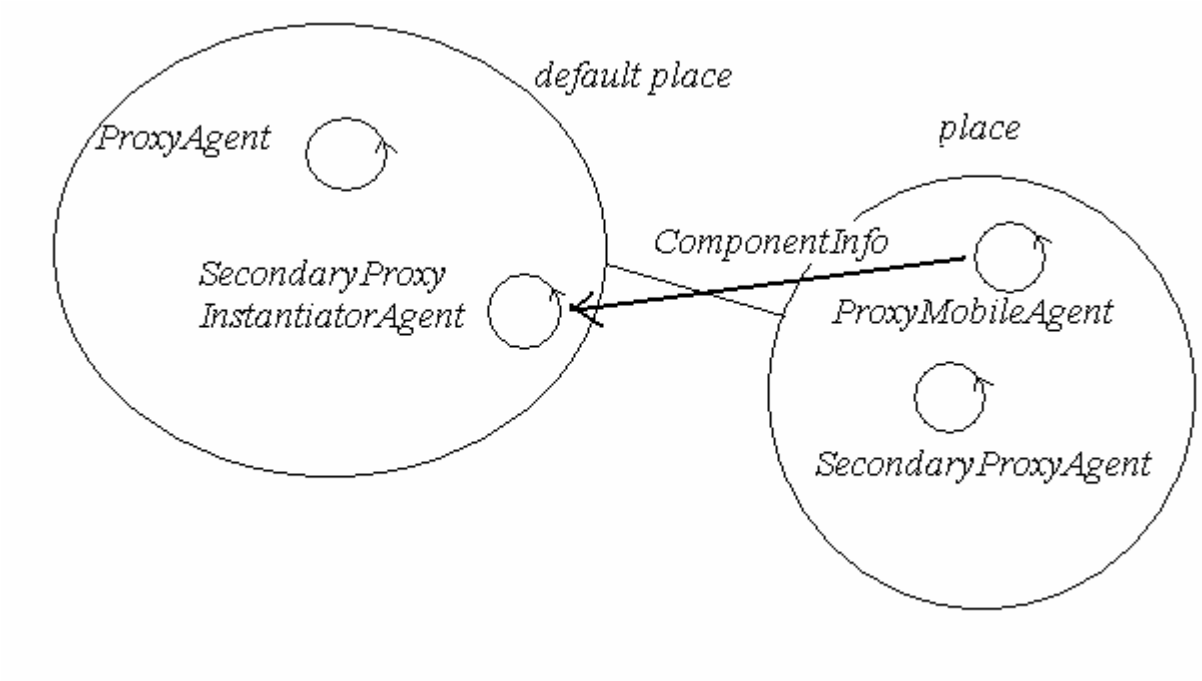

Fig 4.10 – creazione di un proxy secondario e invio delle sue *ComponentInfo* al primario.

Nel caso in cui non siano presenti proxy secondari in grado di accettare nuove connessioni viene creato un nuovo agente chiamato *SecondaryProxyInstantiatoAgent*. Questo agente ha il compito di creare un altro agente, chiamato *ProxyMobileAgent* : si tratta di un

agente con il compito di spostarsi sulla topologia, e il nodo su cui andrà a continuare la sua esecuzione viene scelto fra i nodi liberi nella località considerata. Una volta creato viene quindi inviato sul nodo su cui dovrà essere istanziato il proxy secondario.

Giunto a destinazione il *ProxyMobileAgent* richiede al sistema il download del codice per poter creare un proxy secondario. Terminato il dowload, questa nuova entità viene inizializzata con le informazioni per ricevere lo stream dal proxy principale e il suo endpoint viene inviato dal *ProxyMobileAgent* al *SecondaryProxyInstantiatorAgent*. Il *SecondaryProxyInstantiator-Agent* a questo punto scatena un evento che corrisponde alla notifica di fine di inizializzazione del proxy secondario. L'evento viene recepito dal *ProxyAgent*, il quale ricava da esso l'end-point della nuova entità costituita.

A questo punto il proxy secondario richiede l'inizio del flusso video. Questo viene effettuato subito per evitare possibili corse critiche in caso di richieste di partecipazione alla connessione da parte dei client molto ravvicinate, che potrebbero portare dei problemi di sincronizzazione dei meccanismi di istanziamento. Questo modo di agire rende poi più veloce l'inizio degli stream in uscita.

Non appena il proxy principale riceverà la richiesta di una nuova connessione potrà inviare al richiedente l'endpoint disponibile ad inviargliela.

E' quindi a questo punto necessario apportare una modifica al componente client. A seguito della connessione infatti, deve essere pronto a leggere un messaggio in ingresso, contenente l'indirizzo del proxy a cui essere rediretto, se necessario.

Letto il messaggio, è prevista la disconnessione dal proxy principale, e la successiva connessione al proxy secondario, il cui flusso in ingresso è incominciato a subito dopo la fine della sua inizializzazione.

A questo punto il proxy principale si può mettere nuovamente in ascolto per l'arrivo di una nuova connessione e, se non verranno effettuate disconnessioni da parte dei client correntemente connessi,

sarà necessaria la redirezione dei nuovi client verso il proxy secondario appena creato.

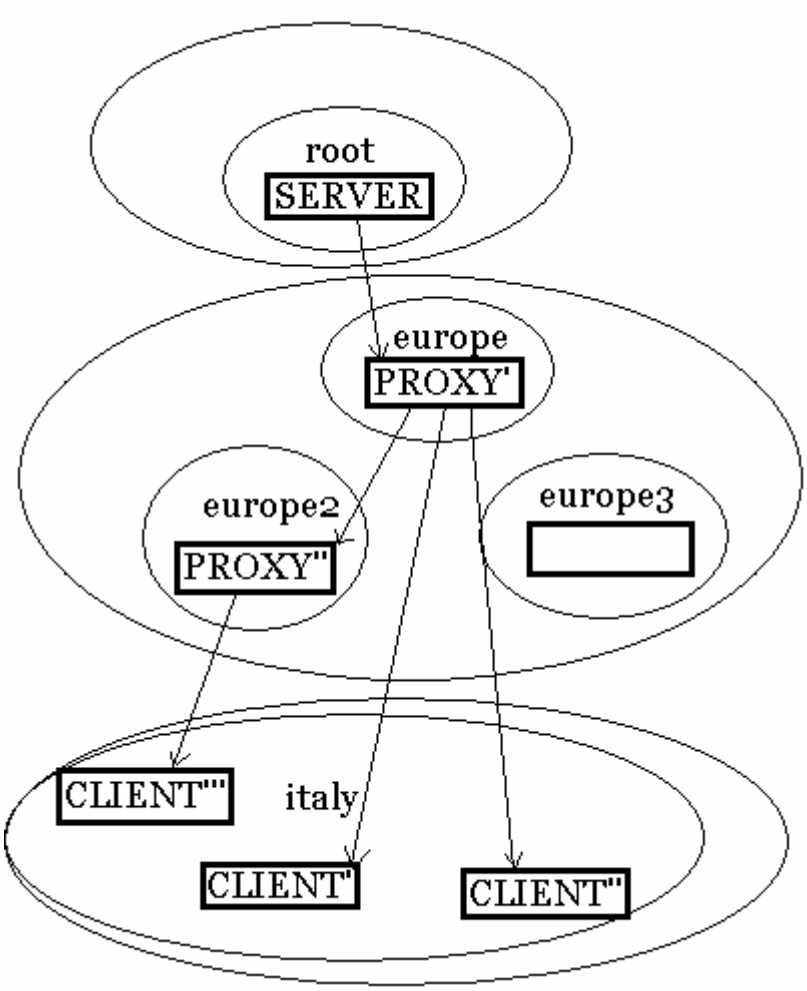

fig. 4.11 – Rappresentazione degli stream nella versione progettata

Sintetizzando i passaggi effettuati, fino a questo punto è stato implementato il sistema di creazione di un proxy secondario, la sua inizializzazione e la ricezione del suo end point di comunicazione. L'endpoint verrà inviato a tutti i client che si collegheranno al proxy principale e sarà loro compito collegarsi alla nuove entità disponibile ad inviare il flusso.

4.5 SISTEMA DI COORDINAMENTO

A questo punto però si pone il problema del coordinamento dei proxy creati, poiché anche il secondo proxy potrebbe non essere in grado di sopportare nuove connessioni e quindi potrebbe essere necessario il lancio di un nuovo proxy. Questo comporta anche il fatto di dover tenere sempre conto delle nuove connessioni e disconnessioni su ogni proxy e verificare che essi siano correttamente bilanciati dal punto di vista del carico computazionale.

Seguendo le indicazioni individuate nell'analisi delle problematiche si è deciso di affidare la responsabilità di gestire lo stato dei proxy esistenti al proxy principale. E' quindi necessario creare una tabella sul nodo principale, contenente su ogni tupla l'endpoint dei proxy instanziati e il numero di connessioni attive su ognuno di essi. In questo modo il proxy principale è in grado di redirigere ogni connessione in ingresso sul proxy al momento più scarico, e se non sono presenti proxy con "slot" liberi, è allora necessario far partire il processo di inizializzazione di un nuovo proxy su un nodo libero.

L'aggiornamento dello stato dei nodi sulla tabella dei proxy viene eseguito per mezzo di un apposito protocollo di comunicazione, con il quale ognuno dei proxy secondari comunica a quello principale il numero di connessioni attualmente presenti sul proprio nodo ad ogni nuova connessione o disconnessione di un client. La scelta di inviare l'attuale numero di connessioni attive piuttosto che un singolo valore indicante l'avvenuta connessione o disconnessione è stata fatta per garantire una maggiore solidità e maggiore probabilità di evitare errori.

La scelta del protocollo UDP è avvenuta per la caratteristica di questo protocollo di avere un funzionamento a scambio di messaggi, molto leggero e quindi in grado di occupare poca banda. TCP non è stato scelto poiché fornisce funzionalità e garanzie sui dati non necessarie per questo tipo di protocollo di aggiornamento.

La realizzazione del protocollo a messaggi viene effettuata in un'apposita classe, che agisce in modo indipendente dall'agente

Proxy, e segnala, per mezzo di eventi l'arrivo di nuovi messaggi contenenti indicazioni sullo stato dei proxy esistenti. Questa classe, lancia eventi di tipo *IncomingProxyStateEvent. ProxyAgent* deve quindi implementare l'interfaccia *IProxyStateEventListener* e gestire l'arrivo dell'evento con un metodo apposito, in cui viene aggiornata la tabella degli stati.

Grazie alla presenza di questa tabella quindi, è possibile realizzare una versione migliore del metodo *connectionToProxy()* presente su ProxyAgent. Ad ogni segnalazione di evento di connessione in ingresso infatti si può implementare un metodo nuovo di redirezione del client nel caso in cui il proxy principale sia troppo carico. Viene quindi effettuato un controllo su tutti i proxy già instanziati, e viene rediretto il client su quello piu scarico. Se non sono presenti proxy scarichi è allora opportuno generarne uno nuovo, e passare le sue *ComponentInfo* al client. Ovviamente a questo punto sarà necessario aggiungere una tupla alla tabella degli stati dei proxy.

Le disconnessioni da parte dei client devono essere mano a mano registrate e i proxy che si trovano a non dover più gestire flussi devono implementare un'opportuna funzione di spegnimento, che comporta l'interruzione del flusso in ingresso, la deallocazione delle risorse impiegate e la fine dell'esecuzione di tutti gli agenti e le classi attive. Nel caso in cui sia a questo punto necessario istanziare un nuovo proxy su questo stesso nodo, allora bisognerà ri-inizializzare tutta la procedura di creazione di un nuovo proxy.

Occorre infine prevedere la possibilità che un proxy su un nodo per qualche motivo non sia più reperibile. Questo può ad esempio avvenire a seguito di uno spegnimento del calcolatore su cui è stato eseguito. Per prevenire questa eventualità è quindi aggiungere al protocollo per la gestione degli stati anche una funzione di controllo sull'effettiva presenza dei proxy. Questo può avvenire implementando una funzione simile al "ping" che verifichi la presenza di essi mandando un messaggio a cui essi dovranno rispondere. Se la risposta non è ricevuta allora è possibile presupporre che il proxy su quel nodo non è più attivo, e quindi è necessario considerarlo come nodo vuoto. Essendo vuoto sarà

possibile instanziarvi un nuovo proxy in futuro. I messaggi di “ping” devono essere inviati prima di passare l’endpoint di un proxy disponibile al client richiedente, in modo da verificare la sua effettiva presenza. I client che erano connessi al proxy che è caduto hanno ricevuto un messaggio di errore. Quindi può a questo punto avviarsi un meccanismo di riconnessione al proxy principale il quale tornerà ad affidargli l’indirizzo di un proxy scarico.

Il caso della caduta del proxy principale è il caso piu grave, poichè nessun proxy potrà ricevere più lo stream richiesto. Tutti i proxy secondari ricevevano il flusso da quello principale e quindi nessuno può piu ricevere il flusso. In questo caso tutti gli utenti ricevono una segnalazione indicante la cessazione della disponibilità del servizio e i proxy presenti sui place vengono rimossi. Una possibile soluzione alternativa potrebbe essere quella di utilizzare un protocollo di elezione grazie alla quale il proxy più scarico verrebbe eletto proxy principale e i nuovi client si connetterebbero direttamente a lui. Questo permetterebbe la fine della trasmissione solo per i client connessi al proxy principale, ma comporterebbe l’inizializzazione di un nuovo flusso da parte del server al nuovo proxy principale e la riconfigurazione dei proxy secondari.

CAPITOLO 5

IMPLEMENTAZIONE

In questo capitolo forniremo qualche dettaglio ulteriore per quello che riguarda la parte implementativa di quelle che possono essere considerate le due funzioni essenziali di questo progetto di tesi. Seguendo le linee guida tracciate dal precedente capitolo mostreremo come sono state implementate le scelte effettuate a lato pratico, mostrando qualche spezzone di codice particolarmente importante e spiegando meglio la struttura delle classi presentate.

L'attenzione sarà posta sulla duplicazione degli stream sul nodo proxy e sulla creazione di un proxy secondario a seguito del superamento del livello di soglia di carico sul proxy principale.

5.1 DUPLICAZIONE DEL FLUSSO

Come presentato in precedenza la classe che, lato proxy, ha la responsabilità di accettare le nuove connessioni da parte dei client è quella chiamata *ProxyInProtocolUnit.* Essa viene inizializzata passandole i rifermenti alle classi *ServerVideoAgent, StreamReceiver* e *ProxyOutProtocolUnit*. Queste classi svolgono la funzione di gestire, rispettivamente, i flussi in uscita, i flussi in ingresso e i comandi in uscita diretti verso i nodi a monte della località esistente.

La classe *ProxyInProtocolUnit* è una classe di tipo Thread, è quindi in grado di essere eseguita in modo asincrono rispetto alla classe da cui è stata lanciata. Possiede perciò un metodo *run()* all'interno del quale viene effettuato l'ascolto ciclico dei delle connessioni che vengono effettuate sulla porta assegnata alla socket dei comandi.

```
…
while (proxyEnabled){
  …
       sock = listenSocket.accept();
  …
}
```

La chiamata di *accept()* è sospensiva, blocca cioè l'esecuzione della classe in questione fino a che non si presenta una connessione in ingresso.

Nel caso in cui venga stabilita correttamente una nuova connessione viene letto dalla socket un messaggio proveniente dal client con il quale contenente la richiesta dell'inizio del flusso di dati della presentazione video.

```
…
JoinCommand cmd = (JoinCommand) inps.readObject();
…
```

Anche la chiamata al metodo readObject() sullo stream in ingresso è di tipo sospensivo.

Ipotizzando che questo proxy non abbia ancora raggiunto il livello di soglia di carico, viene eseguito il metodo *beginStreaming()* e lanciato un evento di tipo *ConnectionEvent* che notifica al *ProxyAgent* che è stata attiva una nuove connessione.

Il metodo *beginStreaming()* prevede due comportamenti differenti a seconda del fatto che questa sia la prima connessione su questo proxy oppure una connessione successiva. Viene effettuato un controllo sull'oggetto *dsource*, di tipo *DataSource*; se questo risulta nullo allora significa che si è in presenza della prima connessione attivata su questa entità, altrimenti è vero il contrario.

```
…
if (dsource == null) {
        //inizializzazione dello stream in ingresso e inizio del
        //flusso verso il client
else {
        //duplicazione del flusso in ingresso e invio verso il client
}
…
```

Se si è nel caso della prima trasmissione, vengono allora invocati i seguenti metodi:

```
…
SessionAddress myRTPInAddress = streamReceiver
.getMyInitializedSessionAddress(out);
remoteRTPInAddress = outProtocol.joinTx(myRTPInAddress,
                                cmd.getPresentationMetadata())
streamReceiver.finishInit(remoteRTPInAddress, outProtocol);
dsource = outProtocol.beginTx();
…
```

Questi sono i metodi che permettono l'inizio della ricezione del flusso proveniente dal server.

A questo punto è interessante notare come venga sfruttata un'importante proprietà dei *DataSource* presentati nel Java Media Framework: si tratta della possibilità di creare dei *DataSource* clonabili, e duplicarli per utilizzarli contemporaneamente in più modalità. Vengono quindi creati due *DataSource* clonati, uno che verrà utilizzato per il caching locale ed uno che verrà utilizzato per inviare il flusso al client.

```
…
clone0  = Manager .createCloneableDataSource(dsource);
…
```

L'invio del flusso al client sarà possibile quindi passando il *DataSource* clonato alla classe che si occupa dello stream video.

```
…
returnVal = myVideoAgent.init(
            cmd.getClientSessionAddress(), clone0, out);
…
```

Nel caso in cui invece la connessione con la richiesta di flusso avvenga quando un flusso è già attivo, è possibile semplicemente creare un clone del *DataSource* già utilizzato e inizializzare il *ServerVideoAgent* passandoglielo come argomento.

La stessa cosa si verifica nel caso in cui ci si trovi su un proxy secondario, l'unica differenza sta nel fatto che la richiesta dell'inizio dello stream avviene subito dopo l'inizializzazione del proxy, per cui per tutte le connessioni dei client ci si ritroverà sempre nel caso in cui il *DataSource* in ingresso è già stato inizializzato.

5.2 CREAZIONE DI UN PROXY SECONDARIO

Quando il numero di connessioni su un proxy raggiunge il livello di soglia di carico il sistema deve reagire di conseguenza, passando le nuove connessioni a proxy secondari che non hanno ancora raggiunto questo livello oppure creando nuovi proxy secondari.

Alla raggiungimento della soglia di carico, se non sono presenti proxy disponibili viene inizializzato un nuovo agente *SecondaryProxyInstantiatorAgent* passandogli il nome del place su cui dovrà essere istanziato il nuovo proxy e l'endpoint del proxy corrente. Viene inoltre passato il riferimento all'agente corrente, sotto forma di *SecondaryProxyInstantiationListener*.
A tale scopo è stata creata un'interfaccia chiamata *SecondaryProxyInstantiationListener* implementata dall'agente *ProxyAgent.* Questa interfaccia contiene il metodo *secondaryProxyInstantiation(),* che è quello che sarà chiamato su ProxyAgent al termine dell'istanziamento del proxy secondario. L'evento che esegue questo metodo è *SecondaryProxyInstantiationEvent,* un evento contenente le *ComponentInfo* del nuovo proxy. Finita l'inizializzazione questo *SecondaryProxyInstantiator* viene lanciato.

Questo agente è quello che si assume il compito di inviare un agente mobile sul nodo secondario per lanciare il proxy secondario. Fatto ciò l'agente rimane in attesa di un messaggio dall'agente mobile indicante l'avvenuta inizializzazione del proxy secondario. Infatti, il metodo *run()* di questo agente prevede innanzitutto l'inizializzazione dell'agente *ProxyMobileAgent*, passandogli i parametri ricevuti da *ProxyAgent.* Una volta fatto

questo, questo agente viene lanciato ed inviato sul nodo di destinazione. Questo è possibile utilizzando i seguenti metodi previsti da SOMA:

```
…
AgentWorker proxyMobileAgent = agentSystem
        .getEnvironment().agentManager
        .createAgent("MUM.initService.ProxyMobileAgent",
        args, true, true);
…
proxyMobileAgent.go(placeToGo);
…
```

Segue allora l’attesa di un messaggio segnalante la finita inizializzazione, contenente le *ComponentInfo* del nuovo proxy. Questo avviene sfruttando le possibilità di scambio di messaggi fra agenti offerte da SOMA.

Una volta ricevuto il messaggio viene lanciato l’evento *SecondaryProxyInstantiationEvent*, che sarà ricevuto da *ProxyAgent*, gestito nel metodo *secondaryProxyInstantiation()*, e conterrà le *ComponentInfo* del proxy appena instanziato.
Questo agente quindi si presenta come un agente intermedio, che prevede solo il compito di lanciare un altro agente che andrà a instanziare il proxy secondario, di ricevere il messaggio di conferma di finita inizializzazione e di avvisare il ProxyAgent di ciò.

L’agente *ProxyMobileAgent*, dopo essersi spostato sul nodo di destinazione ha il compito di instanziare il proxy secondario, passargli le informazioni necessarie per collegarsi a quello principale e notificare l’avvenuta connessione all’agente che lo ha generato.

All’interno del metodo *run()* è innanzitutto necessario scaricare sul nodo corrente tutte le componenti necessarie alla creazione di un nuovo proxy. Viene quindi creata la lista di tutti i componenti necessari (vale a dire *Proxy*, *ProxyInProtocolUnit*, *ProxyOutProxotolUnit ServerVideoAgent*, *StreamReceiver* e le loro

relative interfacce implementate) e si interroga l'*AgentSystem* per ottenere il *DownloadManager*.

Una volta ottenuta la conferma del termine del download da parte del *DownloadManager* viene inizializzato il componente *Proxy* passandogli l'endpoint del proxy principale.

Non appena giungeranno le nuove connessioni a questo proxy secondario sarà possibile richiedere un solo stream dal proxy principale con il quale gestirle tutte, impegnando solamente la CPU del calcolatore su cui è eseguito questo proxy secondario.

A questo punto può essere lanciato l'agente *SecondaryProxyAgent* che è effettivamente l'agente corrispondente al proxy secondario. A questo nuovo *SecondaryProxyAgent* viene passato il riferimento relativo al componente *Proxy* poiché così potrà legarsi ai componenti che gestiscono l'effettiva sessione dello stream video e della comunicazioni.

E' ora necessario rilevare l'endpoint del nuovo proxy istanziato. Questo può essere fatto chiamando il metodo *getComponentInfo()* sull'entità di comunicazione Proxy.

```
…
proxyInfo = secondaryProxySession.getComponentInfo();
...
```

Salvato questo indirizzo, è ora possibile creare un messaggio, da inviare al *SecondaryProxyInstantiatorAgent*.

```
…
msg = new Message(proxyInfo, getID(), creatorID);
agentSystem.sendMessage(msg);
...
```

Quindi, *SecondaryProxyInstantiatorAgent* notificherà il *ProxyAgent* principale della riuscita terminazione dell'inizializzazione del nuovo proxy.

Il *ProxyAgent* principale può ora inviare l'endpoint del nuovo proxy alla sua *ProxyInProtocolUnit.*

A questo punto il Proxy principale, sapendo che ha raggiunto il livello di numero massimo di connessioni disponibili, e conoscendo l'endpoint del proxy secondario, è in grado di gestire correttamente la connessione di un nuovo client. Questo infatti avviene a livello della ProxyInProtocolUnit, dopo che è stata effettuata l'*accept()* di una nuova connessione.

La gestione dell'arrivo di un nuovo client è stata già spiegata nell'analisi delle problematiche affrontate nella progettazione. E' quindi necessario comunicare al client l'endpoint del nuovo proxy e quindi questo vi si collegherà autonomamente. Dopo l'*accept(),* infatti viene scritto sulla socket stabilita con il client un oggetto serializzabile, più precisamente un instanza della classe *RedirectToProxy*.

Questa classe è stata creata appositamente con l'intento di rendere l'astrazione di un messaggio. Questo messaggio è l'indirizzo, espresso come *ComponentInfo,* del proxy secondario.

CAPITOLO 6

TEST DEL SISTEMA

In questo capitolo ci accingiamo a presentare la fase finale di sviluppo del progetto, vale a dire quella dei test del sistema realizzato. Si è deciso di dedicare l'attenzione all'analisi del comportamento dei calcolatori che ospitano le entità proxy al crescere del numero di connessioni attive in ricezione del flusso video.

I risultati pervenuti saranno necessari a stabilire il livello di soglia di carico oltre il quale ciascun nodo proxy non può andare, poiché questo superamento prevederebbe la non possibilità di soddisfacimento delle caratteristiche di qualità di servizio richieste dai clienti.

Sarà definita una specifica configurazione di sistema su cui verranno effettuati questi test e verranno presentate le opportune conclusioni che si concretizzeranno nella scelta del livello di soglia raggiunto il quale un proxy dovrà richiedere lo spostamento delle nuove connessioni entranti su un altro nodo, oppure, nel caso in cui non siano presenti proxy disponibili, sarà necessario l'istaziamento di un nuovo proxy secondario.

6.1 CONFIGURAZIONE DI SISTEMA

Per poter effettuare i test appena anticipati è stato necessario stabilire una topologia sulla quale creare le entità necessarie alla corretta fruizione del servizio.

Sono state utilizzati cinque calcolatori Sun {INSERIRE CARATTERISTICHE MODELLI} ; su ognuno di essi era presente come sistema operativo Solaris, una particolare implementazione di Unix. Queste macchine sono state scelte poiché la loro architettura è particolarmente efficiente per la realizzazione di applicazioni di rete, infatti sono spesso utilizzate per svolgere la funzione di server o proxy in sistemi professionali.

Su uno di questi calcolatori è stato deciso di lanciare l'entità server, su un'altra quella proxy mentre gli altri tre sono stati impiegati per il lancio di un elevato numero di client.

Nella figura seguente mostriamo graficamente la topologia corrispondente alla configurazione utilizzata:

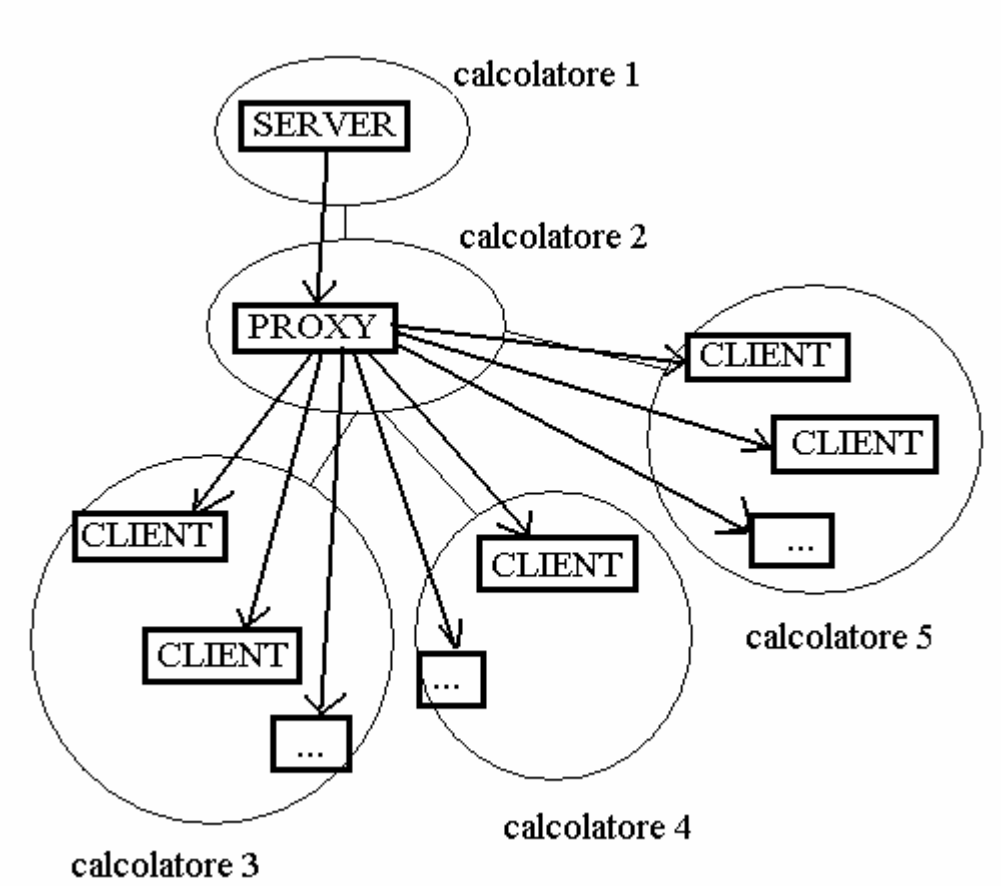

Fig 6.1 Topologia utilizzata per i test.

La scelta di utilizzare tre calcolatori separati per il lancio dei client è legata al fatto che la gestione di molte entità di questo tipo che ricevono flussi video sullo stesso calcolatore è molto onerosa, ed è stato verificato che ciascuno di questi calcolatori raggiunge la soglia critica molto prima di quello sul quale è presente il proxy. In una normale implementazione questo sistema prevederebbe la presenza di un solo client su ogni computer, ma è ovviamente impensabile avere a disposizione un numero così elevato di calcolatori per test di questo tipo. L'utilizzo di tre calcolatori separati su cui lanciare i client si è verificato sufficiente.

Su ognuno di questi calcolatori è stato lanciato un apposito agente che simula ciclicamente la richiesta della medesima presentazione e dell'inizio dello streaming.

I dati che si è scelto di analizzare durante il test sono stati i seguenti:

- Sul nodo proxy è stato lanciato ciclicamente un apposito comando Unix che permette, in riferimento al processo "java" (cioè quello riguardante l'esecuzione del sistema realizzato), il rilevamento del consumo attuale percentuale della CPU, della quantità percentuale di memoria occupata e della quantità effettiva di memoria occupata. Il comando utilizzato è il seguente: *"ps -eLP -o args,pcpu,pmem,vsz"* ; esso è stato lanciato ogni secondo e sono stati salvati i risultati.
- Sul nodo client invece è stato effettuato il salvataggio di un log contenente il tempo di inizializzazione di ogni client, effettuando la differenza fra l'istante della richiesta da parte dell'utente e quello della fine dell'inizializzazione.

6.2 RISULTATI DEI TEST

Una volta decisa la topologia e il tipo di test da effettuare si è iniziato a mettere sotto stress l'architettura, lanciando ciclicamente i client sui tre calcolatori disponibili.

I test purtroppo non è stato possibile concluderli interamente poiché, raggiunto un elevato numero di connessioni attive contemporaneamente sul proxy, è stata generata un'eccezione che ha impedito la corretta continuazione del lancio dei client. La causa di questa eccezione è stata l'impossibilità di aprire più di un certo numero prefissato di socket su ogni sistema operativo Unix. Purtroppo non è stato sufficiente il tempo per ulteriori test dopo aver alzato il numero di socket lasciate disponibili, ma osservando i dati ottenuti è stato possibile azzardare un'estrapolazione per poter stabilire un livello di soglia di carico ragionevole. Questo poiché l'andamento dei dati ottenuti si è rilevato regolare e lineare.

Nelle seguenti figure mostriamo l'andamento dei dati rilevati:

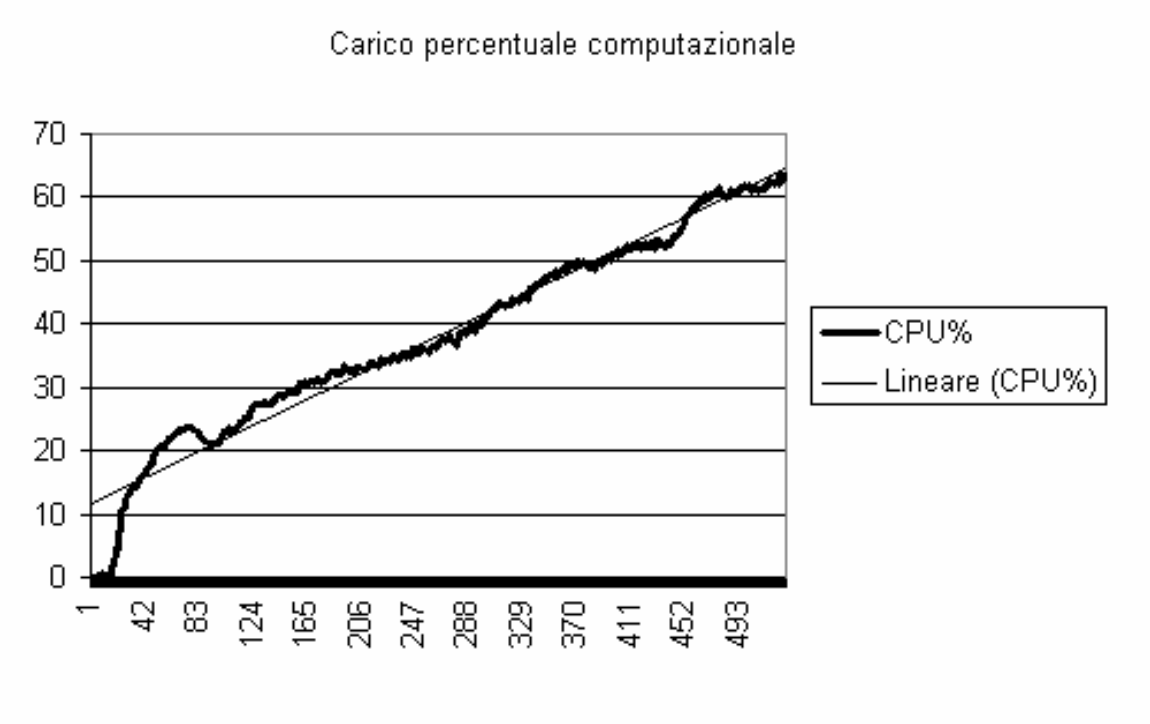

Fig 6.2: Grafico relativo al carico computazionale espresso in percentuale in funzione del tempo.

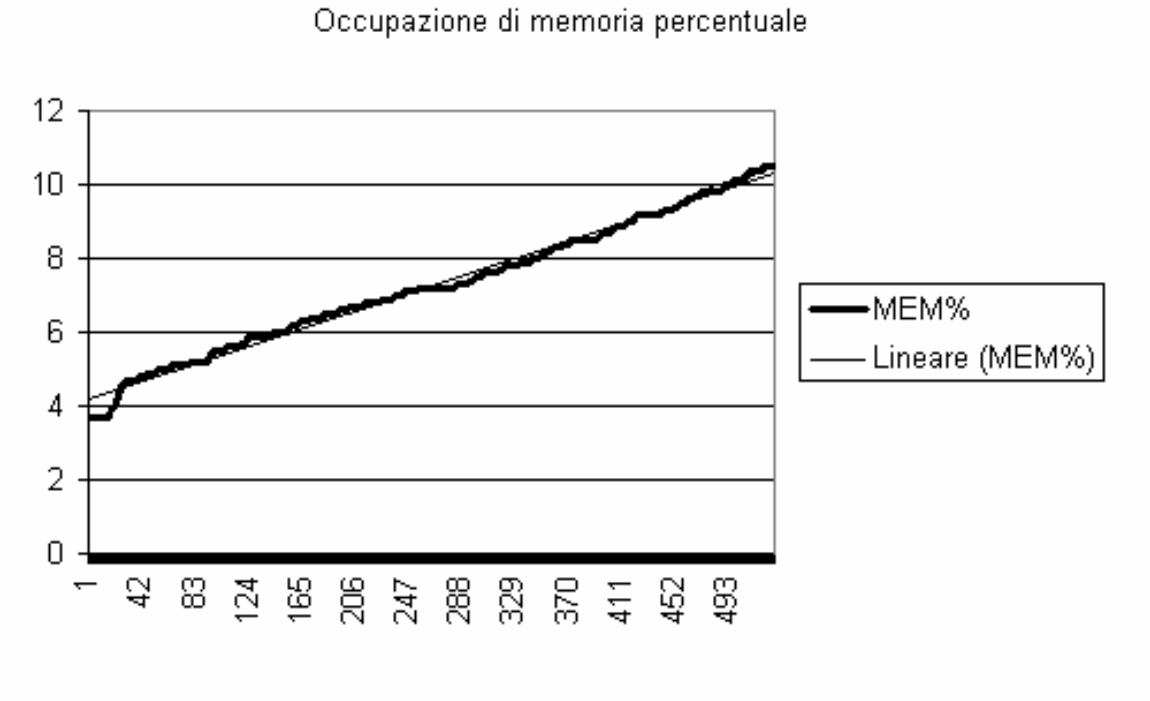

Fig 6.3: Grafico relativo all'occupazione di memoria percentuale in funzione del tempo.

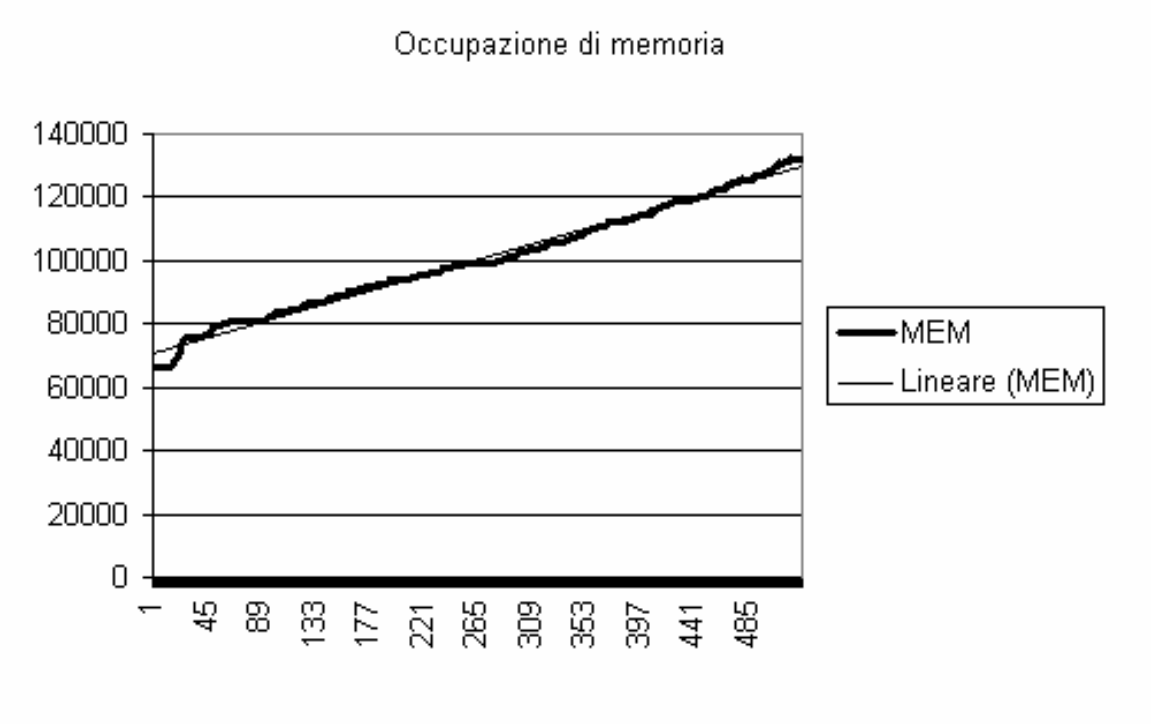

Fig 6.4: Grafico relativo all'occupazione di memoria effettiva rispetto al tempo.

Fig 6.5 Grafico relativo al tempo di inizializzazione dei client.

Dai grafici riportati è possibile trarre alcune conclusioni importanti. Innanzitutto è possibile notare come l'incremento dell'impiego della CPU sia di tipo lineare, per cui è possibile

supporre che, poiché a seguito della connessione di 80 client il carico percentuale è del 65%, allora il carico dovrebbe raggiungere il 100% con la connessione di circa 120-130 client (questo dato è da prendere come valore strettamente indicativo).

E’ interessante notare inoltre come anche i valori di occupazione della memoria crescano in modo percentuale, ma si mantengano a valori percentuali relativamente bassi rispetto al consumo della CPU. E’ quindi corretto il considerare il calcolo computazionale come principale indice del livello della soglia di carico per i calcolatori analizzati.

Il tempo di inizializzaizone dei client invece, segue un andamento decisamente meno lineare rispetto ai grafici precedentemente analizzati. E’ comunque giustificabile il picco iniziale, dovuto alla prima inizializzazione del sistema. Questa comporta infatti anche la creazione delle entità proxy e server e l’inizio del flusso fra essi, non solo l’inizializzazione del client come avviene nei casi successivi. Nonostante i tempi siano abbastanza vari è però possibile osservare il graduale innalzamento di questi tempi in funzione del numero di nuove connessioni.

Sarebbe stato interessante osservare l’andamento del tempo di inizializzazione dei client a seguito del raggiungimento della soglia del 100% della CPU del calcolatore, poiché probabilmente in questo caso si sarebbe dovuto avere un sostanziale innalzamento di questi valori.

E’ importante notare però, che il carico percentuale della CPU rilevato è quello relativo al solo processo relativo al progetto, non al carico generico della CPU. Quindi bisogna tenere conto del fatto che sul calcolatore sui cui è presente il proxy potrebbero essere presenti altri processi in esecuzione, i quali potrebbero contribuire a portare il livello computazionale a saturazione ben prima del raggiungimento del 100% rilevato con test simili a questi. Nel nostro caso ci siamo accertati precedentemente che non fossero presenti parametri di disturbo di questo tipo.

6.3 CONCLUSIONI

In conclusione, i test effettuati hanno permesso di verificare che il parametro dell'impegno computazionale è il parametro più importante da tenere in osservazione per monitorare il livello di soglia, in quanto è quello che aumenta più velocemente.

Nel nostro caso è possibile ipotizzare che, in assenza di altri processi attivi che occupino intensivamente la CPU, il livello di soglia possa essere impostato attorno a 100-110 connessioni, per mantenere la possibilità di una certa variabilità in favore del costante mantenimento delle caratteristiche qualitative del servizio offerto agli utilizzatori.

Potrebbe essere necessario in future implementazioni progettare un meccanismo di riconoscimento dinamico del livello di soglia, in modo che sia il proxy stesso a valutare istante per istante se è il caso di attribuire ad un'altra entità parte del proprio carico. Questa soluzione sarebbe anche indipendente dal tipo di calcolatore utilizzato e solleverebbe gli amministratori dal compito di verificare questo parametro prima di effettuare la prima esecuzione del sistema sui propri calcolatori.

CONCLUSIONI

Con la realizzazione di questa tesi è stato possibile fare un' analisi generale del funzionamento delle soluzioni di broadcast attraverso la rete, e di ciò che avviene in seguito ad elevate sollecitazioni applicate a sistemi di questo tipo. Da questa analisi è stato possibile trarre delle conclusioni sulle soluzioni da adottare per fornire un servizio di broadcast video dotato di funzionalità di bilanciamento di carico.

In un modello client-server, utilizzando nodi proxy per la gestione della duplicazione del flusso proveniente dal server, si è notato come siano proprio i proxy ad essere le entità più sollecitate nel caso in cui sia presente un grande numero di connessioni attive. Si è osservato come l'impegno più rilevante dei nodi di questo tipo consista nell'impiego di risorse computazionali per l'adattamento e il processing continuo di tutti i singoli flussi diretti ai client, ancor più che l'utilizzo di memoria o di banda trasmissiva. La valutazione del comportamento di queste entità sotto sollecitazione è stato verificato mediante appositi test che hanno permesso di valutare il livello di soglia di carico per le macchine utilizzate, passata la quale non sono più in grado di fornire un servizio che garantisca una sufficiente qualità di servizio.

La soluzione adottata per migliorare le prestazioni del sistema è stata quella di creare nuove entità proxy in grado di prendersi la responsabilità di gestire le nuove connessioni in arrivo. E' stata fornita a queste entità la possibilità di comunicare fra loro per mantenere la conoscenza dello stato di ognuna di esse e quindi fornire un corretto bilanciamento delle connessioni. In questo modo si ha un frazionamento del carico proporzionale al numero di entità create. Il numero di connessioni che l'architettura studiata è in grado di fornire, quindi, non è legato solo alla potenza di calcolo delle singole macchine, ma anche alle caratteristiche della rete locale presente nella località sulla quale sono presenti i proxy. Una rete con a disposizione molti calcolatori potrà di conseguenza mantenere più connessioni di una con meno calcolatori, a parità di potenza computazionale.

Il modello ad agenti si è dimostrato versatile ed adeguato al tipo di sistema creato, poiché la mobilità del codice viene utilizzata per la creazione dinamica di nuovi proxy, in grado di disporsi automaticamente adattandosi alle caratteristiche della rete presente. E' stato quindi utile avvalersi di un sistema di gestione degli agenti come SOMA, e delle entità e dei meccanismi di inizializzazione forniti da MUM.

Il sistema di gestione delle interazioni fra i proxy creati potrà essere migliorato implementando sistemi di comunicazione più raffinati, gestendo in maniera migliore i problemi dovuti alle possibili cadute delle entità sui nodi e potrà essere inoltre migliorata la gestione del deploy su topologie molto complesse. Sarebbe opportuno, utilizzando funzioni offerte dalle API di Java, introdurre un meccanismo di riconoscimento automatico del carico del calcolatore su cui è eseguita un'entità proxy, per monitorare lo stato effettivo di parametri come l'impegno della CPU, la banda e la memoria occupata. Questo eviterebbe la fase di test per la calibrazione del sistema prima della prima inizializzazione del servizio.

BIBLIOGRAFIA