Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea in Informatica

Un'applicazione VoIP per BlackBerry

Tesi di Laurea in Architettura dei calcolatori

Relatore:
Dott.
Vittorio Ghini

Presentata da:
Luca Salucci

Sessione II
Anno Accademico 2010/2011

Indice

Introduzione

Negli ultimi anni la tecnologia VoIP è passata dall'essere una tecnologia di nicchia ad essere conosciuta e quotidianamente utilizzata da un numero sempre crescente di utenti in ambito sia privato che lavorativo; la grande diffusione di questa tecnologia è stata in larga parte dovuta alla diffusione e al successo di software commerciali VoIP come *Skype*^{TM}. Il termine VoIP non indica un singolo protocollo di comunicazione o un singolo standard, bensì un insieme di tecnologie, metodologie, protocolli di comunicazione e tecniche di trasmissione che permette la comunicazione vocale, e video, tramite reti IP, come Internet. Portare un servizio di tipo telefonico, a cui noi tutti siamo abituati, dalla rete tradizionale a commutazione di circuito ad una rete a commutazione di pacchetto basata su IP, come Internet, non è un compito semplice: è infatti caratterizzato da numerose difficoltà, soprattutto in termini di QoS, sicurezza e altre problematiche che nelle reti tradizionali non è necessario tenere in considerazione. Internet infatti non è progettata per garantire nessuna QoS minimale, nè certo per garantire sicurezza.

I primi software VoIP commerciali sono nati a metà degli anni '90 e solo successivamente sono stati standardizzati ufficialmente alcuni dei protocolli che costituiscono l'ossatura delle applicazioni VoIP odierne, come SIP, la cui specifica è stata rilasciata nel 1999. Vediamo quindi che il VoIP risulta essere una tecnologia relativamente giovane; dico relativemente perché 15 anni possono portare a sviluppi e cambiamenti notevoli in ambito informatico. Inizialmente la tecnologia VoIP era disponibile solo su PC, ed aveva quindi

un'utilità limitata; la proliferazione di dispositivi mobili degli ultimi anni e la disponibilità di reti wireless ad ampio raggio (come 3G) ha incrementato notevolmente le potenzialità delle tecnologie VoIP, portandole anche a competere sul mercato con le tecnologie di telefonia cellulare classica. Ormai ogni dispositivo mobile, gli smartphone in particolare, mette a disposizione dei propri utenti diverse applicazioni VoIP, fornite direttamente dai produttori stessi dei dispositivi oppure costruite sulla base delle API *(Application Programming Interfaces)* messe a disposizione dagli stessi.

Questa tesi descrive per l'appunto il lavoro che è stato compiuto per contribuire allo sviluppo di un'applicazione VoIP open-source su piattaforma *BlackBerry*; contribuire perché questo lavoro è già stato avviato [1] ed io mi accingo a proseguirlo da dove era stato lasciato.

RIM, l'azienda propietaria della piattaforma *BlackBerry*, non è entrata in competizione nel mercato delle applicazioni VoIP e non ha inserito nella piattaforma alcune delle componenti fondamentali per lo sviluppo di un'applicazione VoIP; il mio collega Fabio Pini si è quindi dovuto occupare di sopperire a queste mancanze, che descriveremo nel dettaglio successivamente.

Un'altro obiettivo di questo progetto è di realizzare nella nostra applicazione un modello di architettura di rete denominato ABPS *(Always Best Packet Switching)*; tale modello cerca di sfruttare le potenzialità dei dispositivi che presentano interfacce di rete multiple, utilizzandole simultaneamente. Il modello normalmente adottato, denominato ABC *(Always Best Connected)* prevede infatti che venga utilizzata una sola interfaccia per volta, quella che presenta la migliore connessione, la quale viene abbandonata nel caso di perdita della connessione o di degrado delle prestazioni di rete al di sotto una certa soglia. Il modello ABPS prevede un modello di architettura più avanzato che permetta di inviare ogni singolo pacchetto su una diversa interfaccia

e l'interfaccia scelta potrà essere stabilita sulla base di parametri differenti.

Questa tesi è suddivisa in quattro capitoli; il primo capitolo presenta una panoramica sul VoIP e su alcuni dei protocolli utilizzati per realizzare applicativi VoIP; nel secondo viene illustrato ciò che abbiamo a disposizione per la realizzazione del nostro applicativo. Nella parte iniziale del capitolo è stato illustrato il modello ABPS e l'architettura progettata per realizzare tale modello, si prosegue poi con una panoramica sulla piattaforma *BlackBerry* e su cosa viene messo a disposizione dalle API *BlackBerry* andando poi a vedere come si è sopperito alle carenze di queste API. Non essendo infatti presente un supporto al VoIP nativo è stato necessario costruire dal basso tale supporto verificando cosa fosse già presente e cosa mancasse, realizzando prima il supporto per le interfacce di rete e poi cercando il supporto per RTP e SIP, eseguendo il porting su *BlackBerry* di alcune librerie esterne, come MjSip. La costruzione di questo supporto per il VoIP è l'argomento principale della tesi e del lavoro di Fabio Pini; nel caso in cui si voglia approfondire e vedere con maggior dettaglio tali questioni rimando alla tesi stessa [1]. Nel terzo capitolo viene trattato il vero e proprio lavoro da me svolto ovvero la progettazione e l'implementazione dello streaming audio a livello applicativo, basandomi anche su suggerimenti ed indicazioni forniti dal mio predecessore. Nel momento in cui ho iniziato il mio lavoro ho effettutato una ricerca preliminare per verificare se esistessero già applicazioni VoIP open-source serie per *BlackBerry*, senza però trovare nulla di interessante; ho dovuto quindi iniziare il lavoro senza avere la possibilità di confrontarmi con implementazioni funzionanti preesistenti. Giunto ormai alla conclusione del mio lavoro sono venuto a conoscenza dell'esistenza di un'applicazione VoIP open source e cross-platform che prevede anche una versione per *BlackBerry*. Nel quarto ed ultimo capitolo viene quindi brevemente presentata tale applicazione, Linphone [2], alla quale ho fatto riferimento per confrontarmi riguardo ad alcuni problemi incontrati e che potrebbe essere tenuta in considerazione in futuro per eventuali confronti nelle prossime fasi di sviluppo

dell'applicazione.

Il codice e tutta la documentazione relativa sono a disposizione di chiunque voglia dare il suo contributo per la prosecuzione del progetto e il raggiungimento degli obiettivi sopra illustrati.

Capitolo 1

VoIP - Voice Over IP

1.1 Una panoramica introduttiva

Il termine VoIP indica un insieme di protocolli che permettono di effettuare conversazioni telefoniche attraverso reti che utilizzano il protocollo IP, come Internet. Questa tecnologia sta conoscendo una diffusione sempre maggiore e viene ormai quotidianamente utilizzata, oltre che da aziende e grandi società di telecomunicazioni, anche da un numero sempre crescente di utenti privati. La grande diffusione di questa tecnologia sta venendo enormemente stimolata anche, e sopratutto, dalla diffusione sempre più capillare di dispositivi mobili, quali netbook, palmari, smartphone e tablet, che ha conosciuto una vera e propria esplosione negli ultimi anni. Tali dispositivi permettono di rendere questa tecnologia estremamente versatile e l'hanno liberata dalla relegazione al solo ambito desktop, portandola sulle piattaforme più disparate.

Attualmente il VoIP non è ancora uno standard definito, anche se diversi enti internazionali, tra cui l'IETF, stanno lavorando da tempo a tale proposito. La possibilità di effettuare comunicazioni vocali tramite Internet nasce già nel 1995, con l'arrivo dei primi software per chiamate vocali tramite un PC collegato ad Internet (ad es. Internet Phone Software). Da quel momento in poi gruppi commerciali e compagnie telefoniche hanno cominciato

a sviluppare tale tecnologia utilizzando la suite di protocolli H.323. Nel 1999 avviene una svolta, quando, grazie all'IETF nasce il protocollo SIP (*Session Initiation Protocol*) [3], che rimpiazza H.323 e diventa standard de facto per lo sviluppo di applicazioni VoIP. SIP, che descriveremo con maggior dettaglio più avanti, è un semplice protocollo di signaling che ha il grande vantaggio di essere compatibile con diversi altri protocolli di livello applicativo e di trasporto. Intorno ai primi anni 2000 nascono provider VoIP commerciali (ad es. Skype o Vonage) che diffondono l'utilizzo ad utenti privati, anche grazie alla possibilità di effettuare chiamate verso la rete tradizionale PSTN a tariffe agevolate.
I vantaggi del VoIP sono per lo più riassumibili in termini di:

- **Costi ridotti**, per diversi fruitori:
  - singoli utenti, con la possibilità di non avere più tariffazioni proporzionali soltanto alla durata delle chiamate;
  - compagnie telefoniche, rispetto alla gestione delle centrali a commutazione;
  - aziende, che possono sfruttare l'accesso ad Internet non solo per la trasmissione dati, ma anche per le comunicazioni vocali.
- **Nessuna necessità di nuove infrastrutture**, come nuove linee telefoniche, in quanto si usa lo stesso canale fisico di rete PSTN.
- **Servizi aggiuntivi**, con la possibilità di utilizzare tutti i benefici che una infrastruttura come Internet può offrire: videochiamate, videoconferenze, trasmissione di allegati, ecc...

Le problematiche pricipali inerenti il VoIP, d'altro canto, sono per lo più due:

- **Tutela della sicurezza:** questo problema, e le soluzioni proposte, rappresentano una parte importante e vasta dell'argomento, che noi però non approfondiremo, ma dobbiamo quantomeno accennare. In gran

parte i problemi di sicurezza sono dovuti alla mancanza di sicurezza intrinseca nell'infrastruttura che viene utilizzata. Internet infatti non è stata progettata incorporando un principio di sicurezza al suo interno, quindi una trasmissione vocale che viaggia su tale mezzo può subire attacchi, proprio come qualunque altra trasmissione dati, dato che il canale fisico è condiviso e non protetto. Per tentare di risolvere questi problemi si utilizzano strumenti come la crittografia, per mantenere le informazioni segrete e visibili solo agli attori della comunicazione, e per garantire l'autenticazione degli utenti.

- **Quality of Service:** con il passaggio da PSTN a rete IP, nascono una serie di problemi riguardanti l'integrità e la perdita dei dati ed il tempo di latenza. Occorre infatti che un'applicazione rispetti certi parametri per fare sí che la conversazione risulti fluida e comprensibile; per realizzare ciò possono essere adottate specifiche politiche di reinvio dei pacchetti persi o di selezione di quelli in ritardo.

I parametri cruciali al fine di ottenere una conversazione di qualità accettabile sono:

- **la latenza**, ovvero il tempo di percorrenza del pacchetto da un end all'altro: secondo *ITU-T* [4] una latenza accettabile deve mantenersi inferiore ai 150ms al fine di mantenere interattività sufficiente;

- **la percentuale dei pacchetti persi**, possibilmente non superiore al 10

- **il consumo di banda**, che deve essere garantito dai codec audio utilizzati nella comunicazione;

- **il Jitter** il ritardo tra due pacchetti giunti all'end uno dopo l'altro, dato dalla differenza di latenza. Se tale parametro risultasse troppo alto (per esempio il valore assoluto della differenza fosse maggiore di 20 ms) le prestazioni degraderebbero e la voce risulterebbe spezzettata.

1.2 I protocolli utilizzati

Descriviamo ora una serie di protocolli impiegati per l'implementazione di applicazioni VoIP e che quindi risulteranno utili nello sviluppo del nostro progetto di applicazione VoIP su piattaforma *BlackBerry*. A seconda delle specifiche necessità e del contesto delle applicazioni in questione, tali protocolli possono rivelarsi più o meno adattti alla risoluzione delle problematiche che ci troveremo ad affrontare. Il nostro intento è quello di fornire nozioni di base a riduardo e di offrire una panoramica che ci permetta succcessivamente di comprendere meglio le soluzioni adottate. L'ordine in cui sono presentati è dal più basso al più alto, facendo riferimento ai livelli dell'architettura di rete TCP/IP.

1.2.1 UDP - User Datagram Protocol

UDP [5] è un protocollo di livello trasporto, che si affida al sottolivello IP per la trasmissione e la ricezione di pacchetti, detti datagram. A differenza di TCP(*Transmission Control Protocol*) [6], un altro protocollo di livello trasporto, UDP è connectionless, ovvero orientato alla trasmissione, e non garantisce una trasmissione affidabile dei pacchetti e nemmeno il loro ordinamento. A fronte di questi svantaggi, UDP è compatibile e comunemente usato anche per applicazioni che effettuano trasmissioni broadcast e multicast. La sua caratteristica più importante è che l'overhead dei pacchetti risulta minimo, rendendolo adatto per trasmissioni dati come quelle VoIP, in cui si inviano grandi quantità di pacchetti audio. Inoltre la perdita di alcuni pacchetti è tollerabile, seppur entro certi limiti al fine di garantire QoS.

L'header UDP è costituito da un totale di 64 bit, dei quali 32 sono dedicati alle porte sorgente e destinazione del pacchetto, 16 per la lunghezza del messaggio (in bytes) e 16 di checksum per il controllo degli errori.

1.2.2 SIP - Session Initiation Protocol

il Session Initiation Protocol è fondamentale per le comunicazioni VoIP, e non solo. Esso è costruito al di sopra del livello di trasporto, in particolare si trova al livello di sessione, e consente uno scambio di messaggi testuali per stabilire i parametri di base della comunicazione, come gli indirizzi IP, il protocollo utilizzato a livello di trasporto e i codec audio/video impiegati nella trasmissione. L'handshake avviene attraverso richieste da parte di uno UA (user agent), un'entità logica che può fungere sia da client che da server. Esempi di richieste SIP sono:

- **INVITE**: per invitare un utente in una sessione di chiamata;
- **ACK**: notifica l'accettazione della chiamata;
- **BYE**: chiude una sessione di chiamata;
- **CANCEL**: per terminare una sessione se non ancora iniziata;
- **REGISTER**: utilizzato per richiedere ad un *Registrar Server* di mantenere il proprio indirizzo memorizzato in un certo dominio per un periodo di tempo specificato nel quale lo UA può essere rintracciato all'indirizzo specificato. Per conoscerlo è sufficiente accedere al *location service* del dominio stesso.

Altri messaggi di request meno usati, sono impiegati per gestire funzionalità diverse da quelle relative a sessioni di chiamata audio (come ad esempio MESSAGE, per inviare messaggi istantanei).

Le risposte a tali richieste sono messaggi comprensivi di un codice a tre cifre indicante il tipo di risposta, in stile HTTP. Le classi di risposta possibili sono le seguenti:

- **100-199**: Risposta provvisoria, in attesa di quella definitiva. Esempio tipico è 180, codice di *Ringing* che indica che la richiesta è stata effettuata e si attende la risposta dall'altro user agent;

- **200-299**: Risposta definitiva e positiva da parte dell'altro UA che ha accettato la chiamata;
- **300-399**: *Forwarding* della richiesta ad un UA diverso da quello a cui era predestinata la richiesta;
- **400-499**: Errore da parte di chi ha fatto la richiesta;
- **500-599**: Errore da parte dello UA che deve rispondere;
- **600-699**: Errore globale.

La struttura è però simile per tutte le tipologie di messaggi, formati da headers testuali che identificano precise informazioni. I più importanti sono:

- **Via**: indica il protocollo usato a livello di trasporto, e la *request route*, ossia tutti i *proxy* che hanno effettuato il forward della richiesta;
- **From**: l'indirizzo di chi ha inviato il messaggio (da specifiche RFC *sip:username@domain*);
- **To**: l'indirizzo del destinatario del messaggio;
- **Call-Id**: identificatore della chiamata che include anche l'indirizzo dell'host;
- **Contact**: mostra uno o più indirizzi SIP presso cui reperire l'utente;
- **User Agent**: il nome dello UA che ha iniziato la comunicazione;
- **Content-Length**: la dimensione in byte del contenuto del messaggio.

Facciamo notare che le informazioni riguardanti la sessione audio/video multimediale sono descritte da header differenti che seguono un altro protocollo testuale, SDP (*Session Description Protocol*) [7]. I principali sono:

- **v=** versione del protocollo;

- **o=** proprietario e identificatore della sessione;
- **s=** nome della sessione;
- **c=** informazioni di connessione;
- **t=** tempo passato dall'inizio della sessione;
- **m=** media usato (ad es. RTP) per i dati multimediali;
- **a=** attributi del media usato (ad es. il codec audio/video usato).

1.2.3 RTP - Real-Time Protocol

RTP [8] è un protocollo utilizzato nell'ambito di applicazioni multimediali real-time ed è impiegato per la gestione dei pacchetti contenenti dati audio/video. Di solito è applicato ai datagram UDP, sopra al livello di trasporto, ed è composto da header testuali, tra cui i più importanti sono:

- **Timestamp**: per effettuare operazioni di sincronizzazione ed eventualmente scartare dati inviati da troppo tempo;
- **Sequence number**: numero di sequenza per identificare se sono stati persi pacchetti e poter implementare politiche di recupero;
- **Payload Type**: codice che identifica la codifica audio o video usata per il payload RTP;
- **SSRC**: il *synchronization source* è un codice scelto casualmente che identifica univocamente la fonte dei pacchetti RTP, all'interno della sessione.

Assieme a RTP può anche essere utilizzato RTSP *(Real-Time Streaming Protocol)*, un protocollo di controllo per gestire da remoto il flusso di dati in streaming, effettuando operazioni come start, stop, seek o pause.

Capitolo 2

ABPS - Always Best Packet Switching

2.1 Il modello e l'architettura

Nell'ambito delle telecomunicazioni tra dispositivi mobili, hanno ampia diffusione dispositivi portatili dotati di tecnologia multi-homing. Parliamo di apparecchi multi-homed per indicare strumenti dotati di NIC *(Network Interface Cards)* eterogenee. Grazie ad esse, i dispositivi possono utilizzare varie interfacce di rete e trasmettere dati attraverso reti wireless con caratteristiche differenti.

Attualmente, i dispositivi multi-homed non sfruttano appieno le loro possibilità utilizzando di volta in volta interfacce di rete adatte al contesto, a causa soprattutto di limiti economici e tecnici. Il modello comunemente adottato è detto ABC (*Always Best Connected*) [9]; tale modello prevede che il dispositivo multi-homed individui la migliore delle interfacce di rete disponibili, e la utilizzi finchè le prestazioni non degradano. Ovviamente questa non è l'unica politica adottabile e sicuramente non è la migliore. Una politica alternativa detta ABPS *(Always Best Packet Switching)* si pone l'obiettivo di creare un modello di architettura più avanzato, che permetta la possi-

bilità di inviare ogni singolo datagram su un'interfaccia di rete differente. La scelta dell'interfaccia di rete adatta potrà dipendere da metriche differenti quali QoS, costi economici, esigenze di piattaforma, disponibilità delle reti, potenza del segnale e quantità di radiazioni elettromagnetiche.

Gli obiettivi *long term* di questo modello sono:

- Limitare la quantità di emissioni elettromagnetiche mediante tecnologie di trasmissione di *range* quanto più limitato possibile (ad esempio WiFi, anziché UMTS).
- Aumentare la disponibilità di banda, favorendo nel contempo la trasmissione su canali paralleli.
- Fornire sufficiente interattività *end-to-end*, prevedendo politiche di controllo e reinvio di pacchetti persi (implementate ad attraverso componenti software *Monitor* e *Load Balancer*).
- Utilizzare le strutture wireless già esistenti senza doverle modificare e senza fare affidamento sulla presenza di tecnologie come *IPV6*.
- Superare i limiti imposti dalla presenza di sistemi di protezione come *NAT* e *Firewall* e progettando proposte che non degradino le prestazioni, come accade con alcune soluzioni esistenti allo stato attuale (ad es. sistemi *STUN* o *TURN*).

L'architettura ABPS-SIP/RTP è stata progettata seguendo tre linee guida:

- **Servizio di continuità di comunicazione esterno**
  In figura (Fig. 2.1) possiamo notare la presenza di un FS (*Fixed Server*): si tratta di un server con IP statico al di fuori di ogni sistema firewall o NAT. In sostanza svolge la funzione di un SIP UA che opera in maniera opaca permettendo comunicazione continua tra i due end della comunicazione: MN, mobile node e CN, correspondent Node.

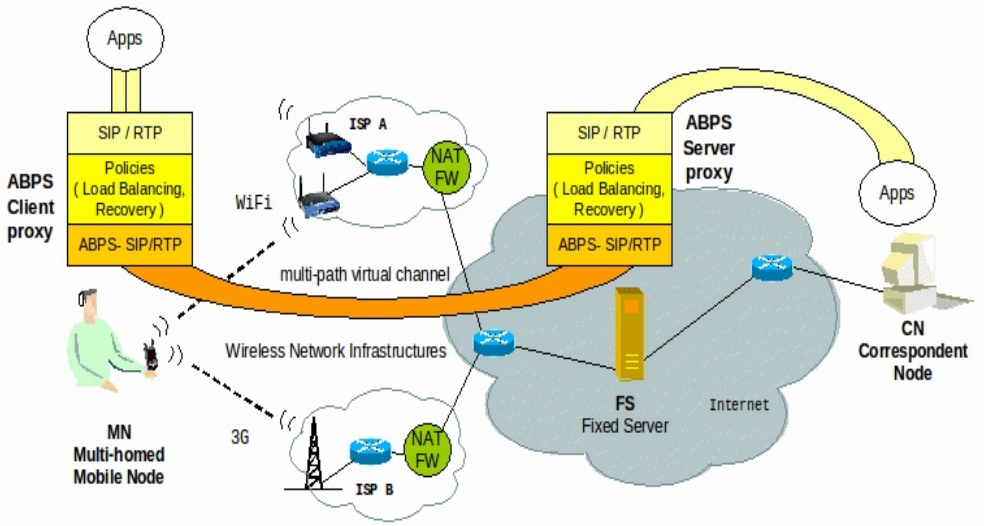

Fig. 2.1: L'architettura ABPS in ambito SIP/RTP

Dal punto di vista del CN, il FS appare come se fosse il MN: infatti il FS, attraverso il server proxy ABPS, riscrive i messaggi SIP nascondendo la locazione corrente del MN. Questo espediente serve per risolvere il problema della presenza di sistemi Firewall e NAT che potrebbero degradare le prestazioni e compromettere la comunicazione, e per fare sì che le applicazioni standard SIP/RTP sul MN possano funzionare correttamente. Inoltre vengono instaurati canali multipli tra il MN e il FS attraverso le diversi reti wireless supportate dal dispositivo multi-homed, mantenute da un *proxy agent*, il cosiddetto *ABPS client*.

- **Uso simultaneo di tutte le NIC**
  L'architettura permette di inviare ogni singolo datagram IP utilizzando una NIC diversa, incoraggiando l'ABPS client ed il server a implementare specifiche politiche di *load balancing* e di reinvio dei pacchetti persi, in modo da massimizzare la banda disponibile e diminuire costi economici e quantità di pacchetti persi, superando così i limiti del modello ABC.

- **Estensioni SIP/RTP**
  L'estensione dell'implementazione classica dei protocolli SIP/RTP è necessaria per ridurre il ritardo introdotto dal cambio continuo dei parametri di comunicazione, causato dalla scelta di una differente interfaccia di rete a seconda delle politiche implementate, come precedentemente spiegato. All'atto pratico le estensioni si traducono in header testuali da aggiungere ai pacchetti SIP/RTP per identificare univocamente il *sender* di un messaggio anche se l'indirizzo IP è cambiato (o per scelta del client stesso, o per effettivi problemi riguardanti la rete wireless utilizzata). In questo modo non occorre un nuovo messaggio SIP per rinegoziare i parametri, ma è sufficiente effettuare lo *switch* del datagram RTP attraverso un'altra NIC disponibile.

Da notare che nella figura è rappresentato un solo server, ma in uno scenario reale l'implementazione prevede più FS distribuiti sul territorio.

Ogni MN può beneficiare dei servizi offerti dall'architettura compiendo i passi che ci accingiamo a descrivere.

Configurazione off-line

Il MN deve effettuare una registrazione off-line per usufruire del servizio di continuità. In questo modo potrà ricevere uno *User Identifier* ed una chiave pre-condivisa con il server ABPS. Questi dati saranno indispensabili per estendere i protocolli SIP/RTP e per fornire un contesto sicuro alla comunicazione.

Configurazione del contesto di sicurezza

Quando il MN configura la NIC e si prepara per la trasmissione, il client ABPS fornisce un contesto temporaneamente sicuro usando una chiave *one-session*, ovvero utilizzata per una sola sessione di comunicazione, costruita a partire dalla chiave pre-condivisa ricevuta dal MN in precedenza. In questo

modo non occorre trasmettere più volte la chiave sul canale, compromettendone la segretezza.

Configurazione e aggiornamento dei percorsi multipli di trasmissione

Il client ABPS inizia la comunicazione inviando un messaggio REGISTER ABPS-SIP, una versione estesa del classico messaggio SIP REGISTER che include l'IP di tutte le NIC funzionanti sul dispositivo mobile. Se il MN cambia la sua configurazione IP, il client ABPS invia un messaggio di UPDATE ABPS-SIP con la lista aggiornata delle NIC. I messaggi SIP/RTP hanno un numero sequenziale usato per scartare i messaggi ricevuti non ordinatamente.

Flusso RTP tra MN e CN

Per iniziare una comunicazione RTP, il MN invia un classico messaggio SIP di tipo INVITE al CN scelto. Il FS esegue il forwarding del messaggio al CN. Prima di fare ciò però, client e server effettuano una serie di operazioni in modo da ricevere sia la response al messaggio SIP che i messaggi RTP:

1. aprono una porta UDP per ricevere i datagram RTP dall'entità successiva (per l'MN il FS, e per il FS il CN);

2. inseriscono il proprio IP come indirizzo del sender nel messaggio SIP (il CN vedrà l'ultimo IP inserito, quello di FS);

3. inseriscono come numero di porta nel messaggio SIP quello della connessione UDP appena creata. Il CN, a questo punto, credendo di comunicare col MN User, invierà la risposta al messaggio SIP e, successivamente, i messaggi RTP direttamente al FS, che poi li instraderà al MN con le opportune estensioni.

   Individuiamo tre percorsi: da MN all'ABPS client usando RTP classico, da ABPS client a FS con SIP/RTP esteso e di nuovo da FS a CN con RTP.

Politiche di QoS per i flussi RTP

Come si può notare in figura, l'ABPS client e l'ABPS server implementano nei sottolivelli politiche di selezione della NIC opportuna, load balancing, *detection* e reinvio dei pacchetti persi, utilizzando le estensioni SIP/RTP. Le politiche vengono implementate andando incontro a specifiche esigenze di QoS: ad esempio, in riferimento alla sezione precedente, nel caso specifico del VoIP è necessario scoprire e reinviare i pacchetti persi, per non superare una quantità superiore al 10% del totale. Ogni applicazione *SIP-based* può scegliere il percorso wireless preferito senza dovere essere modificata, ma semplicemente scegliendo come porta in uscita la porta SIP corrispondente al percorso desiderato.

Scorciatoie per l'aggiornamento del canale multi-percorso

Dopo la fase iniziale di configurazione, il flusso RTP dipenderà dall'implementazione delle specifiche politiche dei sottolivelli di ABPS client e server. Nella gestione classica del livello di sessione, il flusso RTP è fra due nodi con un indirizzo IP fisso mentre con ABPS sappiamo che ciò può non valere, in due casi:

- quando il MN cambia la sua configurazione, senza che l'ABPS client lo abbia deciso (ad esempio nel caso di una rete WiFi, il terminale si associa ad un nuovo *Access Point*).
- se per politiche di sottolivello dell'ABPS client si decide di inviare i pacchetti RTP su una diversa NIC.

In entrambi i casi, anzichè dover effettuare uno scambio dei nuovi parametri di comunicazione tra i due nodi, utilizzando un messaggio di REINVITE, ed introducendo un ritardo non trascurabile (proporzionale al *round trip time* della rete in uso) nella trasmissione, utilizzeremo le estensioni SIP/RTP.

Essenzialmente, in ogni datagram RTP abbiamo tutte le informazioni

per potere continuare la comunicazione attraverso una NIC differente: il FID (*Flow Identifier*) che identifica il flusso RTP tra MN e CN, ed una firma digitale basata sulla chiave one-session stabilita durante la configurazione di chiamata.

L'ABPS server può identificare così il sender e successivamente, con opportune implementazioni a livello di rete, leggere il nuovo indirizzo IP del sender dal campo *source IP address* del datagram IP. Grazie a ciò è possibile effettuare lo *switch* del pacchetto mediante poche informazioni sempre disponibili nel datagram ABPS-RTP.

In aggiunta, al costo di un piccolo *overhead*, il datagram può contenere anche il set completo di indirizzi IP di tutte le NIC del MN per effettuare controlli.

2.2 La piattaforma *BlackBerry*

2.2.1 Cos'è *BlackBerry*?

I dispositivi *BlackBerry* sono *smartphone* prodotti dalla società canadese RIM; lo stesso nome *BlackBerry* è però dato anche all'infrastruttura che sta dietro ai loro servizi e che ne ha decretato il successo commerciale nel corso di questi anni.

I dispositivi mobili *BlackBerry* offrono i servizi disponibili nella maggior parte degli smartphone ad oggi in commercio, tra cui:

- **Connettività** attraverso reti wireless 3G, WiFi (standard IEEE802.11 a/b/g/n) [10], Tri-Band HSDPA e Bluetooth, in differenti combinazioni a seconda dei modelli;
- **Funzionalità GPS**, con applicazioni come *BlackBerry Maps*, per ricevere mappe e avere informazioni su località e luoghi di interesse;

- **Navigazione web** tramite browser proprietario, che consente *streaming* di contenuti audio e video da siti predisposti;
- **Instant Messaging**, con applicazioni come *Windows Live™ Messenger*.

Funzionalità aggiuntive possono variare a seconda dei diversi modelli. Il sistema operativo *BlackBerry OS*, giunto alla versione 6, è proprietario, ma RIM mette a disposizione degli sviluppatori API *open source* JDE (*Java Development Environment*) [11] per creare applicazioni *Java-based* eseguibili sui dispositivi.

Illustriamo ora un quadro generale delle API (release 6.0.0), nonchè degli applicativi software messi a disposizione degli sviluppatori software da RIM per creare e testare le applicazioni.

2.2.2 Lo sviluppo di applicazioni in ambiente *BlackBerry*

BlackBerry JDE API 6.0.0

Le API *BlackBerry* offrono estese funzionalità ad ogni nuova release, cercando di mantenere retrocompatibilità con i servizi presenti nelle versioni più datate. Possiamo suddividere le API in diversi *packages*, di cui alcuni sviluppati direttamente da RIM (categoria *net.rim.**) ed altri importati esternamente, come ad esempio quelli appartenenti alla Java ME (*Micro Edition*) [12] (detta anche J2ME) per applicazioni multimediali e ludiche o quelli *W3C* per la gestione di documenti ipertestuali.

Vediamo ora una vista generale di queste API descritte per categoria funzionale:

- **Application Life Cycle**: contiene strumenti per creare, avviare e terminare le applicazioni. *BlackBerry* OS supporta l'esecuzione parallela

di più thread per applicazione. Esiste però un solo *event thread* principale per ogni applicazione, che può modificare la UI (*User Interface*) del programma, nonchè gestire gli eventi causati dall'utente. Il *lock* degli eventi viene acquisito in un entry point (ad es un metodo main()) con un comando apposito.

Per i thread non event è possibile sincronizzarsi con l'event lock per l'invio di messaggi urgenti all'utente attraverso la UI oppure inviare il messaggio nella *message queue* dell'applicazione.

- **User Interface**: una serie di funzionalità per creare un'interfaccia grafica con la quale l'utente può interagire. Esistono quattro framework diversi disponibili:

  - *BlackBerry* UI API: per creare interfacce grafiche standard ottimizzate ed efficienti, compatibili solo con i dispositivi BB.
  - Mobile Information Device Profile API: package della Java ME usati in applicazioni MIDlets. Utilizzabili su tutti i dispositivi compatibili con la piattafoma Java ME.
  - SVG API: librerie che utilizzano SVG (*Scalable Vector Graphics*), un linguaggio di *markup* per la creazione di primitivi oggetti geometrici come cerchi, poligoni, ed altri ancora. Sono compatibili sia con MIDlets che con applicazioni Java *BlackBerry*.
  - Graphics Utility API: framework basato su OpenVG ed OpenGL, aggiunge funzionalità grafiche alle *BlackBerry* UI API. Disponibile dalla versione 6.

- **Application Integration**: package differenti per potere utilizzare applicazioni *BlackBerry Device Software* come *phone, search, calculator, BlackBerry Maps* attraverso l'uso di Invoke. Altri package servono per avviare il browser *BlackBerry* ed applicazioni di terze parti registrate.

- **PIM - Personal Information Management**: una versione con funzioni estese della J2ME PIM, che permette di manipolare informazioni incluse in liste di indirizzi, calendari, *memopad*, ed altro.

- **Messaging**: include package nativi per l'invio di messaggi SMS, MMS, email, e per le applicazioni, il tutto per interagire con l'utente.

- **Network Connections**: si intendono tutti i package per la gestione dei flussi I/O per le applicazioni. Tra i package più importanti per il nostro progetto abbiamo *javax.microedition.io* che permette di stabilire connessioni di rete di tipo TCP, UDP, HTTP e HTTPS in diverse modalità ed attraverso differenti reti wireless.

  Alcuni package nativi RIM supportano la gestione delle connessioni mettendo a disposizione classi ed interfacce per il controllo delle reti wireless, permettendo di conoscere la potenza del segnale ed altre informazioni che esamineremo meglio nel prossimo capitolo.

  Per tutte le connessioni di rete, comunque, lo schema è fisso e generico, tramite l'uso di un'interfaccia Java di connessione comune che verrà istanziata poi a seconda del protocollo scelto. Questa astrazione gestisce anche l'individuazione di file sul device locale o su un altro esterno, in cui l'URI specificato è la locazione del file stesso sulla memoria del dispositivo, o su una scheda *MicroSD*.

- **CLDC Platform and Utilities**: tutti i package nativi, W3C e standard Java e Java ME per funzionalità legate a lingue, linguaggi di markup e metamarkup (come SVG e XML).

- **Device Characteristics and the *BlackBerry* Infrastructure**: un insieme di API native per avere informazioni riguardo ai device ed interagire con l'infrastruttura BB.

- **Data Storage**: tutte le API per la memorizzazione dei dati. Offrono la possibilità di usare *database* SQLite, creare e gestire file, condividere oggetti a runtime tra più applicazioni (o thread), effettuare backup di dati sensibili.

- **Multimedia**: sono i servizi deputati all'integrazione di audio e video nelle applicazioni e come vedremo più avanti giocheranno anche un ruolo importante nella fase finale del nostro lavoro. Provengono quasi tutti dalla Java ME.

- **Location-Based Services**: è l'insieme dei package che permettono di sfruttare le funzioni GPS dei dispositivi di ultima generazione, con incluse le *features* di ricerca locazioni, visualizzazione mappe e calcolo delle informazioni di viaggio.

- **Security & Cryptography**: una serie di classi ed interfacce native *BlackBerry* che offrono la possibilità di crittare e decrittare dati, con crittazione a chiave pubblica o privata, in modo da rendere sicura la comunicazione attraverso attraverso reti wireless.

Strumenti per sviluppo e testing delle applicazioni

Gli sviluppatori che intendono scrivere applicazioni per i dispositivi *BlackBerry* hanno a disposizione da RIM essenzialmente due scelte: la prima è utilizzare un ambiente di sviluppo specifico creato ad hoc da RIM, il *BlackBerry* JDE(*BlackBerry® Java® Development Environment*).

Permette di creare, importare ed esportare progetti *BlackBerry*, nonchè di compilarli ed eseguirli o effettuarne il *debug* su dispositivi collegati al calcolatore. In alternativa si possono utilizzare degli applicativi che simulano il comportamento dei dispositivi mobili. La compilazione avviene utilizzando il classico *Java Compiler*, verificando nel contempo la compatibilità dell'applicazione con le API *BlackBerry*.

La seconda scelta possibile è quella di utilizzare *Eclipse*, un potente IDE per applicazioni in Java, ed in altri molteplici linguaggi di programmazione come C, C++, Python, PHP, ecc... Eclipse è *plugin extensible* e grazie a questa caratteristica RIM mette a disposizione un plugin per godere, anche in questo ambiente, degli stessi servizi presenti nel suo JDE nativo.

Per l'effettivo debugging delle applicazioni vengono forniti diversi simulatori che riproducono, anche graficamente, le sembianze e le funzionalità dei dispositivi *BlackBerry*. Si noti il fatto che anche i servizi più avanzati, come la navigazione web con il browser interno, e le funzionalità legate all'architettura di rete *BlackBerry*, sono simulate correttamente (seppure in maniera limitata) anche grazie ad altri due simulatori, il *MDS Simulator* e l'*Email simulator*.

Possono essere disponibili anche più simulatori di uno stesso modello di smartphone, differenti a seconda della versione più o meno recente del *BlackBerry* OS. Durante la lavorazione del progetto sono emersi naturalmente alcuni problemi riguardanti i simulatori di alcuni modelli di device che specificheremo nei prossimi capitoli.

In effetti è consigliabile, soprattutto per applicazioni di rete, utilizzare anche veri dispositivi per il testing, in quanto il comportamento con reti wireless reali potrebbe essere differente da quello simulato.

Malgrado ciò, questi strumenti sono stati sufficienti per effettuare il controllo delle applicazioni, e studiarne il comportamento a *runtime* utilizzando la console di output del simulatore fornita su Eclipse dal plugin. Per semplicità, una volta compilata l'applicazione e verificato che sia *fully compliant* con le API *BlackBerry*, è sufficiente avviare il simulatore ed accedere alla voce del Menù principale *Downloads* per trovare le applicazioni create e poterle eseguire.

Il plugin per Eclipse, il JDE , simulatori e la documentazione relativa, sono liberamente scaricabili dal sito web di *BlackBerry*. Il software è utilizzabile solo su sistemi Windows a 32 o 64 bit.

2.3 Componenti realizzative per il modello ABPS

2.3.1 Connessioni UDP su *BlackBerry*

Nell'architettura di rete *BlackBerry* il protocollo UDP non viene considerato: questo perché, data la sua natura connectionless, non è adatto per fornire servizi sicuri e affidabili come quelli supportati attraverso il MDS, in cui infatti le connessioni usate sia da applicazioni che da Browser utilizzano TCP come protocollo *transport level*.

RIM mette comunque a disposizione API per effettuare connessioni UDP attraverso la rete WiFi o la rete wireless supportata dal carrier. Queste API ci hanno permesso di effettuare la prima fase dello sviluppo di un'applicazione VoIP per smartphone *BlackBerry*, verificando l'effettivo supporto necessario a livello di trasporto.

Come già accennato nella descrizione delle API, *BlackBerry* utilizza le librerie della J2ME per creare connessioni di diverso tipo come HTTP, TCP, ed anche UDP. Per farlo si utilizza il metodo *open* della classe *Connector* che prende in input tre parametri (gli ultimi due sono opzionali):

- L'URL a cui effettuare la connessione in forma $scheme : [trgt][parms]$. In generale *scheme* si riferisce al protocollo usato, *trgt* all'indirizzo IP e *parms* ad eventuale parametri di connessione in forma x=y;

- una costante per specificare se la connessione è *read only*, *write only* o *read-write*;
- un parametro booleano per abilitare la possibilità di impostare un timeout sul socket.

Viene ritornata dal metodo open una generica interfaccia *Connection* ed a seconda del *cast* effettuato possiamo scegliere quale interfaccia estesa (specifica per ogni protocollo) possiamo utilizzare. Nel nostro caso avremo, ad esempio:

```
UDPDatagramConnection conn;
conn = (UDPDatagramConnection)Connector.open(
"datagram://host_address");
```

UDPDatagramConnection è un'interfaccia che offre i metodi di base per gestire la connessione UDP, ovvero per creare i datagram (la creazione degli header UDP è automatizzata e dobbiamo occuparci solo del payload), riceverli, ed inviarli. Un datagram è un'istanza della classe *Datagram*, che permette di leggere modificare la dimensione, il payload e l'indirizzo IP di destinazione del datagram.

Facciamo notare che una connessione come quella creata sopra è bloccante ovvero quando si effettua un'operazione di ricezione di un datagram attraverso il metodo *receive*, il thread in cui viene chiamato si blocca e non è in grado di accettare input dall'utente o captare eventi, fino a che non vengono ricevuti dati o non viene lanciata un'eccezione. Per evitare che l'intera applicazione si blocchi completamente è bene quindi che queste operazioni vengano eseguite in thread diversi dal *main event thread*, oppure che vengano creati socket con la possibilità di impostare un timeout.

Inoltre, se si vuole che due end possano comunicare inviando e ricevendo datagram UDP contemporaneamente come nel caso di una conversazione vocale, è necessario che siano utilizzati due thread diversi per la ricezione e la trasmissione dei dati, oppure che sia previsto qualche meccanismo di

sincronizzazione.

Per avere maggiore controllo sulla connessione creata possiamo utilizzare una classe, *DatagramConnectionBase*, al posto della stessa interfaccia *UDPDatagramConnection* che implementa. Essa, oltre ai metodi ereditati, ne possiede altri tra cui *checkForClosed* per controllare se la connessione è stata chiusa oppure *setTimeout* per settare un limite di tempo in ms dopo il quale una chiamata al metodo receive ritorna un eccezione, riattivando in questo modo il thread (utile per evitare blocchi dell'applicazione di lunga durata).

Tra le altre funzionalità, una riguarda la possibilità di utilizzare un *Java Listener*, chiamato *DatagramStatusListener*, per catturare gli eventi riguardanti un datagram inviato e potendo così conoscere informazioni riguardanti la consegna e la ricezione sull'altro end della comunicazione. Questa funzionalità non è stata ancora testata a fondo a causa delle limitazioni dei simulatori *BlackBerry* e della scarsa chiarezza della documentazione associata, ma potrebbe essere utile in futuro per l'implementazione di un componente software che verifichi lo status dei pacchetti inviati e metta in atto politiche di reinvio dei datagram persi.

Ora evidenzieremo alcune note sui parametri opzionali ed aggiuntivi degli URL utilizzati nella creazione della connessione. Anzitutto, in fondo alla stringa dell'URL, è possibile specificare la porta di destinazione se si intende inviare dati, o la porta locale, se si è in ricezione. Se non si specificano altri parametri, la connessione UDP viene lasciata in gestione alla infrastruttura di rete del *carrier*. Occorre fare attenzione, perché non sempre tali operatori forniscono supporto adeguato per la trasmissione di dati UDP sulle reti *long range*, solitamente 3GPP *(Third Generation Partnership Project)* [13], compatibili con i device *BlackBerry*; in alcuni casi le comunicazioni UDP vengono invece bloccate dai carrier per motivi di sicurezza.

Per aprire la connessione tramite WiFi è necessario inserire, in coda all'URL separato da ';', un parametro introdotto da RIM: *interface=wifi*. Sono utilizzabili in aggiunta parametri di estensione di RIM per specificare direttamente l'access point al quale collegarsi, e le eventuali credenziali di autenticazione. Se questi parametri vengono omessi, il dispositivo si affida alle impostazioni predefinite per il WiFi, modificabili direttamente dall'utente nella sezione *network options* di gestione delle connessioni del simulatore.

2.3.2 Monitoraggio delle Interfacce di rete

Per poter implementare un modello ABPS è fondamentale avere la possibilità di monitorare le diverse interfacce del dispositivo e lo stato delle relative connessioni per poter stabilire quale utilizzare in base ad opportuni criteri. A tal fine sono stati utilizzati alcuni strumenti delle API *BlackBerry*, presenti nel package *net.rim.device.api.system*.

Le classi e le interfacce in essopresenti hanno permesso di raggiungere due obiettivi:

- Ottenere informazioni sullo stato delle reti wireless a *runtime* dell'applicazione: ciò grazie alle classi *WLANInfo* per il controllo di reti WiFi e *CoverageInfo* per le restanti.

  WLANInfo è una classe che permette di scoprire se il dispositivo supporta connessioni WiFi, se una connessione ad un access point è avvenuta, e le informazioni che lo riguardano, come SSID, categoria di sicurezza, livello di potenza del segnale, ecc..

  CoverageInfo, analogamente, dà modo di conoscere le WAFs (*Wireless Access Families*), diverse dalla WLAN, supportate dal dispositivo e di sapere se esiste copertura di rete sufficiente per effettuare una connessione dati.

- Costruire particolari *Java listener* che vengono attivati, sempre a runtime, in coincidenza di particolari avvenimenti di rete. In sostanza, per le reti WiFi, il listener attiva un metodo per avvisare l'applicazione se il dispositivo mobile non risulta improvvisamente più associato con un'access point, indicandone anche il motivo (ad esempio si è usciti dalla zona di copertura del segnale radio dell'access point). Un altro metodo viene attivato in caso di riassociazione all'access point.

  Per quanto riguarda invece le altre reti wireless, il listener si basa sulla potenza del segnale per attivare un singolo metodo di *callback* che possiamo utilizzare per prendere decisioni a seconda dei casi. Sono presenti in CoverageInfo anche utili costanti che descrivono il livello minimo di potenza del segnale per effettuare connessioni stabili con i device RIM.

Questi strumenti sono importanti in ottica di implementazione del modello ABPS per gestire nell'applicazione gli *switch* tra una connessione wireless e l'altra, scegliendo in base alle politiche di selezione di rete (per limitare costi, utilizzo di *bandwidth* ed emissioni elettromagnetiche) e di reinvio dei pacchetti (per garantire QoS).

Inoltre assicurano che l'applicazione possa trasmettere e ricevere dati anche se soltanto una delle reti wireless è disponibile in un dato momento.

2.3.3 SIP

Se, come visto nel capitolo precedente, per la parte riguardante il livello di trasporto ci sono bastati solamente i servizi forniti agli sviluppatori da RIM, vedremo che invece per l'implementazione del protocollo *session level* SIP è stato necessario volgere lo sguardo altrove.

A causa di precise scelte commerciali, infatti, RIM non prevede tra le sue API alcun supporto per le applicazioni basate su SIP e di conseguenza per la

maggior parte delle applicazioni VoIP odierne; si è reso quindi necessario implementare il protocollo SIP o portare un'implementazione SIP già esistente su piattaforma *BlackBerry*. Per poter essere portata tale implementazione deve soddisfare certi requisiti:

- *fully compliant* con lo standard RFC 3261;
- Java Based;
- il più possibile indipendente da librerie esterne non presenti tra le API *BlackBerry*;
- open source e ben documentato per poterlo utilizzare, studiare ed adattare alle nostre esigenze.

La scelta è ricaduta su MjSip [14], una libreria scritta in Java e sviluppata dall'Università di Parma in collaborazione con l'università Tor Vergata di Roma, per la creazione di applicazioni basate su SIP. La prima versione di questo software è stata resa disponibile in licenza GPL2 nel 2005. Questa libreria è uno stack SIP completo, contenente non solo l'implementazione dello standard SIP, ma anche una vasta serie di API per poter beneficiare dei servizi senza occuparsi di tutti i dettagli implementativi.

MjSip, oltre a soddisfare i requisiti sopra specificati, con qualche eccezione che illustreremo più avanti, possiede una serie di vantaggi verificati durante lo sviluppo del progetto, e riassumibili in:

- Leggerezza del codice, e possibilità di utilizzo su piattaforme come la nostra, che devono fronteggiare una capacità di calcolo ed autonomia energetica limitata;
- semplicità d'uso e di estensione o modifica delle funzionalità;
- inclusione di API interfacciate con l'utente in maniera identica a JAIN(*Java APIs for Integrated Networks*) API;

- inclusione di semplici implementazioni SIP server e SIP UA.

Ora vediamone una panoramica strutturale.

2.3.4 MjSip: un'architettura a livelli

Seguendo la RFC 3261, MjSip utilizza tre livelli (o strati) per l'implementazione del protocollo SIP: *Transport*, *Transaction* e *Dialog*. Al di sopra di essi stanno API di livello applicativo che sono invece fornite da MjSip, dette di *Call control*.

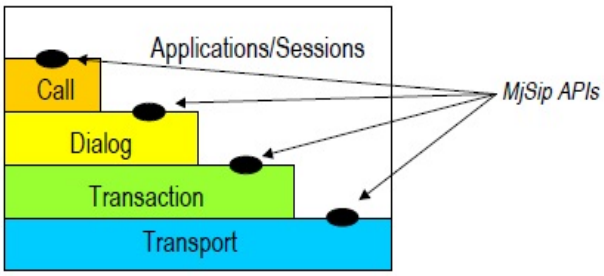

Fig. 2.2: L'architettura a livelli MJSIP

Transport Layer

Al livello inferiore del'architettura troviamo i servizi necessari per il trasporto dei messaggi SIP.

Mediante l'entità *SipProvider*, qualunque elemento MjSip è in grado di ricevere ed inviare messaggi attraverso i protocolli UDP o TCP a livello di trasporto, nonchè di effettuare l'inoltro dei messaggi ricevuti ai livelli superiori.

Quando SipProvider riceve un messaggio da inviare da un'entità superiore, decide quale sia il *next hop* della comunicazione ed effettua il forward

del messaggio attraverso un'opportuna connessione transport level. Per decidere il next hop, si basa sulle informazioni contenute nel messsaggio SIP stesso, controllando determinati header del messaggio: se si tratta di una request, utilizza un *outbound proxy*, se previsto dal *SIP agent*, oppure le informazioni nell'header Route; se quest'ultimo non è presente utilizza gli header *To* e *Contact*. Se invece si ha una response, utilizza l'host specificato nell'header *Via* e il numero di porta in *rport*.

Quando SipProvider riceve un messaggio su una connessione socket, si occupa di processarlo e stabilisce quale entità di livello superiore è destinataria del messaggio, attraverso particolari identificatori, dopodichè lo passa al *listener* dell'entità selezionata.

Sotto al SipProvider troviamo le entità che gestiscono le connessioni TCP e UDP, come *UdpSocket* e *TcpSocket*. Esse utilizzano le API di *java.net* per la creazione dei socket ed entità ausiliarie con funzioni di listener per la notifica eventi particolari (ad esempio la chiusura del socket).

Transaction Level

Secondo le specifiche RFC 3261, una transaction consiste in una singola request e di una o più response ad essa. Sono presenti perciò una componente client che si occupa delle request ed una server che si occupa delle response: in MjSip sono gestite rispettivamente dalle entità *ClientTransaction* e *ServerTransaction*.

Esistono però transaction particolari che non possono essere *two-way* ovvero consistenti in una singola request ed una o più response: ne esistono infatti alcune in cui potrebbe esserci un grande ritardo nella risposta da parte di una entità server. Questo può accadere se occorre un input umano, e ciò accade per l'accettazione di una request di tipo INVITE. Queste transazioni sono dette quindi *three-way*, perché oltre alla request ed alla response, il

client invia anche un ACK al server per fargli sapere che la risposta è stata ricevuta in tempo accettabile e concludere così l'*handshake*.

In MjSip le transazioni three-way per le INVITE request sono gestite dalle entità *InviteTransactionClient* ed *InviteTransactionServer*.

Dialog Level

Il terzo livello è quello di Dialog, che combina diverse transazioni in un'unica sessione. Tale sessione persiste per una certa durata di tempo.

In MjSip è l'entità *InviteDialog*, con l'aiuto di entità ausiliarie, che gestisce la creazione della sessione, univocamente identificata dai due utenti della destinazione e dal valore dell'header *Call-Id* del messaggio SIP. Per creare la sessione, l'entità InviteDialog utilizza le entità di transaction descritte prima. InviteDialog gestisce anche il metodo CANCEL, per annullare una richiesta di INVITE fatta in precedenza.

Call Control Level

Al livello più alto dello stack, troviamo le Call API fornite da MjSip, che forniscono una interfaccia semplificata per gestire una sessione di chiamata SIP, formata da uno o più Dialog. L'entità principale che si occupa di questo compito è *Call*.

2.3.5 Porting di MjSip su BlackBerry

Tra i criteri di scelta di MjSip come stack SIP per lo sviluppo di applicazioni VoIP per *BlackBerry*, abbiamo incluso l'indipendenza da librerie esterne non supportate dalle API *BlackBerry*. Purtroppo, nella pratica, tale indipendenza è solo parziale: perciò abbiamo provato a risolvere almeno in parte le incompatibilità, lavorando ad un *porting* affinchè MjSip potesse utilizzare gli strumenti offerti dalle API *BlackBerry* o implemdentazioni native in Java

dei servizi (anche limitate se necessario), al posto delle API non supportate.

Ci siamo limitati a modificare le parti dello stack effettivamente utili per un'applicazione VoIP, tralasciando funzionalità ulteriori di MjSip il cui porting sarebbe stato costoso e superfluo, come ad esempio l'implementazione del trasporto dati attraverso TCP.

Modifiche a livello di trasporto

Descrivendo MjSip a livello di trasporto, abbiamo accennato al fatto che per creare connessioni UDP (o TCP) e per gestirle utilizza la libreria java.net. Sui dispositivi *BlackBerry* invece, abbiamo visto come si utilizzino API delle librerie J2ME o create da RIM.

Grazie alla struttura a livelli di MjSip, abbiamo dovuto modificare soltanto la classe *UdpSocket* che si occupa di creare le connessioni a livello di trasporto. Lo abbiamo fatto sostituendo il codice presente con chiamate alle funzioni per connessioni socket messe a disposizione dalle API *BlackBerry*. Per fare ciò abbiamo sfruttato le conoscenze acquisite studiando l'implementazione del protocollo UDP su *BlackBerry*, ma abbiamo anche adoperato allo stesso tempo le funzionalità descritte nel capitolo due per il monitoraggio delle reti wireless.

In questo modo al livello di trasporto abbiamo potuto ottenere, nell'entità *UdpSocket*, non solo la gestione della ricezione e dell'invio di datagram UDP, ma anche il controllo e la notifica in tempo reale di eventuali cambiamenti nello stato delle reti wireless.

Lo svantaggio maggiore di queste modifiche è stato dato dalla necessità di usare socket bloccanti per effettuare connessioni UDP: ricordiamo dal capitolo precedente che questo implica il blocco del thread quando si utilizzano le primitive di ricezione, fino a che non viene ricevuta una qualsiasi quantità

di dati sul socket.

In MjSip questo problema viene risolto da un entità che lavora al di sopra di UdpSocket, detta *UdpProvider*: si tratta di un thread che si occupa di gestire ricezione ed invio dei dati separatamente.

Capitolo 3

Obiettivo: lo streaming audio

Andiamo ora a progettare e implementare, utilizzando i livelli sottostanti messi a nostra disposizione da MjSIP e dalle API *BlackBerry*, la parte della nostra applicazione che dovrà occuparsi della registrazione, invio, ricezione, ordinamento e riproduzione dei dati voce.

3.1 Progettazione

MjSip mette a disposizione alcune classi per permettere la registrazione di flussi audio, ed il successivo invio attraverso datagram UDP/RTP. Anzitutto si può notare come il caso sia differente dal nostro: noi necessitiamo di trasmissione e ricezione di dati continua e quasi contemporanea per tutta la durata della sessione vocale, e non di inviare e ricevere un flusso preregistrato. Abbiamo però fatto uso dei servizi offerti per l'invio e la ricezione dei datagram RTP, con opportune estensioni e modifiche. In MjSip, sono le classi `RtpStreamSender` ed `RtpStreamReceiver` che si occupano di questo compito, entrambe istanziate dall'entità `JAudioLauncher`, a sua volta invocata al termine dell'handshake tra i due *User Agent* . `RtpStreamSender` si accerta di avere ricevuto in input un socket UDP ed uno stream preregistrato di dati; questo flusso viene suddiviso in molteplici frame di dimensioni predefinita, ed ogni singolo frame a sua volta incapsulato come payload di

datagram RTP. A tal fine MjSip fornisce un'entità, `RtpPacket`, per creare e manipolare pacchetti RTP, con metodi per estrarre o settare tutti gli header del protocollo RTP ed il payload stesso. Nella versione originale di MjSip, viene fornito un flusso audio con codifica PCM *(Pulse Code Modulation)* ed algoritmo di compressione μ-law, e dimensione fissa dei frame di 500 bytes. `RtpStreamSender` incapsula ogni frame dello stream dato in input e setta gli header timestamp, sequence number, e payload length servendosi di un socket UDP per trasmettere ogni pacchetto. Analogamente `RtpStreamReceiver` fa uso di un socket UDP per la ricezione ed inizializza uno stream vuoto di dati. Attraverso il socket, si mette in attesa di ricevere i pacchetti RTP in entrata; con il payload di questi, provvede a ricostruire lo stream audio originale mediante operazione di bufferizzazione. Data la sua natura cross platform, MjSip non specifica particolari entità per registrare e riprodurre input vocale, che devono essere scelti ed implementati a seconda dellla piattaforma specifica.

Saranno quindi queste due classi `RtpStreamSender` ed `RtpStreamReceiver` che dovremo andare a modificare per ottenere le funzionalità che ci servono. Infatti a differenza dell'implementazione di MjSip noi dobbiamo gestire uno streaming real-time e non un flusso preregistrato, perciò dovremmo occuparci anche della registrazione e della riproduzione dello stream audio; inoltre dovremmo utilizzare connessioni UDP compatibili con le API *BlackBerry*, utilizzare un codec audio supportato dalle stesse e gestire l'ordinamento dei pacchetti.

Per semplicità partiamo dall'implementazione di una comunicazione one-way, dove un client parla e l'altro ascolta. Per quanto concerne il livello di trasporto, come detto prima, la modifica principale da apportare è quella di utilizzare i socket forniti dalle API *BlackBerry*, al posto di quelli usati in MjSip. La connessione UDP utilizzata per i datagram RTP voce, come da specifiche, non sarà la stessa adottata per l'handshake tra i due

user agent, ma verrà instaurata al termine di questa fase di handshake ed i socket corrispondenti verranno poi passsati alle entità `RtpStreamSender` e `RtpStreamReceiver`.

Analizziamo ora quello che presumibilmente sarà uno dei punti di massima criticità della nostra applicazione, ovvero la riproduzione dello streaming; affermo che si tratti di uno dei punti più critici in quanto l'unico modo per poter gestire la riproduzione di dati audio passa attraverso le API di *BlackBerry* ed in particolare attraverso la classe `Player`, che viene utilizzata sia per la riproduzione di flussi audio sia per la registrazione di input vocali provenienti dal microfono del dispositivo mobile. Sembra infatti che le API di RIM non mettano a disposizione un altro modo per gestire le periferiche audio dei propri dispositivi per ciò questa classe `Player` potrebbe rappresentare un potenziale collo di bottiglia nel caso in cui non soddisfi le nostre esigenze.

Innanzitutto analizziamo come funziona la suddetta classe `Player`; durante il proprio ciclo di ezecuzione un'oggeto di questo tipo può trovarsi in 5 stati distinti:

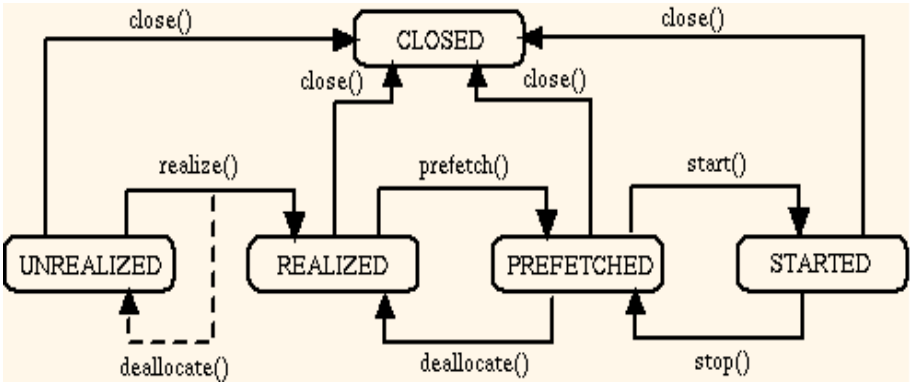

Fig. 3.1: Diagramma di transizione degli stati di `Player`

- **UNREALIZED:** lo stato iniziale alla creazione del `Player`, quando non ha informazioni per gestire le risorse con le quali deve lavorare;

- **REALIZED:** il `Player` ha ottenuto sufficienti informazioni sulle risorse multimediali che deve gestire, e le può perciò utilizzare, ad esempio leggendo un file o interagendo con un server;

- **PREFETCHED:** prima di essere avviato, il `Player` passa in questo stato se ha necessità di acquisire risorse esclusive oppure eseguire operazioni di bufferizzazione sugli stream multimediali per ottenere latenza ridotta durante le operazioni successive. Il `Player` ritorna in questo stato anche quando viene fermata la sua esecuzione con il metodo `stop()`;

- **STARTED:** Il `Player` è stato avviato e sta processando i dati in I/O.

Per controllare la registrazione è necessario utilizzare anche un oggetto `RecordControl` istanziato a partire dal `Player` stesso, fornendogli uno stream vuoto da riempire con i dati audio. `RecordControl` deve essere avviato assieme al `Player` per effettuare la registrazione, e interrotto con il metodo `commit()` prima della chiusura del `Player` al fine di terminare la registrazione. Da questo punto di vista il `Player` sembra soddisfare a grandi linee le nostre neccessità, anche se dovremo andare ad analizzare meglio le sue caratteristiche, quali i codec audio da esso supportati, la possibilità di poter riprodurre un flusso audio mentre viene utilizzato il microfono per registrazione e le sue prestazioni.

Per quanto riguarda il lato ricevente e la riproduzione dei pacchetti voce la questione appare più complessa. Per ovvi motivi non possiamo passare al player i datagram uno alla volta; in caso contrario si presenterebbero i seguenti problemi:

- **Degrado delle prestazioni:** data la gran quantità degli stessi pacchetti voce, `Player` sarebbe costretto a molteplici (e computazionalmente insostenibili) passaggi di stato, dovendo essere ogni volta inizializzato per riprodurre ogni frame audio;

- **Impossibilità di lettura dei dati:** la ridotta dimensione dei frame (contenenti dati voce nell'ordine delle decine di ms) impedirebbe comunque la riproduzione di uno stream dati cosi piccolo.

`Player` mette però a disposizione un'interfaccia `DataSource` dal quale può ricevere dati audio senza interessarsi di come questi vengano effettivamente reperiti; tale interfaccia garantisce un livello di astrazione tale da permetterci di implementare un nostro `DataSource` che riceva i pacchetti dalla nostra connessione RTP su UDP, li ordini e li presenti al `Player` per riprodurli.

`DataSource` possiede una serie di metodi utili a questo scopo, oltre a metodi secondari per specificare il tipo di dati in streaming e settare un URI per il recupero dei dati dalla sorgente di provenienza. I metodi più importanti sono `connect()` e `disconnect()`, che servono per istanziare un'implementazione di ogni flusso di dati audio da gestire. Questa implementazione è riferita ad una interfaccia `SourceStream` che rappresenta un flusso audio ; nel nostro caso, l'unico flusso da gestire è quello dei datagram RTP provenienti dallo UA client. Altri due metodi di `DataSource`, `start()` e `stop()`, servono per avviare e chiudere i flussi suddetti.

Affinché sia possibile sviluppare un'applicazione VoIP è fondamentale che sia possibile istanziare due `Player` che operino contemporaneamente uno registrando e l'altro riproducendo uno streaming; fortunatamente questo è possibile anche se è necessario rispettare alcuni vincoli [15]:

- alla creazione del player bisogna aggiungere il parametro *voipMode=true*;
- il codec audio utilizzato deve essere audio/amr o audio/qcelp;
- il codec audio utilizzato per la registrazione e per la riproduzione deve essere il medesimo;
- il bitrate deve essere lo stesso per la registrazione e la riproduzione e deve essere valido per il codec in questione

- l'ordine di creazione dei player non dovrebbe essere rilevante, il player che registra deve per essere avviato prima di quello che riproduce;
- i due player devono essere creati dalla stessa applicazione;
- il player che riproduce deve ricevere i dati da uno stream (`InputStream` o `DataSource`)

Nel progettare la nostra applicazione dobbiamo quindi ricordarci di rispettare queste condizioni, altrimenti anche se riuscissimo ad effettuare una comunicazione one-way sarebbe impossibile procedere all'implementazione di quella bidirezionale.

Cercando in Internet altri sviluppatori che si erano trovati nella condizione di dover implementare un loro custom `DataSource` per gestire flussi in streaming mi sono imbattuto nel seguente articolo [16]. In breve l'autore Mike Kirkup presenta delle API open source per lo streaming di contenuti audio e video tramite RTSP e HTTP; queste API si basano essenzialmente sull'implementazione di un `DataSource` e di un `SourceStream` ad-hoc e presentano un'architettura organizzata secondo lo schema seguente (Fig.3.2):

I componenti di questa architettura sono:

- **Source Media:** il file audio o video che ci interessa riprodurre, posizionato su un server
- **Download Thread:** un thread che si connette al Source Media e scarica il file nel MediaBuffer
- **Media Buffer:** un buffer circolare per immagazzinare i dati finchè non sono mandati al player per la loro riproduzione
- **SourceStream:** un'istanza di javax.microedition.media.protocol.SourceStream class; `SourceStream` si comporta come un input stream per il media player mettendo a disposizione metodi come read() e seek().

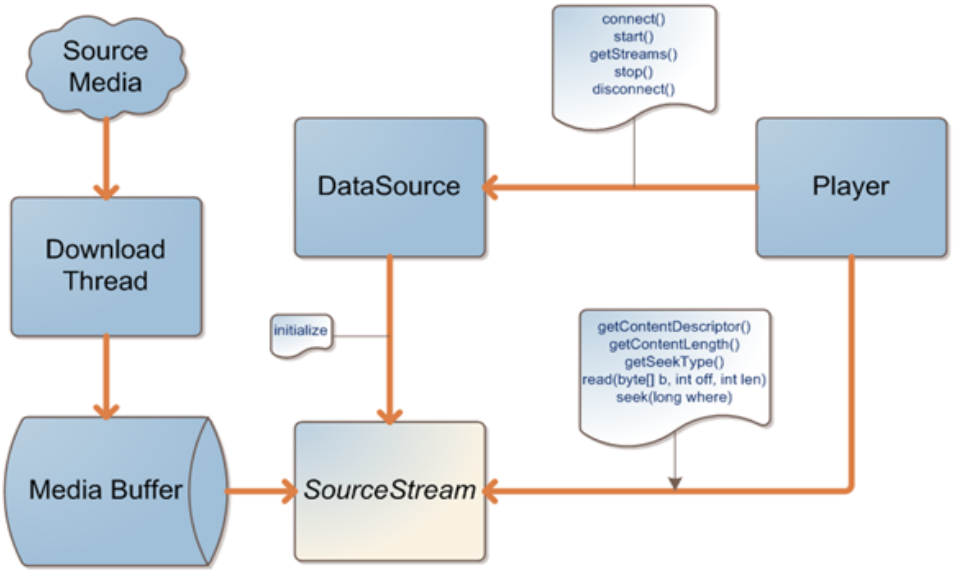

Fig. 3.2: L'architettura del modello proposto

- **DataSource:** un'istanza di javax.microedition.media.protocol.DataSource. `DataSource` esegue le seguenti azioni:

  1. apre una connessione con il file remoto;
  2. avvia il download thread;
  3. inizializza `SourceStream` e lo restituisce per riprodurlo;
  4. chiude la connessione.

- **Player:** un'istanza di javax.microedition.media.Player. Il `Player` riproduce il contenuto del `SourceStream` via via che questo viene scaricato.

Queste API si offrono come soluzione ad un problema simile a quello che ci troviamo ad affrontare. Effettuando le modifiche opportune si potrebbe ottenere uno `StreamingPlayer` in grado di riprodurre dati provenienti da una connessione UDP. Sarebbe infatti sufficiente prendere l'architettura sopra proposta e sostituire il Download thread con un thread ricevente che riceve i pacchetti dal socket UDP li ordina e li immagazzina nel buffer per essere poi consumati da `SourceStream`.

In realtà andando ad analizzare il codice si vede che la modifica non risulta così semplice come potrebbe sembrare dalla sola analisi dell'architettura proposta; sono presenti infatti numerose modifiche da apportare nel codice e, sopratutto, salta all'occhio una grande quantità di attività che risultano superflue per la nostra implementazione (dovute al fatto che queste API nascono per risolvere un problema di natura simile, ma comunque diverso dal nostro).

Anche se l'architettura in questione prevede l'utilizzo di un buffer circolare non ne è tuttavia presente un'implementazione e le librerie *BlackBerry* non la mettono a disposizione. Sono quindi andato alla ricerca dell'implementazione open source di un buffer circolare in Java e ho trovato questa implementazione di Stephen Ostermiller [17] che rispondeva alle mie esigenze.

Dopo aver modificato il codice dove era necessario ed aver cercato di eliminare tutto ciò che era presente di superfluo al momento di dover integrare il codice con `RtpStreamReceiver` mi sono trovato ad affrontare difficoltà superiori al previsto. Dopo aver insistito per qualche tempo nel tentativo di integrare le due componenti senza ottenere grossi risultati mi sono reso conto che l'implementazione modificata per adattarsi alle mie esigenze risultava comunque molto più complessa di quel che era effettivamente necessario e globalmente l'implementazione del `DataSource` e del `SourceStream` risultava poco pulita mentre sarebbe potuta essere molto più semplice ed elegante. Per questo motivo ho deciso di modificare leggermente l'architettura della soluzione e di procedere all'implementazione di `DataSource` e `SourceStream` da zero per poter ottenere maggior semplicità ed eleganza e, soprattutto, un'implementazione funzionante.

Andando ad analizzare l'architettura osserviamo che potrebbe essere sem-

plificata integrando il Downloader thread all'interno di `SourceStream`; questo potrebbe occuparsi di ricevere i pacchetti, ordinarli, immagazzinarli in un proprio buffer personale e fornirli al player per mezzo della chiamata read().

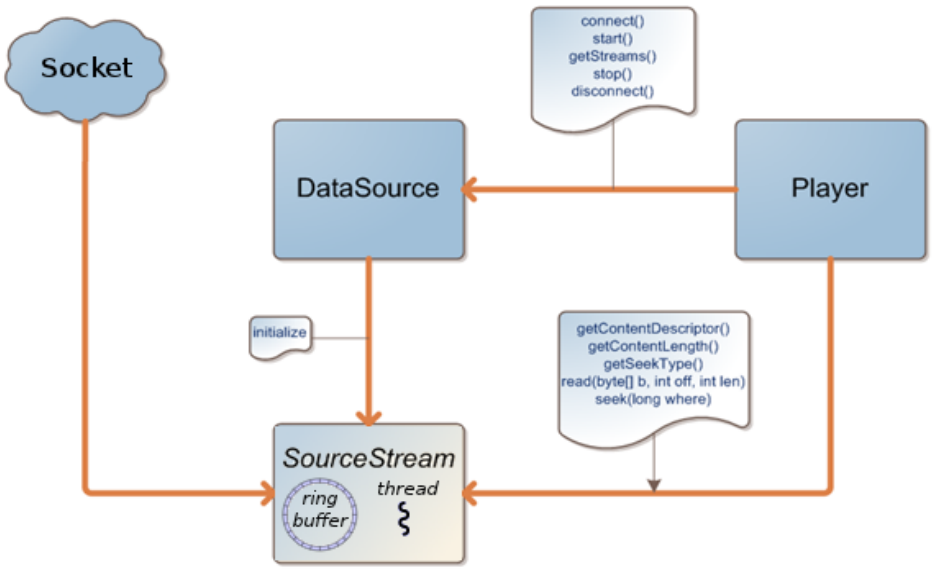

Fig. 3.3: Il nuovo modello

Il nostro modello dal lato ricevente prevede quindi la presenza di due distinti thread che operano concorrentemente. Un thread, `RtpStreamReceiver`, si occuperà di istanziare il player a partire da un `DataSource`, di avviarlo e di monitorarlo; d'altra parte `DataSource` sottende un proprio `SourceStream`, il quale è esso stesso un thread, il cui ciclo di vita consisterà nel ricevere datagram, ordinarli e metterli a disposizione su un proprio buffer che sarà acceduto da parte del `Player` per mezzo della chiamata read() della classe `SourceStream`.

Come abbiamo accennato nel paragrafo precedente il compito di `RtpStream`

`Receiver`, una volta avviato il `Player`, sarà quello di monitorare il `Player` stesso: infatti nel caso in cui il `Player` non riceva più dati da parte del `SourceStream`, sia per qualche problema nella connessione, che per qualche ritardo dovuto all'esecuzione del programma, questo interromperà la riproduzione. Sarà quindi necessario intervenire con opportune politiche per prevenire questo fatto e per riavviare il `Player` in caso di interruzione. Notiamo che dal lato mittente non avremo questo problema in quanto una volta che viene avviata la registrazione da parte del `Player` questo non si interromperà fino a che non verrà esplicitamente richiamato il metodo `stop()` su di esso; una volta avviato non sarà quindi necessario monitorarlo e si procederà entrando in un loop che ad intervalli regolari (intervalli equivalenti o comunque di poco inferiori alla durata di un frame audio) recupera i dati dal buffer di registrazione, li impacchetta e li spedisce. Tale loop procederà fino al termine dello streaming quando verrà fermato il `Player` e verranno deallocate tutte le strutture dati impiegate nel corso dell'esecuzione.

3.2 Implementazione

Avendo bene in mente come strutturare le varie componenti che costituiscono il cuore della nostra applicazione, ho iniziato lo sviluppo del codice vero e proprio; nella fase di sviluppo ovviamente ho incontrato degli ostacoli che non erano stati previsti in fase di progettazione: alcuni di queste problematiche sono state risolte, altre invece rimangono aperte e dovranno essere affrontate e risolte per poter arrivare ad avere un'applicazione funzionante. Andiamo ora a vedere con maggior dettaglio il lavoro che è stato svolto nella fase di sviluppo e le principali problematiche incontrate.

Innanzitutto per ridurre la complessità delle operazioni di sviluppo si è iniziato implementando un'applicazione da lanciare su un singolo dispositivo

che generasse un thread mittente ed un thread ricevente e cercasse di effettuare la comunicazione di uno streaming audio tra di essi in locale. Infatti il simulatore per dispositivi *BlackBerry* a disposizione è già pesante di per sè, se poi viene collegato ad esso il debugger di Eclipse diventa ancora più pesante; in conclusione tentare di effettuare il debugging lanciando simultaneamente due simulatori con due debugger collegati e provare a farli comunicare fra loro utilizzando un'applicazione che è ancora in fase iniziale di sviluppo (quindi piena di errori e di inefficienze) risulta essere tedioso e complesso, portando ad uno sviluppo estremamente lento. Senza considerare il fatto che il problema in questione è già di per sè piuttosto complicato visto che ci troviamo ad operare in un ambiente concorrente in cui la sincronizzazione deve essere pressoché perfetta per garantire una QoS sufficiente. Inoltre, anche se questo approccio non ci permette di controllare eventuali problemi dovuti all'instradamento dei pacchetti o problemi di connessione in genere, esso ci consente però di verificare se su uno stesso dispositivo possono convivere ed operare efficacemente sia `RtpStreamReceiver` che `RtpStreamSender`, cosa che dovrà necessariamente avvenire quando si avrà l'applicazione VoIP funzionante e quindi una trasmissione di stream voce bidirezionale.

Inizialmente ho deciso di concentrarmi maggiormente sul ricevente, che sembrerebbe essere la componente più complessa del sistema, perciò ho messo temporaneamente da parte la questione della registrazione della voce da parte del mittente e mi sono concentrato sull'invio di pacchetti preregistrati (nella fattispecie un file audio presente sul dispositivo). Inoltre ho posticipato la faccenda dell'ordinamento dei pacchetti da parte del ricevente (anche perché in questo contesto i pacchetti arrivano ordinati dato che la comunicazione avviene in locale). In questo modo ho potuto focalizzarmi sulle funzionalità da implementare nel custom `DataSource` e relativo `SourceStream`, che sono stati chiamati rispettivamente `StreamingDataSource` e `RTPSourceStream`.

Una volta verificata la corretta instaurazione della connessione, la suddi-

visione dello stream in frame, l'impacchettamento, la ricezione, lo spacchettamento, l'immagazzinamento del buffer, il reperimento da parte del `Player` dei pacchetti dal buffer e l'effettivo avviamento del `Player` (un lavoro che comunque ha richiesto tempo), ho iniziato ad implementare gradualmente le parti che mancavano.

Per prima cosa mi sono preoccupato di implementare l'ordinamento dei pacchetti. Per ottenere questo risultato ho deciso di reallizzare una struttura dati apposita che permettesse di realizzare in modo semplice l'ordinamento dei pacchetti e anche altre funzionalità necessarie per il corretto funzionamento dell'applicazione. Nella fattispecie il `Player` deve ricevere dati da parte del `RTPSourceStream` ad intervalli regolari, siano essi stati effettivamente ricevuti o meno, altrimenti questo interromperà la riproduzione dello streaming. Nel caso ottimale se nel momento in cui i dati devono essere forniti al `Player` tutti i pacchetti che ci servono sono stati ricevuti allora non abbiamo problemi; nel caso invece in cui qualche pacchetto manchi dobbiamo decidere come comportarci. L'atteggiamento comunemente adottato in ambito VoIP nel caso in cui manchi un frame è quello di ripetere il frame precedente più vicino, in modo da non dover mandare un frame di silenzio o di dati casuali.

Ho quindi implementato un buffer speciale che permettesse di compiere queste azioni in modo semplice nascondendo ad `RtpStreamReceiver` la complessità delle operazioni da eseguire. Questo buffer speciale, chiamato `Slot Buffer`, è un buffer circolare suddiviso in blocchi di dimensione fissa, e tale dimensione sarà uguale alla dimensione dei frame audio; all'interno dell'oggetto `SlotBuffer`, oltre al buffer vero e proprio, è anche presente un vettore di booleani, in modo tale da avere un booleano per ogni cella del buffer principale. Tale vettore di booleani serve per segnalare se la cella corrispondente è occupata o meno. Per poter scrivere dati nel buffer è necessario chiamare il metodo `write(byte[] buf, int pos)`; tale metodo farà sì che

la pos-esima cella del buffer venga riempita con i dati presenti in `buf`; se `pos` risultasse maggiore di `num` (dove `num` è il numero di blocchi all'interno del buffer) la cella in cui verranno scritti i dati sarà quella di indice `pos % num`. In questo modo per ordinare i pacchetti in ingresso e sufficiente richiamare `write` passando come parametri il payload del pacchetto e il suo sequence number. In questa fase mi sono accorto di un bug presente nel metodo `RtpPacket.setSequenceNumber()` di MjSip; il valore del sequence number infatti non veniva scritto correttamente nell'header del pacchetto causando poi problemi alla ricezione.

L'operazione di read dello `SlotBuffer` è invece rigorosamente sequenziale ed alla sua chiamata si recuperano le successive n celle del buffer (per n fissato opportunamente alla creazione del buffer); nel caso in cui qualche cella sia mancante (e ce ne accorgiamo grazie al vettore di booleani) queste vengono sostituite al momento con il contenuto della cella precedente occupata più vicina. `SlotBuffer` possiede inoltre un indice `slotRead` che ci comunica quanti pacchetti sono stati letti (tale indice coinciderà con il sequence number dell'ultimo pacchetto letto), questo ci permette di scartare automaticamente al momento della write i pacchetti obsoleti arrivati troppo tardi.

Per chiarire il funzionamento del buffer vediamo un esempio. In figura

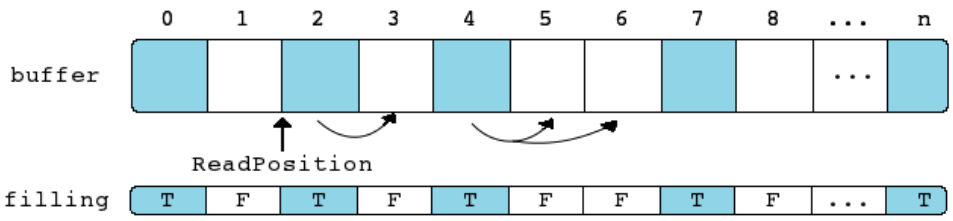

Fig. 3.4: Esempio di SlotBuffer prima della read()

(Fig. 3.4) abbiamo rappresentato lo stato di uno `SlotBuffer`; sono rappresentati il buffer principale, il buffer di booleani, denominato filling, e la posizione dell'indice `ReadPos`, che indica il punto dello stream raggiunto con le letture precedenti. Le celle colorate del buffer sono occupate. Sullo SlotBuffer sono state effettuate delle write sulle posizioni 0, 2, 4, 7 e n; nel momento in cui viene invocata una read (ipotizzando che la read sia impostata per restituire 5 celle alla volta) le celle 3, 5 e 6, trovate vuote, vengono riempite copiando il contenuto della cella precedente occupata più vicina: infatti, come indicano le frecce, il contenuto della cella 2 è copiato nella cella 3 e quello della cella 4 nelle celle 5 e 6.

Nella nuova figura (Fig.3.5) possiamo vedere come ora le celle 3, 5 e

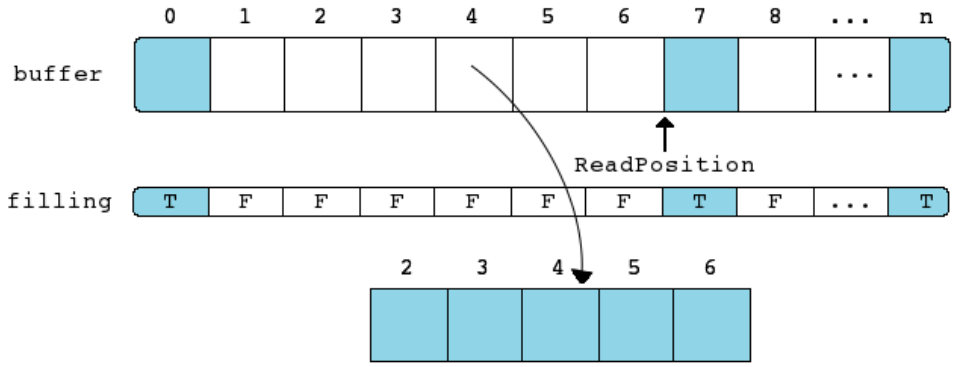

Fig. 3.5: Esempio di SlotBuffer dopo la read()

6 risultino occupate e vengano restituite al chiamante tramite una copia della porzione del buffer che va dalla cella 2 alla cella 6. Infine vediamo come `ReadPos` si sposti in avanti di un numero di celle pari a quello delle celle lette e come gli slot letti risultino ora liberi, essendo state settate a `false` le celle corrispondenti di filling; in realtà i dati presenti nel buffer non sono stati cancellati ma le celle sono disponibili per una futura sovrascrittura.

La corretta implementazione dello `SlotBuffer` è stata verificata per mezzo di un apposito modulo di testing JUnit, anche se potrebbe essere saggio effettuare test più approfonditi perché un'implementazione non conforme con il comportamento descritto potrebbe compromettere l'intera applicazione.

Il nostro `RTPSourceStream` possiede quindi due buffer: un buffer circolare classico, `CircularByteBuffer`, ed un buffer circolare speciale del tipo appena descritto, `SlotBuffer`. I pacchetti vengono inizialmente depositati nello `SlotBuffer` che provvede ad ordinarli e poi vengono periodicamente passati nel `CircularByteBuffer`. Si noti che la funzione `read` fornita da `RTPSourceStream` preleva i pacchetti da `CircularByteBuffer`; questo significa che finchè i pacchetti risiedono nello `SlotBuffer` questi non sono effettivamente disponibili da parte del `Player` ma lo diventano solo nel momento in cui transitano nel `CircularByteBuffer`.

Bisogna quindi che periodicamente ci si occupi di trasferire i pacchetti dallo `SlotBuffer` al `CircularByteBuffer` in modo che il `Player` non rimanga a corto di pacchetti. Per fare ciò bisogna impedire che il thread rimanga per un tempo indefinito sulla receive su un socket bloccante; fortunatamente, come abbiamo visto, le API di *BlackBerry* permettono la creazione di socket non bloccanti modificando il comando di creazione del socket nel modo seguente:

```
_conn =
(UDPDatagramConnection)Connector.open(
"datagram://:"+port, READ, true);
```

Infatti se alla creazione del socket da parte del ricevente il terzo parametro della chiamata `Connector.open()` viene settato a true, la funzione restituir un socket non bloccante; tale socket ammette la chiamata alla funzione `setTimeout(t)` la quale permette di settare il timeout t espresso in millisecondi. Se dopo aver invocato la `receive` sul socket la chiamata non ritorna entro il timeout t, questa viene interrotta e viene sollevata un'eccezione. Pri-

ma di richiamare la `receive` sul socket sarà quindi necessario calcolare il timeout oppportuno e settarlo. Allo scadere del timeout viene catturata l' eccezione e l'handler di questa eccezione provvederà a prelevare dati dallo `SlotBuffer` trasferirli nel `CircularByteBuffer` e ritornare al normale ciclo di esecuzione del thread.

Successivamente mi sono occupato di implementare la registrazione dello stream voce dal lato mittente; ho inizializzato il `Player` per registrare in formato AMR(*Adaptive Multi-Rate Audio Codec*) [18] a 12.2 Kbps, codec adatto per la codifica dei pacchetti voce in applicazioni VoIP (anche perché è un formato compresso). In fase di testing ho verificato il corretto funzionamento del `Player` che registra e carica i dati in un apposito buffer circolare dal quale poi i dati vengono prelevati, suddivisi in frame, impacchettati e spediti. Mi sono reso conto che il player però carica i nuovi dati nel buffer ad intervalli di circa 1 secondo, tempo esageratamente lungo per un applicazione VoIP, tenendo conto che secondo ITU-T una latenza accettabile deve mantenersi inferiore ai 150ms. Andando a controllare la documentazione del `Player` ho verificato che l'intervallo di tempo allo scadere del quale i dati registrati vengono periodicamente caricati nel buffer é settabile per mezzo di un opportuno parametro alla creazione del `Player`. Il comando di creazione del `Player` viene quindi modificato e diventa:

```
_player = Manager.createPlayer(``capture://audio?encoding=audio/amr
&rate=12200&updateMethod=time&updateThreshold=20&voipMode=true'');
```

Dove `&updateMethod=time&updateThreshold=20` svolge per l'appunto questa funzione. Se non fosse che la documentazione *BlackBerry* riporta che questi parametri sono supportati solo da alcuni dispositivi, senza poi specificare quali siano effettivamente questi dispositivi. Pur avendo provato un discreto numero di simulatori (Torch 9800, Bold 9780, Bold 9700, Storm2 9550, Bold 9650) nessuno sembra supportare questa opzione; bisogna sperare che il parametro non sia supportato solo dai simulatori ma funzioni invece sui dispositivi realie bisognerebbe cercare di ottenere da RIM una lista dei

device che supportano tale parametro.

Anche se questo problema rimane al momento irrisolto le funzionalità richieste per avere la trasmissione e riproduzione di uno live-streaming sono presenti; per avere un corretto funzionamento è necessario però che le componenti coinvolte rispettino certi parametri prestazionali, in particolare sono cruciali le tempistiche. È importante che l'esecuzione sia efficiente anche perché al momento ci troviamo in una condizione ottimale, visto che non ci saranno problemi indotti dal malfunzionamento della connessione e dalla perdita di pacchetti, essendo la connessione locale.

Nel considerare i tempi di esecuzione Dovremmo tenere conto di un ritardo di circa un secondo introdotto dal `Player` registrante, come spiegato precedentemente; una volta però che i dati registrati vengono caricati nel buffer per la prima volta questo non dovrebbe causare ulteriori problemi: infatti il mittente avrà a disposizione un secondo circa di conversazione registrata e quando questo sarà stato spedito si saranno resi disponibili nuovi dati da spedire. Tenendo conto di questo ritardo di cui sappiamo la causa possiamo comunque andare a verificare se i pacchetti vengono spediti, ricevuti e passati al `Player` abbastanza velocemente o se viene accumulato un ritardo sullo streaming e quale sia eventualmente l'entità di questo ritardo.

La misurazione di questo ritardo non è comunque semplice da effettuare; in seguito a prove sperimentali mi sono infatti accorto che nel momento in cui il debugger interrompe l'esecuzione dell'applicazione, sia l'interruzione dovuta al raggiungimento di un breakpoint o ad altre cause, il tempo di sistema continua comunque a scorrere ed anche il `Player` che si occupa di registrare prosegue la registrazione. Per poter effettuare delle misurazioni è quindi necessario porre un breakpoint successivamente alle operazioni delle quali si vogliono misurare le tempistiche; qualsiasi misurazione di eventi successivi al breakpoint risulta falsata dall'interruzione della macchina virtuale.

Un altro dettaglio da considerare è che le misurazioni stesse fanno uso di un metodo su `System` (`System.currentmilliseconds()`) che introduce un overhead, che comunque se limitiamo il numero di misurazioni e non ne abusiamo non dovrebbe essere rilevante.

Da alcune misurazioni iniziali ho verificato che su un esecuzione effettiva di circa 2 secondi (ovvero a partire dalla spedizione del primo pacchetto) veniva accumulato un ritardo variabile nell'ordine delle centinaia di millisecondi, un ritardo certo non trascurabile. Andando ad analizzare più nel dettaglio quale parte del codice contribuiva a causare il ritardo mi sono reso conto di un'inefficienza nella funzione che passa periodicamente i dati dallo `SlotBuffer` al `CircularByteBuffer` che ho provveduto a risolvere.

Analizzando il tempo che trascorre tra un ciclo di esecuzione e quello successivo ho constatato che normalmente questo è di gran lunga inferiore ai 20ms, soglia sotto la quale dobbiamo mantenerci considerato che ad ogni ciclo di esecuzione viene ricevuto un pacchetto che contiene un frame audio di 20ms. Saltuariamente però si verifica un ritardo tra un ciclo di esecuzione ed il successivo dell'ordine di 100ms. Visto che tale ritardo sopraggiunge saltuariamente, intuitivamente non può essere dovuto ad una parte specifica del codice, altrimenti si dovrebbe verificare ad ogni ciclo di esecuzione; per giustificarlo ho quindi ipotizzato che tale ritardo fosse dovuto alla preemption e all'avvicendamento fra thread sulla macchina virtuale. Ho provato quindi ad aumentare la priorità del thread ricevente ma senza ottenere miglioramenti soddisfacenti.

Questa lentezza eccessiva potrebbe comunque essere provocata dal debugger e dal fatto che stiamo eseguendo su un simulatore; per avere una misurazione affidabile dovremmo poter eseguire su un dispositivo reale. Un primo passo per verificare questa ipotesi sarebbe comunque quello di eseguire l'applicazione senza debugger collegato. Ho quindi sostituito le stampe delle

informazioni che mi interessano, le quali venivano riportate sul terminale del debugger, con stampe sullo schermo del dispositivo, in modo tale da poterle visualizzare anche senza debugger collegato. Purtroppo però, a causa di un'eccezione sollevata dal simulatore, l'esecuzione non terminae non è quindi possibile vedere le informazione che ci interessano; il team di sviluppo *BlackBerry* afferma che tale eccezione (DE427) sia dovuta ad un errore di Windows e non del simulatore stesso. Anche se il team è a conoscenza di tale problema al momento non è ancora stato patchato.

Resta il fatto che la nostra applicazione potrebbe subire dei ritardi tali da comprometterne il funzionamento. Ho quindi pensato ad un meccanismo per poter, se non rimediare, almeno compensare questi ritardi che, anche nel momento in cui l'applicazione dovesse funzionare correttamente, possono comunque occorrere a causa di fattori esterni, quali perdita di segnale, disconnessioni o ritardi nell'instradamento dei pacchetti. Un tentativo di soluzione può essere quello di implementare la `seek` su `RTPSourceStream`; in tal modo il ricevente, resosi conto di essere in ritardo sulla riproduzione dello stream, potrebbe cercare di "saltare" in avanti annullando il ritardo accumulato. Purtroppo questa operazione di `seek` in avanti potrebbe non essere possibile: ci possiamo infatti trovare il due situazioni differenti. La prima è che il tentativo di `seek` in avanti sia entro i limiti dei dati già ricevuti; ed allora la `seek` è effettivamente implementabile scartando i relativi dati. Il secondo scenario è che si voglia fare una `seek` in avanti su dati non ancora ricevuti; in questo caso l'unica cosa che si può fare è scartare i pacchetti nel momento in cui vengono ricevuti evitandone la bufferizzazione e quindi risparmiando alcune operazioni che altrimenti risulterebbero inutili.

Riflettendo però con più attenzione si può notare che lo sforzo di implementare la `seek` potrebbe rivelarsi inutile, o quantomeno meno utile di quanto si possa pensare. Infatti, per come è stato concepito, il thread ricevente, dal momento in cui riceve il primo datagram audio, fornisce dati al buffer

periodicamente, siano essi stati effettivamente ricevuti o meno, applicando la politica di rimpiazzo dei pacchetti mancanti descritta precedentemente. Per questo motivo dalla ricezione del primo pacchetto in poi il ricevente non dovrebbe più accumulare ulteriore ritardo sul mittente nella riproduzione dello streaming; al più potrebbe non aver ricevuto un numero sufficiente di pacchetti al fine di garantire una qualità sufficiente. Per queste ragioni la `seek` provvederebbe ad annullare solamente il ritardo accumulato precedentemente alla ricezione del primo frame audio. Ho comunque deciso di implementare la `seek`, per annullare questo ritardo iniziale (che finchè non verrà risolta la questione del ritardo in registrazione risulterà importante) e per permettere poi di avere la `seek` a disposizione per sviluppi futuri dell'applicazione, in quanto potrebbe risultare utile e si potrebbe decidere di adottare una politica di funzionamento diversa da quella attualmente realizzata.

Per poter utilizzare la `seek` efficacemente non è sufficiente la sua sola implementazione: bisogna infatti avere la possibilità di sapere dove fare `seek`, in altre parole bisogna conoscere l'entità del ritardo da compensare. Per poter calcolare questo valore è necessario conoscere, il punto dello stream in cui ci troviamo, ovvero l'istante della conversazione che stiamo effettivamente riproducendo, e il punto dello stream in cui in realtà dovremmo trovarci. Il primo valore può essere calcolato, avendo a disposizione il numero di byte passati al player fino a questo momento e il byterate a cui stiamo lavorando; per poter ricavare il secondo valore invece abbiamo bisogno di conoscere l'istante a cui è iniziata la registrazione da parte del mittente, informazione che non è a nostra disposizione. Infatti sottraendo al tempo corrente il valore dell'istante all'inizio della registrazione otterremmo l'istante dello streaming audio in cui dovremmo trovarci. Per questo motivo ho modificato il funzionamento del mittente in modo tale che salvasse l'istante di inizio della registrazione dello streaming e lo spedisse come primo pacchetto; il ricevente invece provvede a salvare questa informazione per utilizzarla successivamente

ogni volta che ne abbia la neccessità.

Capitolo 4

Linphone: un'applicazione VoIP per BlackBerry

Giunto ormai al termine del progetto sono venuto a conoscenza di un software VoIP open source multipiattaforma del quale esiste anche una versione per BlackBerry. La versione di Linphone per BlackBerry è stata rilasciata l'11 Gennaio 2011, purtroppo durante la ricerca condotta all'inizio del progetto per documentarmi su implementazioni preesistenti di applicazioni VoIP per BlackBerry non l'avevo trovata: avrebbe potuto essere utile per poter progettare il nostro applicativo avendo la possibilità di studiare un'implementazione funzionante. Almeno potremo studiarlo adesso per vedere come sono state affrontati e risolti dagli sviluppatori di Linphone alcuni dei problemi rimasti irrisolti nella nostra applicazione e magari per rivedere alcune delle scelte implementative adottate, nel caso in cui trovassimo delle soluzioni migliori.

4.1 Le caratteristiche

Linphone è un applicativo VoIP che presenta le seguenti caratteristiche:

- presenta un SIP user agent compatibile con RFC3261;
- supporta canali multipli simultaneamente con gestione delle chiamate;

- supporta diversi codec audio quali speex, G711, GSM, G722. Per mezzo di plugin addizionali supporta anche AMR e iLBC;
- supporta i codec video: VP8 (WebM), H263, H263-1998, MPEG4, theora e H264 (grazie ad un pluginbasato su x264), con risoluzioni da QCIF(176x144) a SVGA(800x600) dati un bandwith e una potenza della CPU sufficienti;
- supporta IPv6;
- supporta SIP/TLS;
- supporta qualsiasi webcam con driver V4L o V4L2 su linux e con driver Directshow su windows;
- servizio di instant messaging (utilizzando lo standard SIMPLE);
- rubrica;
- DTMF supportato usando SIP INFO o RFC2833;
- comprende SIP ENUMS;
- cancellazione dell'eco acustico utilizzando l'echo canceller disponibile in libspeexdsp (funziona anche con gli altri codec non solo con SPEEX);
- supporta SIP proxies multipli: registrar, proxies, outbound proxies;
- autenticazione con digest;
- Nat friendly: indovina l'indirizzo NAT per i messaggi SIP, usa STUN per gli RTP streams
- Sound backends: Linux: ALSA, OSS, PulseAudio Windows: waveapi MacOSX: HAL Audio Unit iPhone: VoiceProcessing AudioUnit con built-in echo cancellation Android sound system;

- gestione efficiente del bandwidth: le limitazioni del bandwidth sono segnalate tramite SDP (b=AS...), risultando in una sessione audio e video instaurata con un bitrate regolato in base alle capacità della rete in uso dallùtente;

- può utilizzare plugin: per aggiungere nuovi codec, o nuove funzionalità di base, come ad esempio la ricerca remota di cartelle degli indirizzi sip;

Già da questa lista ci si rende conto del fatto che Linphone sia un'applicativo piuttosto completo ed esteso. La versione per BlackBerry presenta però alcune limitazioni; è infatti supportato solo lo streaming audio, basato sul built-in AMR-NB codec, e i device supportati sono solo la serie Bold con OS > 5.0. Nonostante ciò è comunque presente il supporto sia per il wifi che per il 3G, nonchè il supporto per SIP TCP e UDP. Il motivo per cui Linphone su Blackberry supporta solo il codec AMR-NB è perché sul BB 0S 5, questo è l'unico codec disponibile con capacità real time. Per il funzionamento di Linphone è quindi necessario trovare un SIP provider che supporti tale codec.

4.2 L'architettura software

Linphone presenta una separazione interna tra interfaccia utente e nucleo dellàpplicazione, permettendo così di creare tipi diversi di interfacce utente costruite sopra le medesime funzionalità.

Il cuore dellàpplicazione è costituito da liblinphone, una libreria che ne implementa tutte le funzionalità. Liblinphone è un potente SIP VoIP video SDK che chiunque può utilizzare per aggiungere capacità di chiamate audio e video ad unàpplicazione, fornisce infatti delle API di alto livello per instaurare, ricevere e terminare le chiamate. La libreria liblinphone si affida alle seguenti componenti software:

- mediastreamer2,un potente SDK per creare streaming audio/video e processarli;

- oRTP, una semplice libreria RTP;

- eXosip2, la libreria SIP user agent, basata su libosip2.

Liblinphone e tutte le sue dipendenze sono scritte in C. Le funzionalità che ci interessano maggiormente saranno quindi quelle di mediastreamer2, la componente che si occupa della gestione degli streaming sulla quale quindi ci focalizzaremo trascurando le altre componenti.

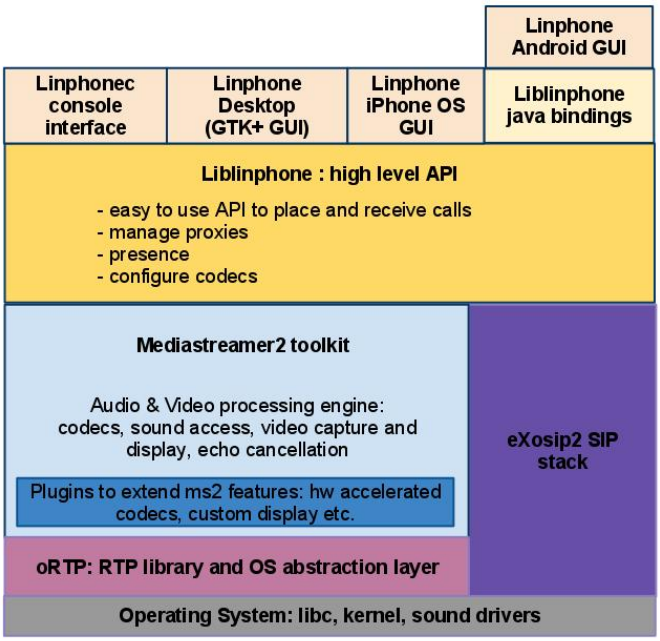

Fig. 4.1: L'architettura software di Linphone

4.3 Mediastreamer2

Mediastreamer2 è un potente e leggero gestore di streaming specializzato per applicazioni di telefonia audio e video; all'interno di linphone è responsabile di ricevere e trasmettere gli streaming multimediali, includendo anche la loro registrazione e il loro encoding, deconding e rendering.

Caratteristiche

Vediamone le caratteristiche principali:

- legge e scrive da e verso device di tipo alsa, oss e windows waveapi
- spedisce e riceve pacchetti RTP
- codifica e decodifica i seguenti formati: speex, G711, GSM, iLBC, AMR, H263, theora, MPEG4, e H264
- legge e scrive da e verso file wav
- legge immagini YUV da una webcam (se fornita di driver video4linux v1 o v2)
- visualizza immagini YUV (utilizzando la libreria SDL o le APIs native in windows)
- cancellazione dell'eco, usando l'algoritmo di cancellazione della libreria speex
- gestisce conferenze audio
- equalizzatore audio parametrico che utilizza un filtro FIR
- controllo del volume, controllo del gain automatico
- può essere esteso con plugin dinamici, attualmente sono disponibili codec plugin per H264, ILBC e AMR.

Già dalla lista delle caratteristiche vediamo come tutte le funzionalità di Linphone che ci interessa maggiormente esaminare siano contenute all'interno di questo componente, dalla registrazione e riproduzione fino all'incapsulamento e la spedizione dei pacchetti RTP.

Design

Il design e i principi di funzionamento che stanno dietro a mediastreamer2 sono piuttosto interessanti ed andiamo quindi a presentarli brevemente; inoltre nulla esclude che in futuro si possa decidere di utilizzare questa libreria per la gestione degli streaming audio all'interno della nostra applicazione quindi una breve introduzione sul suo funzionamento risulta sicuramente utile.

Ogni entità di processamento è contenuta all'interno di un oggetto MSFilter il quale ha input e/o output che possono essere utilizzati per connetterlo ad altri MSFilter. In pratica possono essere messe in sequenza unità che eseguono un compito relativamente semplice al fine di ottenere una funzionalità molto più ricca e complessa.

Procediamo con un rapido esempio; abbiamo a disposizione diversi tipi di MSFilter quali ad esempio:

- MSRtpRecv: un MSFIlter che riceve pacchetti RTP dalla rete li spacchetta e li mette sul proprio output.
- MSSpeexDec: un MSFilter che prende quello che gli viene dato in input assumendo che si tratti di pacchetti codificati con speex, lo decodifica e restituisce il risultato sul suo output.
- MSFileRec: un MSFilter che prende il suo input e lo salva in un file wav.

Prendendo i tre filter presentati sopra e mettendoli in sequenza si ottiene una componente che riceve pacchetti RTP, li decodifica e salva il risultato in un

file wav.

L'esecuzione di questi oggetti è schedulata da un MSTicker, un thread che si sveglia ogni 10 ms per processare i data in tutte le catene di MSFilter che gestisce; l'intervallo di tempo può essere modificato e diversi MSTicker possono essere utilizzati simultaneamente, per processare indipendantemente audio e video, per esempio.

Inoltre la creazione e l'implementazione di MSFilter non è complessa perciò si possono creare nuovi filtri per incrementare le funzionalità di mediastreamer2.

Mediastreamer2 è adatto a sistemi embedded poiché è leggero, il passaggio dei dati al suo interno è ottimizzato, è scritto in C, compilabile su arm con gcc e le sue uniche dipendenze minimali sono oRTP e libc, mentre tutte le altre (ffmpeg, speex, alsa...) possono essere aggiunte opzionalmente.

Conclusioni

Questo lavoro offre una visione d'insieme sulle fasi di sviluppo di un'applicazione VoIP. Dopo una doverosa panoramica introduttiva sul VoIP in generale e su alcuni protocolli che ne rendono possibile l'implementazione si è passati ad analizzare il lavoro che era già stato compiuto: dallo studio di un modello di architettura di rete innovativo (ABPS) all'analisi della piattaforma BlackBerry e alla successiva implementazione delle componenti base per poter arrivare allo sviluppo a livello applicativo; in questo modo si è quindi messo a disposizione una gestione delle interfacce tale da rispettare i principi il modello ABPS ed i protocolli SIP e RTP. Qui è iniziato il lavoro vero e proprio di cui questa tesi è il risultato ultimo.

Dopo aver studiato ed analizzato l'ambiente di sviluppo e gli strumenti a disposizione, mi sono documentato su implementazioni preesistenti (senza però ottenere grossi risultati) ed ho quindi iniziato l'implementazione dell'applicazione partendo dalla sua progettazione. A posteriori mi rendo conto che sarebbe stato più saggio dedicare più tempo alla fase di ricerca e documentazione; infatti, come già abbiamo visto, in fase di chiusura del progetto ho scoperto l'esistenza di un'implementazione funzionante di applicazione VoIP per BlackBerry. Dopo la fase di progettazione ho illustrato nel dettaglio l'avanzamento progressivo nella fase di implementazione, cercando di dare l'idea di come lo sviluppo sia proceduto per livelli successivi ed incrementali, partendo dalle funzionalità di base ed andando poi ad arricchirre l'applicazione in modo da ottenere le funzionalità desiderate. Si è visto anche che nella fase

di stesura del codice si sono presentate alcune difficoltà non previste in fase di progettazione, le quali a volte hanno richiesto di rivedere l'architettura progettata inizialmente ed altre volte hanno evidenziato alcuni limiti delle API BlackBerry alle quali si è posto, e si deve ancora, porre rimedio.

Al termine del lavoro purtroppo non sono riuscito ad ottenere un applicativo funzionante; tuttavia una volta risolte un paio di questioni particolarmente critiche non dovrebbe risultare molto complesso ottenere il funzionamento di un live streaming audio one-way.

Di seguito raggruppiamo e riassumiamo le problematiche irrisolte già menzionate e descritte con maggior dettaglio in precedenza. Queste saranno le prime questioni di cui dovrà occuparsi chi vorrà contribuire nel proseguire questo lavoro.

Il primo problema che sarà necessario risolvere è quello che riguarda l'aggiornamento troppo lento del buffer di registrazione da parte del player che registra lo streaming audio. Bisogna capire se i dispositivi reali supportano l'opzione, fondamentale per il funzionamento dell'applicazione, che permette di modificare l'intervallo di aggiornamento.

Analizzando l'implementazione di Linphone ho potuto constatare che per registrare lo streaming audio viene utilizzato un `Player` creato con gli stessi parametri che vengono utilizzati anche nella nostra implementazione ovvero:

```
"capture://audio?encoding=audio/amr&updateMethod=time
&updateThreshold=20"}
```

Questo ci dovrebbe dare una relativa tranquillità sul fatto che i dispositivi reali supportano realmente questa opzione, visto che gli sviluppatori di Linphone sostengono che la loro applicazione funziona su dispositivi BlackBerry. Questo potrebbe essere uno dei motivi per cui Linphone non funziona su tutti i dispositivi BlackBerry ma solo su quelli della serie Bold con BB

OS > 5, ovvero il fatto che solo questi dispositivi supportano effettivamente il parametro `updateMethod=time`, ma si tratta di una mera supposizione perché la documentazione non riferisce niente a riguardo.

Un'altro punto importante è quello di decidere il comportamento da adottare nel caso in cui il `Player` che riproduce lo streaming audio in arrivo si fermi. La classe `Player` presenta due metodi `setMediaTime()` e `getMediaTime()` che sembrano fare al caso nostro; se funzionassero probabilmente permetterebbero di risolvere il problema. Intuitivamente questi due metodi dovrebbero chiamare i metodi `seek()` e `tell()` dell'`RTPSourceStream` dal quale è istanziato il `Player`; al momento però non è ben chiaro se questi funzionano correttamente e se fanno effettivamente uso dei metodi appena menzionati. Per poter implementare correttamente `seek()` e `tell()` bisognerebbe capire se questi ricevono e restituiscono posizioni dello stream in termini di tempo o di spazio, in pratica se per indicare una posizione nello stream si usano i ms o i byte; dalla documentazione di questi metodi non è infatti specificata questa informazione.

Linphone non ci viene in aiuto perché la `seek` non è stata implementata, anche perché risulterebbe impossibile per come è stato implementato il `SourceStream`: i dati infatti vengono ricevuti dal socket e passati direttamente al player; il `SourceStream` non ha quindi la capacità di decidere cosa passare al player. Ciononostante possiamo comunque ottenere qualche informazione utile da Linphone; infatti la funzione `tell` del `SourceStream` è implementata e restituisce un valore che corrisponde al tempo in millisecondi trascorsi dall'inizio della riproduzione dello streaming. Sappiamo quindi che `tell` si riferisce a un tempo in millisecondi e non ad una quantità in byte e probabilmente anche `seek` utilizzerà la stessa metrica.

Sarà anche importante verificare il funzionamento dell'applicazione su un dispositivo reale, in modo da correggere gli errori che potrebbero venire

introdotti dalle differenze di funzionamento tra i simulatori ed i dispositivi reali. Il testing su un dispositivo reale permetterebbe inoltre di verificare che l'applicazione abbia delle prestazioni tali da permetterne l'effettivo funzionamento.

Un punto fondamentale, fino ad ora non menzionato, riguarda anche il fatto che al momento l'applicazione utilizza socket UDP normali che non implementano la politica ABPS; il nostro obiettivo è quello che la nostra applicazione implementi il modello ABPS quindi una volta che si avrà l'applicazione funzionante utilizzando i socket UDP normali sarà necessario sostituire questi con i socket modificati integrati in MjSip che implementano il modello ABPS per verificare che l'applicazioni funzioni e funzioni continuando a mantenere parametri prestazionali accettabili.

Il codice e tutta la documentazione relativa a questo progetto sono a disposizione di chiunque volesse contribuire al suo proseguimento e sviluppo per arrivare ad ottenere un'implementazione funzionante.

Bibliografia

[1] Fabio Pini. Voip su blackberry. Master's thesis, Alma Mater Studiorum - Universitá di Bologna, 2009/2010.

[2] Linphone. *Linphone*.

[3] J. Rosenberg H. Schulzrinne G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler. *SIP - Session Initiation Protocol*. RFC 3261. IETF, June 2002.

[4] ITU-T Recommendation G.1010. *End-user multimedia QoS categories*. November 2001.

[5] Jon Postel. *UDP - User Datagram Protocol*. RFC 768. *IETF*, August 1980.

[6] Jon Postel. *TCP - Transmission Control Protocol*. RFC 793. *IETF*, September 1981.

[7] M. Handley V. Jacobson. *SDP - Session Description Protocol*. RFC 2327. IETF, April 1998.

[8] H. Schulzrinne S. Casner R. Frederick V. Jacobson. *RTP - Real-time Transmission Protocol*. RFC 3550. IETF, July 2003.

[9] E. Gustafsson A. Jonsson. *Always Best Connected*. in IEEE Comm. vol. 10, no. 1, May 2003.

[10] WLAN Standards Working Group. *IEEE 802.11 - Wireless Local Area Network. IEEE 802 LAN/MAN Standards Committee*, URL:http://www.ieee802.org/11/, 1997.

[11] RIM. *BlackBerry JDE APIs 6.0.0*. URL:http://www.blackberry.com/developers/docs/6.0.0api/index.html, 2010.

[12] Sun MicroSystems. *Java Micro Edition*. URL:http://www.oracle.com/technetwork/java/javame/index.html, 2006.

[13] RIB CCSA ETSI ATIS TTA TTC. *Third Generation Partnership Project*. URL:http://www.3gpp.org/, 1998.

[14] Luca Veltri. *MjSip*. University of Parma, URL:http://www.mjsip.org/, 2005.

[15] RIM. *Everything you need to know about simultaneous recording.*

[16] Mike Kirkup. *Streaming Player API - Bringing the Big Screen to the Small Screen.*

[17] Stephen Ostermiller. *CircularByteBuffer implementation.*

[18] A. Lakaniemi J. Sjoberg, M. Westerlund. *Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multi-Rate (AMR) and Adaptive Multi-Rate Wideband (AMR-WB) Audio Codecs*. RFC 3267. *IETF*, June 2002.