CORE
Metadata, citation and similar papers at core.ac.uk
Provided by AMS Tesi di Laurea

Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

VoIP su BlackBerry

Tesi di Laurea in Architettura dei calcolatori

Relatore:
Dott.
Vittorio Ghini

Presentata da:
Fabio Pini

Sessione II
Anno Accademico 2009/10

A coloro che non smettono di sopportarmi

Indice

Introduzione **7**

1 Background **9**

1.1 VoIP - Voice over IP 9

1.1.1 Le origini e l'utilizzo 9

1.1.2 Lo stack di protocolli 11

1.2 ABPS - Always Best Packet Switching 15

1.2.1 Stato dell'arte ed obiettivi 15

1.2.2 Architettura proposta 16

1.2.3 RWM-PC - Robust Wireless Multi-Path Channel . . . 21

1.3 La piattaforma BlackBerry® 22

1.3.1 Cos'è BlackBerry? 22

1.3.2 Lo sviluppo di applicazioni in ambiente BlackBerry . . 22

1.3.3 L'architettura di rete BlackBerry 26

2 Livello di trasporto su Blackberry **29**

2.1 Connessioni UDP su Blackberry 29

2.2 Monitoraggio di reti wireless su BlackBerry 31

2.3 Testing e problematiche di simulazione 32

3 Implementazione di SIP su BlackBerry **35**

3.1 Requisiti 35

3.2 Perché MjSip? 36

3.3 MjSip: un'architettura a livelli 37

3.3.1 Transport Layer 37

3.3.2 Transaction Level 38

3.3.3 Dialog Level 38

3.3.4 Call Control Level 38

3.3.5 API aggiuntive 39

3.3.6 Il modello Provider→Listener 39

3.4 Porting di MjSip su Blackberry 40

3.4.1 Modifiche a livello di trasporto 40
3.4.2 Test e verifica di operazioni base 41
3.4.3 Handshake per le chiamate vocali 43

4 Audio streaming su BlackBerry 49

4.1 Le soluzioni proposte da MjSip 49
4.2 MjSip: stato dell'arte per la comunicazione via RTP 50
4.3 Streaming real time per comunicazione voce su BlackBerry . . 51
4.4 Problemi, limiti di simulazione e TODO 55

Conclusioni e sviluppi futuri 57

Bibliografia 59

Introduzione

A partire dalla prima metà degli anni duemila, grazie al successo di applicazioni commerciali come *Skype*^{TM}, ha cominciato a diffondersi tra utenti privati l'utilizzo della tecnologia VoIP (*Voice over IP*).

VoIP non identifica un singolo protocollo di comunicazione, ma un insieme di tecnologie, in prevalenza protocolli di comunicazione su rete, che interoperano tra loro al fine di fornire un meccanismo di trasmissione del mezzo vocale attraverso reti IP come Internet, anziché PSTN (*Public Switched Telephone Network*).

Come vedremo più specificatamente nel primo capitolo, questa tecnologia comporta una serie di vantaggi, soprattutto in termini di riduzione dei costi economici.

Molti produttori di dispositivi di comunicazione mobile hanno cominciato negli ultimi anni a supportare questa tecnologia, fornendo piattaforme compatibili con i client VoIP presenti sul mercato o mettendo a disposizione degli sviluppatori gli strumenti necessari per crearne di nuove ad hoc.

In altri casi ancora, gli stessi dispositivi basano il loro funzionamento su sistemi operativi ed ambienti di sviluppo *cross-platform* (come ad es. *Android*) che forniscono anche API (*Application Programming Interface*) apposite per la creazione di client *VoIP based*.

In questa tesi descriveremo invece gli sforzi compiuti per lo sviluppo di un prototipo di applicazione VoIP su una piattaforma proprietaria ideata dall'azienda RIM (*Research In Motion*)[1]. Essa produce gli *smartphone* denominati *BlackBerry* e provvede, tramite un'omonima architettura, ad offrire servizi agli utilizzatori di questi dispositivi. Vedremo più dettagliatamente nel primo capitolo quali sono questi servizi e come si differenziano in base alla categoria di utenza.

Per motivi di strategia commerciale, tuttora RIM non è entrata nella competizione per il mercato VoIP: questa scelta si traduce di fatto nella mancanza di servizi per i protocolli alla base della tecnologia VoIP, tra le API destinate agli sviluppatori di *Blackberry Apps*.

Abbiamo cercato di sopperire alle mancanze di questa piattaforma per

mezzo di implementazioni esterne da portare sul nostro ambiente di sviluppo BlackBerry, o con soluzioni native laddove possibili.

Oltre a questo, ci siamo posti come ulteriore obiettivo quello di ricercare soluzioni (perlomeno parziali) per l'impiego, sempre in ambiente BlackBerry, di un innovativo modello di architettura su reti *wireless* denominato ABPS (*Always Best Packet Switching*)[2]. Come verrà spiegato dettagliatamente nel primo capitolo, si tratta di una proposta per sfruttare al meglio le potenzialità dei dispositivi in grado di trasmettere dati attraverso reti wireless differenti.

Con alcune modifiche ai protocolli usati attualmente è possibile creare *client ABPS* che stabiliscono diversi canali di comunicazione sulle reti wireless supportate dal dispositivo e consentono di utilizzarne una a scelta per ogni singolo pacchetto dati da inviare. Questa scelta viene effettuata in base ad obiettivi legati a *QoS* (*Quality of Service*), benefici economici, ed altri ancora che illustreremo. Vedremo anche che si hanno numerosi vantaggi rispetto al modello attuale, in cui si utilizza una sola interfaccia di rete fino a che le prestazioni non degradano in maniera consistente.

A beneficiare di questi vantaggi possono essere proprio le applicazioni VoIP, viste le loro necessità in termini di QoS, per questo abbiamo ricercato strumenti tra le API BlackBerry da utilizzare per la creazione di un client ABPS, possibilmente integrato nella stessa applicazione VoIP.

La struttura del documento è delineata per dare nel primo capitolo una base teorica degli argomenti discussi e passare a descrivere nei successivi le fasi di ricerca e sviluppo sulla piattaforma BlackBerry. Nel capitolo due verrà trattata la fase di lavoro relativa alla trasmissione dati *transport level*, con riferimenti al lavoro svolto sull'implementazione del modello ABPS in tale ambito. Il capitolo tre descriverà la parte più estesa, incentrata sull'implementazione dei protocolli *session level* su BlackBerry. Infine il quarto ed ultimo capitolo si occuperà della fase di trasmissione vocale *application level*.

Lo sviluppo del software è ancora in corso ed il codice scritto è a disposizione di chiunque volesse utilizzarlo per interesse personale o per contribuire al lavoro già svolto.

Capitolo 1

Background

In questo capitolo cercheremo di fornire una base teorica riguardante gli argomenti trattati durante tutto l'arco del nostro lavoro, sforzandoci di comprendere gli aspetti generali e strutturali delle tecnologie di cui usufruiremo. Partiremo con la descrizione del VoIP, cercando di capire cosa si intende con questo termine di uso ormai comune e quali tecnologie vi stanno dietro.

Esploreremo poi la proposta di modello architetturale ABPS che si propone di migliorare le prestazioni nell'ambito delle comunicazioni con dispositivi mobili. Lo analizzeremo in maniera fortemente orientata ad implementazioni per applicazioni basate su SIP e VoIP.

Infine descriveremo la piattaforma BlackBerry, soffermandoci sulle caratteristiche tecniche ed i servizi offerti da questi dispositivi e dalla infrastruttura retrostante.

In questo capitolo vengono in questo modo offerte le conoscenze minime necessarie per comprendere lo sviluppo del progetto, il quale coinvolge in diversa misura tutte le tematiche presentate.

1.1 VoIP - Voice over IP

1.1.1 Le origini e l'utilizzo

Il termine VoIP indica un insieme di protocolli che permettono di effettuare conversazioni telefoniche attraverso reti che utilizzano il protocollo IP, come Internet.

Oggi questa tecnologia viene utilizzata, oltre che da aziende e grandi società di telecomunicazioni, anche da un numero sempre crescente di utenti privati che ne fruiscono sia da dispositivi mobili, che da pc desktop. Attualmente il VoIP non è ancora uno standard definito anche se diversi enti

internazionali, tra cui l'IETF (*Internet Engineering Task Force*), sono da tempo al lavoro.

La possibilità di effettuare chiamate audio tramite Internet nasce già nel 1995, con l'arrivo dei primi software per comunicazioni vocali tramite un PC connesso ad Internet (ad es. *Internet Phone Software*). Da quel momento gruppi commerciali e compagnie telefoniche cominciano a sviluppare la tecnologia utilizzando la suite di protocolli H.323.

Una svolta avviene nel 1999, 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 a breve e di cui tratteremo anche successivamente, 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. il già citato Skype o *Vonage*™) che diffondono l'utilizzo di questa tecnologia ad utenti privati, anche grazie alla possibilità di effettuare chiamate a tariffe agevolate verso la rete tradizionale PSTN.

I vantaggi del VoIP sono per l'appunto 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...

Gli svantaggi, invece, sono in particolare due:

- **Tutela della sicurezza**: questo problema, e le soluzioni ad esso proposte, rappresentano una parte importante e vasta dell'argomento, che noi non approfondiremo. Un accenno è comunque doveroso: per la maggior parte dei casi i problemi sono dovuti al fatto che una trasmissione vocale che viaggia su una rete Internet può subire attacchi proprio

come qualunque altra trasmissione dati, visto 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 comprensibili solo agli attori della comunicazione e la firma digitale, per garantire l'autenticazione degli utenti.
Vedremo come questo problema verrà affrontato anche nel modello di architettura ABPS.

- **Quality of Service**: con il passaggio da PSTN a rete IP, nascono una serie di problemi riguardanti l'integrità, la perdita dei dati ed il tempo di latenza. Occorre che un'applicazione rispetti certi parametri per fare in modo che la conversazione risulti fluida e comprensibile, inoltre può prevedere specifiche politiche di reinvio dei pacchetti persi o di selezione di quelli in ritardo.
  I parametri in particolare sono:

  - **Latenza**, ossia il tempo di percorrenza del pacchetto da un end all'altro: secondo *ITU-T* [4] deve essere inferiore a 150 ms per mantenere interattività sufficiente;
  - **Percentuale dei pacchetti persi**, possibilmente non superiore al 10%;
  - **Consumo di banda**, che deve essere garantito dai codec audio utilizzati nella comunicazione;
  - **Jitter**, ossia il ritardo tra due pacchetti giunti all'end uno dopo l'altro, dato dalla differenza di latenza. Se il Jitter risulta troppo alto (valore assoluto della differenza maggiore di 20 ms) le prestazioni degradano e la voce risulta spezzettata.

1.1.2 Lo stack di protocolli

Descriviamo ora una serie di protocolli impiegati per l'implementazione di applicazioni VoIP e quindi utilizzati nel nostro progetto. A seconda delle necessità e del contesto delle applicazioni che li utilizzano, l'utilizzo o meno di alcuni dei seguenti protocolli potrebbe variare.

Quello che ci interessa è dare nozioni di base e fornire una panoramica che ci permetta di comprendere meglio le implementazioni che descriveremo nei capitoli successivi.

L'ordine in cui sono descritti i protocolli è dal basso verso l'alto, facendo riferimento ai livelli dell'architettura di rete *TCP/IP*.

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 chiamati *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 consegna affidabile dei pacchetti e nemmeno che essi giungano ordinati. 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 è minimo, rendendolo adatto per trasmissioni come quelle VoIP, in cui si inviano grandi quantità di pacchetti audio e la perdita di alcuni di essi (entro certi limiti al fine di garantire Qos) è accettabile.

Abbiamo, nel dettaglio, un totale di 64 bit nella struttura di ogni datagram, di cui 32 per le porte sorgente e destinazione del pacchetto, 16 per la lunghezza del messaggio (in bytes) e 16 di *checksum* per il controllo degli errori.

SIP - Session Initiation Protocol

Costruito al di sopra del livello di trasporto, in particolare al livello di sessione, il *Session Initiation Protocol* è fondamentale per le comunicazioni VoIP, in quanto permette uno scambio di messaggi testuali per stabilire i parametri di base della comunicazione come gli indirizzi IP, il protocollo usato a livello di trasporto o i *codec* audio/video usati 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 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 sono messaggi comprensivi di un codice a tre cifre indicante il tipo di risposta, in stile *HTTP*. Queste sono le classi di risposta possibili:

- **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 riguardante 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).

Lasciamo ulteriori specifiche al terzo capitolo in cui accenneremo anche all'implementazione di questo protocollo.

RTP - Real-Time Protocol

RTP [8] è un protocollo (ancora una volta *nomen omen*) che serve per le applicazioni multimediali real-time ed è utilizzato dalle applicazioni 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*.

Come di SIP, anche di RTP torneremo a parlarne quando tratteremo di ABPS e dell'implementazione di SIP su BlackBerry.

1.2 ABPS - Always Best Packet Switching

1.2.1 Stato dell'arte ed obiettivi

Attualmente, 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 e spesso peculiari.

Nell'odierno scenario abbiamo un'evoluzione di queste tecnologie che segue principalmente tre *trend* differenti:

- Lo sviluppo di reti a banda larga per contesti differenti, come WiMAX e ZigBee;
- L'espansione di reti WiFi (standard IEEE802.11a/b/g/n) [9] tramite *access point* (ad es. Boing) o tramite la cooperazione tra utenti (ad es. Fonera);
- L'evoluzione delle reti 3GPP (*Third Generation Partnership Project*)) [10] a *3GPP2* grazie agli sforzi compiuta da parte di compagnie di telecomunicazione mobile per migliorare standard come *UMTS*.

Attualmente i dispositivi multi-homed non sfruttano appieno le loro possibilità per utilizzare di volta in volta interfacce di rete adatte al contesto, a causa soprattutto di limiti economici e tecnici.

Il modello utilizzato è l'ABC (*Always Best Connected*) [11], secondo il quale il dispositivo multi-homed MN (*Mobile Node*) individua la migliore delle reti wireless disponibili tramite le NIC e la utilizza finché le prestazioni non degradano.

L'obiettivo del progetto ABPS è quello di creare un modello di architettura più avanzato che permetta l'invio di ogni singolo *datagram* su una interfaccia di rete differente. La scelta dell'interfaccia di rete adatta potrà dipendere da requisiti di QoS, costi economici, esigenze di piattaforma, disponibilità delle reti, potenza del segnale, 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 già esistenti strutture wireless 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*).

Vediamo ora com'è effettivamente strutturata l'architettura ci concentreremo soprattutto sulle modifiche ai protocolli SIP/RTP necessarie per raggiungere gli obiettivi preposti.

Accenneremo poi soltanto gli aspetti architetturali più dettagliati che riguardano la protezione e l'autenticazione dei dati e casi di studio particolari come RWM-PC.

1.2.2 Architettura proposta

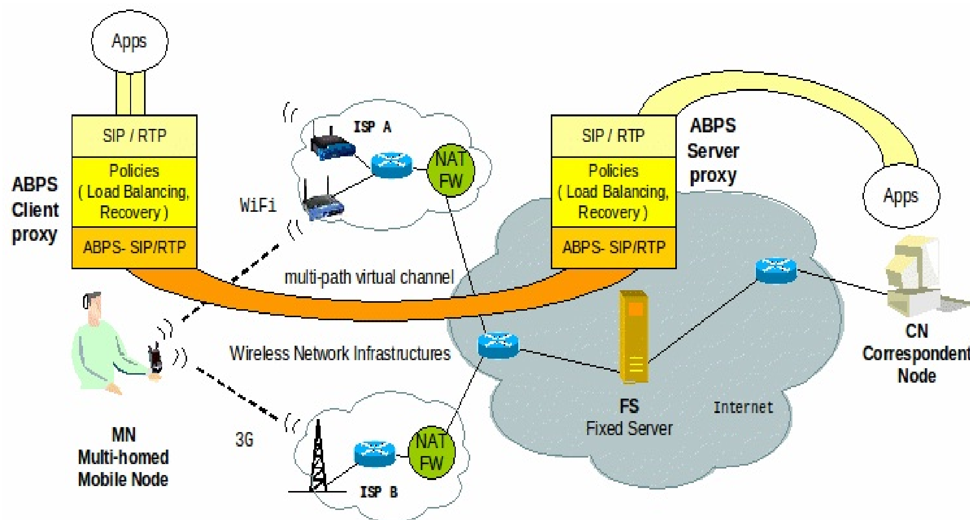

Fig. 1.1: L'architettura ABPS in ambito SIP/RTP

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

- **Servizio di continuità di comunicazione esterno**
  In figura 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.
  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 stiamo per 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 stesa 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 dl 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 QoE 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 dalle 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 ambedue i casi, anziché dovere 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.

Estensioni ABPS al protocollo SIP

Un'estensione al protocollo SIP è prevista per garantire autenticazione, integrità del messaggio e comunicazioni sicure tra ABPS client e server. Come accennato, attraverso la chiave condivisa scambiata durante la configurazione off-line, viene creata ogni volta una chiave one-session, condivisa e temporanea.

Ogni messaggio SIP viene crittografato utilizzando una funzione HMAC e la chiave one-session, generando così un *fingerprint* del messaggio. Viene aggiunto così ad ogni messaggio SIP un header di 36 bytes contenente:

- **UID** del sender, per identificarlo univocamente;
- **numero di sequenza**, per prevenire eventuali *replay attack*;
- **fingerprint** del messaggio stesso (con l'header addizionale senza il fingerprint stesso).

Grazie a questo header, un eventuale *attacker* (in senso crittografico) non può inviare messaggi all'end del destinatario e nemmeno modificare messaggi messaggi intercettati, dato che non possiede la chiave one-session stessa.

Estensioni ABPS al protocollo RTP

Un'altra estensione di sicurezza viene effettuata anche agendo sui datagram RTP che viaggiano nei diversi flussi tra MN e FS. Si utilizza il protocollo SRTP (*Secure Real-Time Protocol*) che aggiunge un header di autenticazione al datagram RTP usando una chiave di crittazione associata al flusso RTP cui appartiene il pacchetto. Per essere generata occorre, secondo SRTP, una

fase preliminare in cui si utilizza un protocollo di *key-agreement*: tra i vari possibili è stato scelto ZRTP per le sue prestazioni.

Questo protocollo inizia ad operare al momento dell'invio del primo messaggio di INVITE tra i due end: ZRTP calcola un valore hash e lo inserisce nel corpo SDP del messaggio SIP che descrive il flusso RTP. Onde evitare che un attacker si sostituisca al sender, si utilizza l'estensione ABPS-SIP per garantire autenticazione.

1.2.3 RWM-PC - Robust Wireless Multi-Path Channel

Accenniamo ad un'implementazione *middleware* costruita sopra al *kernel Linux* (v. 2.6.27.4) che sfrutta l'architettura ABPS, soddisfando le richieste di QoS per applicazioni VoIP secondo le linee guida ITU-T G.1010, ottenendo un ritardo di trasmissione inferiore a 150 ms ed una perdita di pacchetti inferiore al 3%. Il tutto potendo inviare ogni datagram attraverso NIC differenti. Questo meccanismo *cross-layer* è formato da tre componenti principali:

- **Monitor**: gestisce le interfacce di rete verificando la loro disponibilità ed eventuali problemi dovuti a disattivazione o degradazione del segnale.

- **TED - Transmission Error Detector**: un meccanismo cross-layer che verifica le perdite dei pacchetti sulla rete wireless, ed invia notifiche al componente ULP.

- **ULP - UDP Load Balancer**: riceve notifiche dal Monitor e dal TED e si occupa di inviare i pacchetti all'altro end della comunicazione, implementando politiche di reinvio di pacchetti persi, eventualmente tramite un'interfaccia di rete diversa.

Non approfondiamo oltre i meccanismi di questo sistema in quanto sebbene si ponga scopi simili a quelli del nostro progetto è stata utilizzata un'architettura proprietaria profondamente diversa, in cui non è possibile utilizzare le stesse soluzioni implementative (come ad es. moduli aggiuntivi al kernel nel caso del TED).

Nel prossimo capitolo esploreremo alcuni meccanismi alternativi specifici per l'ambiente BlackBerry individuati per risolvere alcuni dei problemi citati, ma ora ci occuperemo di offrire una visione completa della piattaforma su cui si è svolto il lavoro.

1.3 La piattaforma BlackBerry®

1.3.1 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(IEEE802.11a/b/g/n),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 di 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*) [12] per creare applicazioni *Java-based* eseguibili sui dispositivi.

La peculiarità principale dei dispositivi BlackBerry risiede nella particolare architettura di rete che offre servizi rivolti sia ad aziende che a privati. Prima di analizzarla illustriamo un quadro generale delle API (release 6.0.0), nonché degli applicativi software messi a disposizione degli sviluppatori software da RIM per creare e testare le applicazioni.

1.3.2 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*) [13] (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, nonché 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, 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, nonché 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 che descriveremo successivamente, 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.

1.3.3 L'architettura di rete BlackBerry

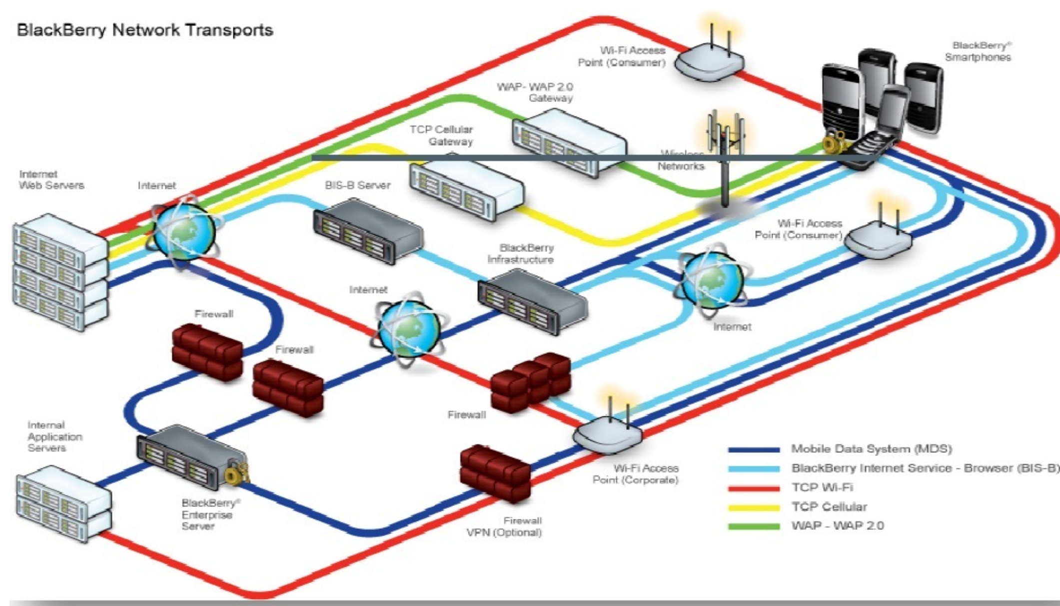

Fig. 1.2: Schema illustrato dell'architettura di rete BlackBerry

Come accennato in precedenza, l'architettura di rete BlackBerry è molto particolare ed offre una serie di servizi differenziati per esigenze professionali. Vedremo ora quali sono i servizi offerti dai diversi sistemi di connessione di rete, seguendo i diversi percorsi in figura.

MDS - Mobile Data System

Il BlackBerry MDS, evidenziato in blu nella figura, è il sistema che permette l'accesso al BES (*BlackBerry Enterprise Server*) illustrato. Si tratta di uno

dei servizi più importanti messi a disposizione da BlackBerry, che consente ad un'azienda di sincronizzare i propri server di posta o di applicazioni (in figura *Internal Application Servers*) con i cellulari in dotazione ai dipendenti, i quali possono ricevere nella propria casella di posta le email tramite un canale wireless sicuro.

Oltre alla posta elettronica è possibile sincronizzare i dispositivi mobili con dati PIM personali e modifiche a calendari, attività, rubriche.

Il MDS è il canale utilizzato dal device per collegarsi al BES: nella prima parte, attraverso una qualsiasi connessione ad internet 3G o WiFi, si connette alla *BlackBerry Infrastructure*, anche conosciuta come NOC (*Network Operation Center*). Da qui i dati vengono inviati al BES che è in genere protetto dall'azienda tramite un firewall.

Tutta la comunicazione dal device al BES attraverso il MDS è protetta mediante cifrari a chiave privata come Triple-DES e AES. MDS è anche un proxy HTTP e TCP sicuro che permette ad applicazioni *third party* in Java ed al Browser BlackBerry di comunicare con le applicazioni ed i web server aziendali.

Si noti che, una volta richiesta la connessione al server BES, il device seleziona automaticamente la rete wireless sulla quale fare viaggiare i dati, in genere quella che richiede il costo minore in termini di consumo energetico, ovvero la rete WiFi. Il passaggio attraverso il NOC, come si può vedere in figura, è evitabile utilizzando un access point aziendale, ed eventualmente una rete VPN.

BIS - BlackBerry Internet Service

Se il servizio BES è rivolto a grandi aziende, per utenti *consumer* o piccoli gruppi RIM mette a disposizione il BIS, che consente di effettuare una sincronizzazione delle email senza un apposito server (e quindi senza i servizi annessi), ma facendo svolgere lo stesso ruolo ad un operatore telefonico che, previa registrazione dell'utente, accede tramite IMAP o POP3 alla *user mailbox* ed invia in modalità *push* le email ricevute al dispositivo BlackBerry dell'utente.

Possiamo vedere in figura che il percorso del BIS (in azzurro) è analogo a quello del MDS, ma senza l'accesso al BES, ed a server di applicazioni aziendali. Il BIS risulta essere un servizio limitato, in quanto non offre le funzionalità di sicurezza del BES e di solito nemmeno la possibilità di sincronizzare altro se non email, dati i limiti dei protocolli POP3 e IMAP usati dall'operatore. Come per il MDS, la connessione wireless per le connessioni BIS viene selezionata dal dispositivo, dando la precedenza all'uso di reti WiFi.

Connessioni wireless WAP e TCP dirette

Per quel che concerne le connessioni WAP (percorso di colore verde) la gestione è dipendente dal *carrier* di rete che trasmette i dati dai server Web al dispositivo senza utilizzare infrastrutture o servizi RIM e lo stesso si può dire delle connessioni TCP Direct (percorso di colore giallo).

Per avere prestazioni migliori, conviene utilizzare connessioni TCP tramite reti WiFi (percorso di colore rosso), accedendo ad Internet da un access point qualunque o anche da uno aziendale per comunicare con applicazioni predisposte.

Connessioni UDP

Come si può vedere, nell'architettura di rete BlackBerry il protocollo UDP non viene considerato: questo perché, data la sua natura connection-less, 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. Nel prossimo capitolo vedremo anzitutto descritta proprio questa prima fase di ricerca e sviluppo.

Capitolo 2

Livello di trasporto su Blackberry

Descriviamo in questo breve capitolo la prima fase di studio e sviluppo del software in ambiente Blackerry.

Si è cercato di effettuare ricerca e testing sui servizi offerti dalle API BlackBerry per verificare il funzionamento delle comunicazioni a livello di trasporto (in particolare usando il protocollo UDP), in modo da poterle utilizzare come base di un'applicazione VoIP.

Si è inoltre ricercato un modo per poter supportare, almeno in parte, un'eventuale implementazione del modello architetturale ABPS, limitandoci alla parte riguardante la trasmissione dei dati *transport level* e trascurando per il momento le modifiche ai livelli superiori.

2.1 Connessioni UDP su Blackberry

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 specificare una condizione di timeout.

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 vengano eseguano queste operazioni di rete in thread diversi dal *main event thread*.

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 importante 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 svolga (almeno parzialmente) i compiti del componente TED nell'implementazione del modello ABPS RWM-PC, in modo da poter conoscere lo status dei pacchetti inviati e mettere 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) 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.

2.2 Monitoraggio di reti wireless su BlackBerry

Lanciando uno sguardo sul sistema ABPS, si è cercato un modo per poter supportare pienamente il controllo degli eventi riguardanti le reti wireless. Proprio come nell'applicazione Monitor nell'implementazione RWM-PC si vuole verificare se ci siano problemi riguardanti una interfaccia di rete, in modo da poter cambiare tempestivamente la NIC selezionata per la trasmissione dati, anche qui si vuole fare lo stesso, facendo in modo di selezionare una connessione creata anziché un'altra per la trasmissione dei pacchetti.

Come visto prima, è possibile aprire connessioni WiFi oppure utilizzando le altre reti supportate dal dispositivo, in genere 3GPP o CDMA. Possono essere aperte molteplici connessioni ad un altro end tramite reti diverse senza controindicazioni, ma l'invio e la ricezione dovrebbero essere svolte su quella opportuna in base ai criteri descritti in precedenza.

A questo fine abbiamo utilizzato alcuni strumenti delle API BlackBerry,

presenti nel package *net.rim.device.api.system*.

Le classi e le interfacce in esso ci 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 Testing e problematiche di simulazione

Le funzionalità descritte sono state testate creando applicazioni Blackberry apposite con le API v. 4.6/4.7/5.0. Non sono stati introdotte nelle nuove

versioni strumenti più sofisticati degni di nota. Le applicazioni di prova create si occupano di monitorare la rete ed inviare poacchetti, utilizzando una connessione anziché un'altra a seconda della disponibilità.
Per effettuare il testing abbiamo utilizzato diversi simulatori di dispositivi, tra cui i seguenti modelli:

- BlackBerry Bold 9000 (OS v.4.6);
- BlackBerry Curve 8900 (OS v.4.6);
- BlackBerry Bold 9700 (OS v.4.7).

Oltre alla semplice esecuzione delle applicazioni, al fine di ricreare le stesse difficoltà rinvenibili con reali apparecchi, abbiamo usato le funzionalità di simulazione delle proprietà di rete. Grazie ad esse è possibile modificare a propria scelta le impostazioni riguardanti parametri come la copertura di rete, la potenza del segnale in dBm (*decibel milliwatt*) e la possibilità di *roaming*; tutto questo per provocare, dal punto di vista del simulatore, disconnessioni improvvise o perdite di *signal strength*.

In realtà queste funzionalità non sono state pensate per essere usate a runtime di un'applicazione, ma si sono rivelate comunque sufficientemente funzionanti, mettendo l'applicazione in condizione di gestire disconnessioni, perdite di segnale, e riconnessioni improvvise durante la sessione di trasmissione e ricezione dei dati.

Naturalmente il comportamento dei simulatori è risultato non realistico in alcuni casi: ad esempio è accaduto che dopo alcuni secondi di disconnessione manuale dalla rete (simulata) WiFi, il simulatore segnalasse automaticamente una riconnessione ad un access point, a prescindere dal controllo delle impostazioni. Questo perché, come accennato prima, le funzionalità non sono fatte per modifiche del sistema a runtime di un'applicazione.

Il discorso andrebbe inoltre ampliato al testing dell'intera applicazione: sebbene il simulatore sia uno strumento utile, le prove effettive attraverso dispositivi fisici sono indispensabili per avere un quadro completo di tutti problemi da risolvere, soprattutto in ambito di *network applications*. Non potendo invece disporre di apparecchi reali, abbiamo dovuto utilizzare connessioni UDP associate soltanto all'indirizzo di *localhost*. Questo perché il software per simulare il sistema di rete BlackBerry, MDS simulator, permette alle applicazioni ed al BlackBerry Browser di simulare solo una connessione MDS mediante la connessione ad Internet presente sull'host in cui è installato il software. Una reale connessione MDS opera invece attraverso TCP ed infrastrutture di rete specifiche di RIM e descritte nel primo capitolo.

Proseguendo sul nostro percorso nell'implementazione di un'applicazione

VoIP abbiamo fatto uso delle tecniche descritte per gestire la trasmissione *transport level* dei pacchetti SIP ed RTP, integrandole in un sistema per l'implementazione di applicazioni *SIP based*: *MjSip*. Nel prossimo capitolo vedremo di cosa si tratta, quali servizi offre e i passi compiuti per poterlo utilizzare sulla piattaforma BlackBerry.

Capitolo 3

Implementazione di SIP su BlackBerry

La fase di lavoro compiuto che ci apprestiamo ad analizzare è indubbiamente quella più estesa ed importante del progetto.

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.

Scopriremo come abbiamo potuto sopperire a questa mancanza e descriveremo proprietà e funzionalità del sistema software utilizzato e della sua implementazione sulla piattaforma BlackBerry.

3.1 Requisiti

Abbiamo già visto nel capitolo di background teorico come per lo sviluppo di un'applicazione VoIP la scelta migliore sia quella di utilizzare come protocollo session level di signaling ed handshaking il SIP, come descritto nella RFC 3261 dello standard IETF.

Esistono numerose implementazioni complete ed open source di questo protocollo, in diversi linguaggi e spesso multipiattaforma (ad es. PjSip). Tali implementazioni vengono definite *stack SIP*, e si tratta in generale di librerie scritte in qualche linguaggio (in genere C, C++ o Java). Per noi è stato inoltre doveroso cercare un'implementazione che fosse quanto più

compatibile con la piattaforma BlackBerry.
Riassumendo, occorreva uno stack SIP che fosse:

- *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, di cui ci apprestiamo ad analizzare le caratteristiche e l'architettura.

3.2 Perché MjSip?

MjSip [14] è 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.

MjSip è stato utilizzato nella sua ultima release 1.6. Ora vediamone una panoramica strutturale.

3.3 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*.

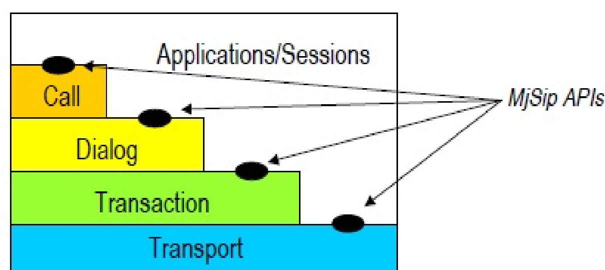

Fig. 3.1: L'architettura a livelli MJSIP

3.3.1 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, nonché 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 una 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, dopodiché 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).

3.3.2 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 piò 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*.

3.3.3 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.

3.3.4 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*.

3.3.5 API aggiuntive

Altre API di MjSip sono usate per supportare direttamente le entità di Call, Dialog, Transaction e Transport viste sopra, e sono direttamente incluse negli stessi package dell'entità supportata.

Altri package importanti in ambito strettamente SIP (dominio *org.zoolu.sip*) sono:

- **Message**, per la creazione dei messaggi SIP;
- **header**, per la creazione degli header dei messaggi;
- **sdp**, per l'inclusione degli header SDP;
- **address**, per la gestione dei nomi e degli indrizzi SIP.

Altri package del dominio *local* gestiscono le funzionalità del sistema che non riguardano strettamente SIP, come le connessioni a livello di trasporto ed il supporto a RTP, di cui tratteremo più avanti.

3.3.6 Il modello Provider→Listener

MjSip è come detto una struttura a livelli ed in quanto tale ogni livello usufruisce di servizi del livello sottostante (ad esempio, un'entità Dialog utilizza più entità transaction).

La comunicazione con i livelli sottostanti avviene utilizzando il modello Provider→Listener di Java in cui una classe che intenda interagire con entità del livello sottostante (classi Provider) deve estendere le funzionalità di Listener di quelle interfacce. Se la classe soddisfa questo requisito, può aggiungersi come Listener di eventi di uno o più Provider e catturare gli eventi generati dai Provider stessi attraverso i metodi ereditati dai Listener (*callback methods*).

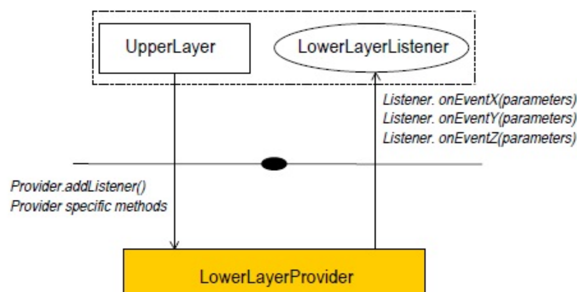

Fig. 3.2: Il modello Provider→Listener per le API MjSip

Per fare un esempio concreto, prendiamo la classe SipProvider: essa implementa l'interfaccia *TransportListener*, che contiene il metodo *onReceivedMessage*. Si tratta di un callback method che viene richiamato quando al livello inferiore di Transport viene ricevuto un messaggio sul socket. Ereditandolo, SipProvider riceve notifica dell'evento di ricezione di un messaggio ed esegue in quel momento le operazioni dettate dall'implementazione del metodo, come ad esempio l'analisi del messaggio per capire di che tipo è.
Questa modello offre una grande flessibilità a MjSip, consentendo di:

- Modificare in modo semplice il comportamento delle entità al verificarsi di un dato evento, reimplementando soltanto i metodi di callback associati;
- Aggiungere nuovi eventi corrispondenti a funzionalità estese o aggiuntive a MjSip.

Adesso che abbiamo descritto la struttura di MjSip, possiamo prendere in considerazione il lavoro svolto per renderlo compatibile con i dispositivi BlackBerry.

3.4 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* affinché 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.

3.4.1 Modifiche a livello di trasporto

Descrivendo MjSip a livello di trasporto, abbiamo accennato al fatto che utilizzava per creare connessioni UDP (o TCP) e per gestirle, utilizzava 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 introducendo sostitutivamente l'uso delle connessioni socket messe a disposizione dalle API BlackBerry. Abbiamo perciò sfruttato le conoscenze acquisite studiando l'implementazione del protocollo UDP su Blackberry, ma abbiamo allo stesso tempo anche adoperato 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.

3.4.2 Test e verifica di operazioni base

A questo punto, per testare le modifiche, abbiamo effettuato alcune operazioni SIP di base eseguite da uno UA (*user agent*) da noi creato e da eseguire come applicazione BlackBerry sul simulatore. Per prima cosa, l'entità SipProvider è stata istanziata nello UA in modo che comunicasse attraverso una connessione UDP su localhost alla porta 5060, di default per il signaling SIP a livello di trasporto.

Abbiamo cercato di effettuare una SIP request di registrazione (REGISTER) tramite la quale uno user può essere localizzato ad un certo indirizzo SIP (valido per il dominio di registrazione) per un certo periodo. Per fare ciò lo user agent sfrutta *RegisterAgent*,entità di MjSip che permette allo user agent di inviare una richiesta di questo tipo, prendendo in input l'istanza di SipProvider ed una di *UserAgentProfile*. Quest'ultima classe rappresenta un'entità attraverso la quale poter impostare i dati dell'utente. Contiene perciò campi come l'URL del contatto SIP, lo username, il realm (una stringa univoca per il dominio) ed altri ancora per definirne il comportamento.
La richiesta generata da MjSip risulta:

```
REGISTER sip:127.0.0.1:5060 SIP/2.0

Via: SIP/2.0/UDP 127.0.0.1:8323;rport;branch=z9hG4bK74689
```

```
Max-Forwards: 70

To: <sip:bbclient@127.0.0.1:5060>

From: <sip:bbclient@127.0.0.1:5060>;tag=z9hG4bK43050094

Call-ID: 619805312178@127.0.0.1

CSeq: 1 REGISTER

Contact: <sip:bbclient@127.0.0.1>

User-Agent: mjsip stack 1.6

Content-Length: 0

Expires: 5000
```

Expires è un header che indica il tempo di validità della registrazione in secondi.

La richiesta illustrata sopra viene ricevuta da un *Registrar server* da noi simulato in locale da un entità di MjSip, chiamata *Registrar*. Essa, ricevuta la request, controlla che provenga effettivamente dall'interno del proprio dominio (localhost in questo caso) ed aggiorna un *database* interno, rappresentato da MjSip come un file di estensione .db. Successivamente invia una SIP response nel caso sia andato tutto bene con codice di risposta 200 (*Success*). Sono stati risolti a tal proposito alcuni bug minori, che impedivano ad esempio la corretta identificazione dell'host del sender in alcuni casi.

Mentre lo UA da noi creato è stato avviato attraverso il simulatore BlackBerry, il Registrar è stato avviato come applicazione Java.

Le impostazioni di base sono state gestite direttamente nell'applicazione per lo user agent, mentre per il Registrar Server si è usato un file di configurazione testuale apposito, come da specifiche MjSip.

Una volta testata la creazione e l'invio di messaggi SIP REGISTER, si è effettuata una prova analoga utilizzando lo stesso user agent e l'entità *Proxy* simulante un Proxy *stateless* che si limita a veicolare request e response tra UA diversi senza mantenere lo stato della transazione. In sostanza si tratta di un server che non fa uso del *transaction level* di MjSip.

Per fare questo nella nostra applicazione abbiamo fatto uso di un'altra entità detta *MessageAgent* che consente di inviare messaggi di tipo SIP MESSAGE contenenti nel body del messaggio comunicazioni testuali. Questo test è stato eseguito al solo scopo di controllare ulteriormente la creazione di messaggi SIP di tipo differente e le funzionalità offerte dalle implementazioni MjSip degli user agent.

Inviando dall'applicazione un messaggio di testo destinato ad un'altro contatto SIP è stato generato un messaggio simile a quello di tipo REGISTER, ma senza l'header Expires e con *Subject* e *Content-Type* riferiti al messaggio inviato.
Il proxy è stato in grado di:

- ricevere la request dal nostro UA;
- verificare che fosse per un contatto del proprio dominio;
- leggere nel database (simulato dal file .db) se il contatto dell'utente fosse registrato, e l'*expire time* non ancora scaduto (in caso contrario ha correttamente inviato una SIP response all'UA con codice 404 - *Not Found*);
- effettuare il forward del messaggio all'host dell'utente destinatario.

Il lavoro svolto finora ha comportato modifiche al codice sorgente di MjSip, al fine di:

- capire e controllarne il comportamento in modalità di *debug*, utilizzando la console di output del simulatore BlackBerry;
- risolvere una serie di bug riguardanti fra gli altri l'utilizzo di parametri predefiniti o estratti da file di configurazione, anziché impostati dall'applicazione;
- eliminare comandi utilizzati da MjSip per il *logging* di eventi su file di testo, inutilizzabili con le API BlackBerry.

3.4.3 Handshake per le chiamate vocali

Finora abbiamo visto una comunicazione da parte di uno user agent ed un entità server MjSip. Il passo seguente è stato quello di mettere in contatto due diversi user agent, mettendoli entrambi in condizione di accordarsi su una serie di parametri da stabilire una chiamata vocale.

Prima di tutto, abbiamo dovuto verificare il corretto funzionamento di due simulatori BlackBerry operanti simultaneamente.

Nella pratica, vista anche la necessità di utilizzare connessioni socket soltanto in locale, abbiamo dovuto disporre di due applicazioni BlackBerry in esecuzione con due istanze d'esecuzione diverse,in modo da poter simulare in locale il funzionamento di due diversi device in collegamento attraverso il nostro UA.

Le applicazioni eseguite su ogni istanza del simulatore risultano identiche, eccetto che per il comportamento dello UA, che in un'applicazione effettua la chiamata all'utente destinatario dotato di un contatto SIP differente, mentre l'altra rimane in attesa di questa chiamata entrante e la accetta automaticamente.

In entrambe le applicazioni, gli UA utilizzano a questo scopo l'entità *UserAgent*: essa si può istanziare passando come parametri un'istanza di SipProvider ed una di *UserAgentProfile*, come per gli agent di registrazione e proxy visti in precedenza.
L'entità UserAgent possiede i seguenti metodi:

- **call(contact)**, per invitare il contatto dato ad una sessione vocale;
- **hangup()**, per terminare la chiamata;
- **listen()**, per accettare una chiamata.

Con questi semplici metodi e settando il profilo utente dello UA in attesa della chiamata in modo che risponda automaticamente, i nostri user agent possono entrare in una sessione di handshake per la comunicazione vocale.

Precisiamo che, seguendo il modello Provider→Listener, i nostri user agent diventano anche listener dell'entità UserAgent, con la possibilità di catturarne gli eventi e di implementare i metodi di callback.

Sebbene i metodi siano molto semplici all'apparenza, è bene ricordare che si tratta di astrazioni di ciò che accade ai livelli inferiori dello stack MjSip: in particolare, il metodo call avvia almeno una sessione di Dialog, che descrive l'handshake tra i due user agent prima dell'effettivo scambio dei pacchetti contenenti il traffico voce. Vediamo le transaction coinvolte durante questa sessione singola di Dialog.

La prima transazione all'opera viene iniziata da un'entità *InviteTransactionClient* istanziata sullo UA che richiede la chiamata (lo chiameremo *sender*) inviando una request di tipo INVITE. L'altro UA (detto *receiver*) rimane in attesa con il metodo hangup() e tramite l'entità *InviteTransactionServer* risponde temporaneamente con un messaggio TRYING (codice 100), a cui seguono una serie di altri messaggi RINGING (codice 180), sempre da parte del receiver. Questi messaggi sono inviati ripetutamente fino a che il receiver non accetta la chiamata su input umano, o nel nostro caso, con una risposta automatica dopo un certo numero di secondi.

Quando il receiver accetta la chiamata, invia una risposta definitiva al sender con codice 200 (OK) e comprensiva degli header SDP che descrivono i parametri della sessione multimediale.

Precisiamo che per potere modificare questi dati occorre istanziare l'entità

SessionDescriptor ed utilizzare il metodo *addMedia* per settare gli header ed eventuali attributi.

A questo punto il receiver crea un *AckTransactionServer* che provvede ad inviare la risposta positiva al sender. InviteTransactionServer entra invece nello status TERMINATED e viene terminata. Il sender che riceve questa risposta invia un ultimo messaggio di ACK al receiver con una nuova transaction creata con l'entità *AckTransactionClient*, non prima di avere terminato Invite Transaction Client che entrato anch'esso nello status TERMINATED. Il messaggio di ACK contiene i dati della sessione SDP che possono essere semplicemente gli stessi inviati dal receiver o meno, a seconda della compatibilità del sender. Una volta che il receiver ha ricevuto questo messaggio AckTransactionServer viene terminata, così come AckTransactionClient dopo l'invio del messaggio ACK.

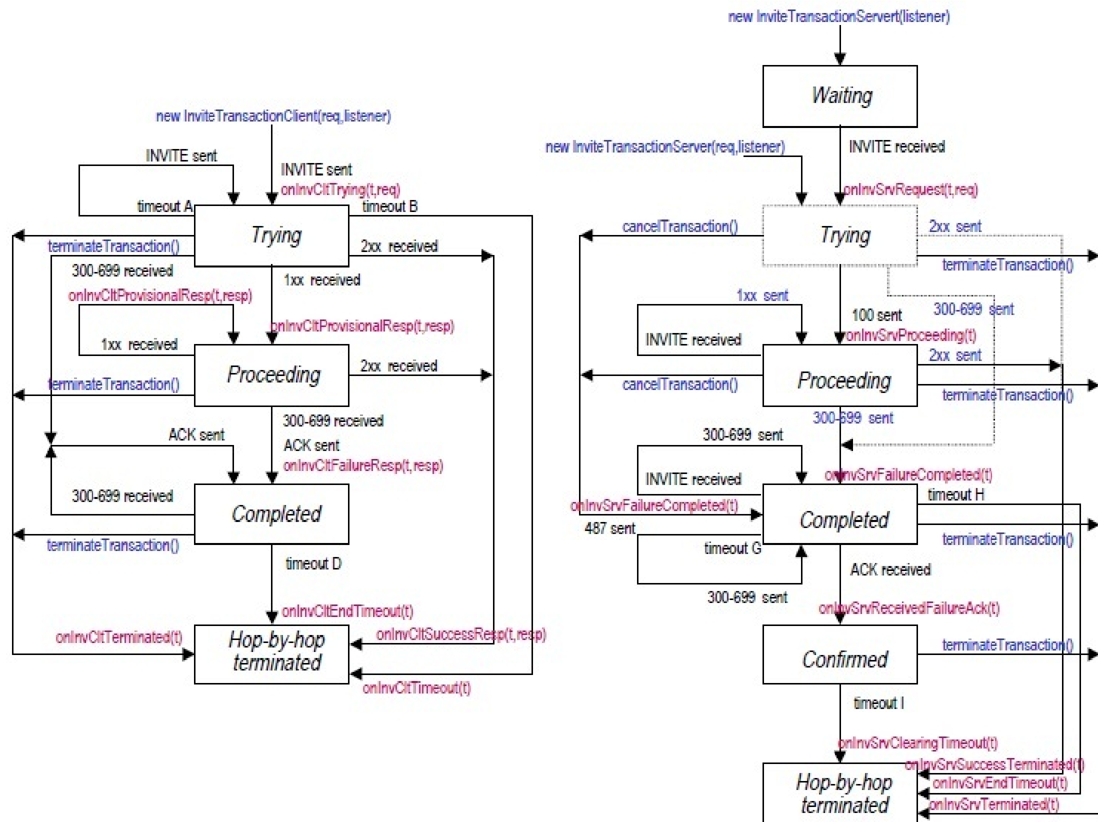

Fig. 3.3: Diagramma degli stati a runtime di InviteTransactionClient ed InviteTransactionServer

Dopo questa fase l'handshake è concluso ed entrambi gli UA possono effettuare lo scambio dei pacchetti audio. Sull'implementazione di questa

parte ci concentreremo più avanti; per ora consideriamo la fase successiva alla comunicazione vocale. In questa fase vengono avviati ancora gli Invite Transaction Client e Server al fine di scambiarsi l'uno con l'altro un messaggio BYE (conclusione di chiamata) in maniera analoga a quanto fatto per le transazioni di INVITE. Fatto ciò, l'intera sessione Dialog può considerarsi terminata.
In figura possiamo osservarne un riepilogo:

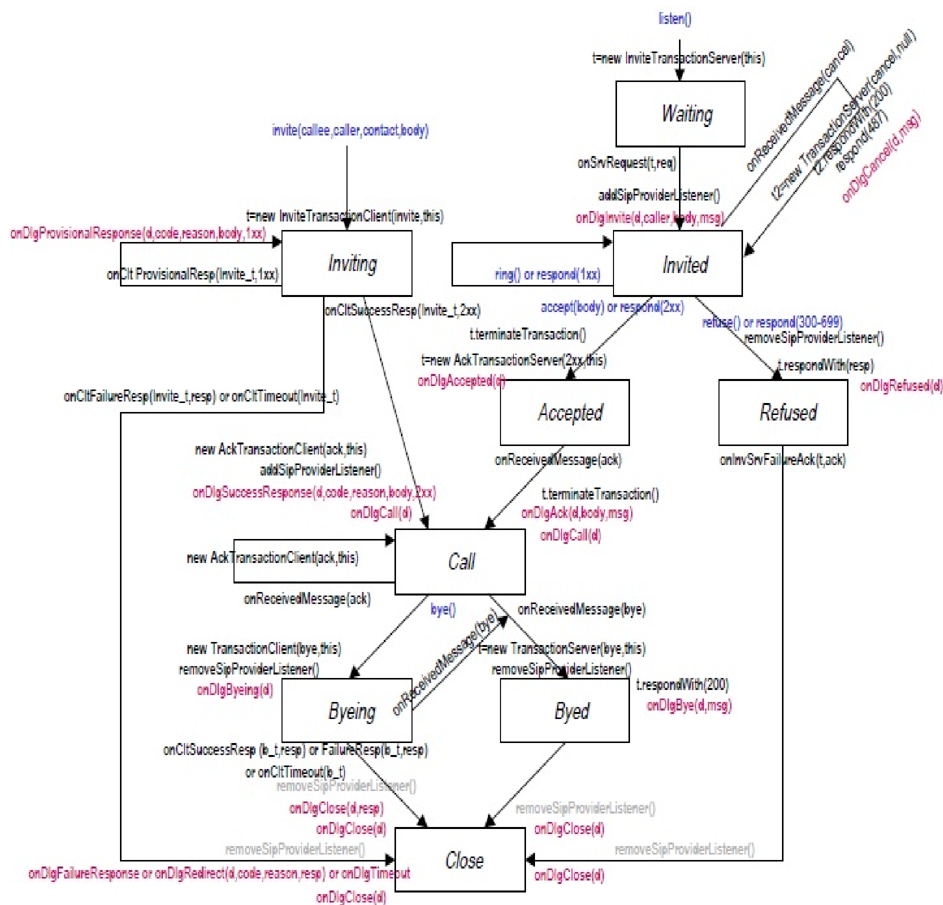

Fig. 3.4: Sessione completa di Invite Dialog

Durante il nostro lavoro non abbiamo trattato casi di chiamate con sessioni Dialog multiple, adottate nelle implementazioni comuni, ma da noi per il momento trascurate.

Quello che avviene dopo la fine dell'handshake e prima dello scambio dei messaggi conclusivi concerne la parte di *streaming* dei dati voce che devono essere scambiati tra i due user agent, tenendo conto dei parametri accordati ed inseriti in precedenza negli header SDP.

Nel prossimo capitolo vedremo le attività svolte per implementare questa parte, partendo dallo studio delle funzionalità di MjSip e tenendo conto della compatibilità con la piattaforma BlackBerry.

Capitolo 4

Audio streaming su BlackBerry

In questo capitolo introduciamo il tassello mancante alla realizzazione di un prototipo di applicazione VoIP.

Siamo riusciti ad implementare MjSip ed a testarne il funzionamento per la gestione di sessioni di registrazione, invio di messaggi ed infine handshake per comunicazioni vocali.

Ciò che manca è allora un servizio che consenta di effettuare registrazioni audio dal dispositivo, incapsularle ed inviarle sotto forma di pacchetti dati ad un'altro utente, che attraverso la stessa applicazione potrà ricevere questi pacchetti, estrarre l'audio contenuto e riprodurlo. Nel seguito descriveremo le parziali soluzioni ottenute al problema, cominciando dall'analisi dalle proposte incluse in MjSip che ci apprestiamo ad illustrare.

4.1 Le soluzioni proposte da MjSip

Lo stack SIP che abbiamo utilizzato e portato parzialmente su BlackBerry viene incontro alle necessità degli sviluppatori fornendo alcune proposte per la gestione dei flussi audio in *streaming*.

La prima soluzione proposta si affida ad un applicazione esterna open source, RAT (*Robust Audio Tool*). Si tratta di un'applicazione per la comunicazione in streaming e per conferenze audio basata su standard IETF che fa uso del protocollo UDP (con al di sopra RTP) per il trasporto dei dati. L'utilizzo di tale applicazione è stato scartato a priori in quanto *C based* e non compatibile con il nostro sistema operativo BlackBerry.

Un'altra soluzione fornita consiste invece nell'utilizzo delle librerie JMF (*Java Media Framework*) [15] che permettono di creare sofisticate applicazioni (o *applet*) audio e video cross platform. Purtroppo queste librerie risultano di fatto inutilizzabili a causa del fatto che le API BlackBerry non le suppor-

tano. Nel corso dello svolgimento del progetto è stata anche valutata la plausibilità di un'eventuale porting di queste librerie o di una parte di esse; data però la quantità di servizi ed il numero di librerie Java richieste, un'implementazione attraverso le sole API di tipo multimediale su BlackBerry non sarebbe stata possibile.

L'ultima proposta di MjSip rimasta da vedere è in realtà un abbozzo di implementazione nativa *pure Java* per gestire i flussi multimediali attraverso l'utilizzo di RTP. Si tratta di un semplice servizio di invio e ricezione di datagram RTP, ma per quanto incompleto ed inutilizzabile nel caso di streaming in *real-time*, ha costituito l'impalcatura su cui costruire un sistema più complesso e conforme ai nostri obiettivi.

4.2 MjSip: stato dell'arte per la comunicazione via RTP

MjSip mette a disposizione alcune classi per permettere la registrazione di 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 pre registrato.

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 descritto nel capitolo precedente.

RtpStreamSender si accerta di avere ricevuto in input un socket UDP ed uno stream preregistrato di dati, poi questo flusso viene suddiviso in molteplici *frame* di dimensioni predefinita ed ogni singolo frame incapsulato come payload di datagram RTP. 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 come stream registrato da trasmettere via RTP, un flusso audio con codifica PCM (*Pulse Code Modulation*) ed algoritmo di compressione μ-law. La dimensione dei frame in cui viene suddiviso tale stream è fissata a 500 bytes.

RtpStreamSender incapsula così ogni frame dello stream dato in input e setta gli header timestamp, sequence number e payload length, servendosi poi 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 una sempliceoperazione 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. è stato quindi nostro dovere provvedere a questa necessità.
Altre modifiche sono state indispensabili da parte nostra per:

- gestire flussi di dati in real-time streaming, come i dati voce anziché flussi pre-registrati;
- utilizzare connessioni UDP compatibili con le API BlackBerry al posto di quelli non bloccanti usati nello stack;
- gestire eventualmente codec audio differenti da quello PCM usato in MjSip.

4.3 Streaming real time per comunicazione voce su BlackBerry

In questa sezione consideriamo il lavoro svolto sulle entità RtpStreamSender e RtpStreamReceiver: sono state implementate, secondo i nostri obiettivi, ognuna su uno UA diverso e ci riferiremo allo UA client, o sender, intendendo quello che inizia la trasmissione audio ed invia i datagram RTP con i dati voce, mentre lo UA server, o receiver, sarà quello che riceve i dati e li riproduce. Si tenga presente che si tratta degli stessi UA che consideriamo gi accordati sui parametri di conversazione attraverso le operazioni di signaling SIP e SDP descritte nel capitolo precedente. Per semplicità, è stata quindi presa in considerazione nel nostro lavoro 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 fatta è stata quella di utilizzare i socket non bloccanti forniti dalle API BlackBerry, al posto di quelli usati in MjSip. Ovviamente la connessione UDP utilizzata per i datagram RTP voce, come da specifiche, non è la stessa adottata per l'handshake tra i due user agent, ma una specifica instaurata al termine di questa fase.

Per la comunicazione vocale è servito anzitutto un servizio per la registrazione vocale, nel nostro caso l'interfaccia *Player* appartenente alla J2ME e

presente nelle API BlackBerry. Mediante essa è possibile sia riprodurre flussi audio (anche su dispositivi remoti) sia registrare gli input vocali, nel nostro caso provenienti dal microfono del dispositivo BlackBerry.
Per controllare il Player durante il suo ciclo di esecuzione si possono individuare cinque stati diversi:

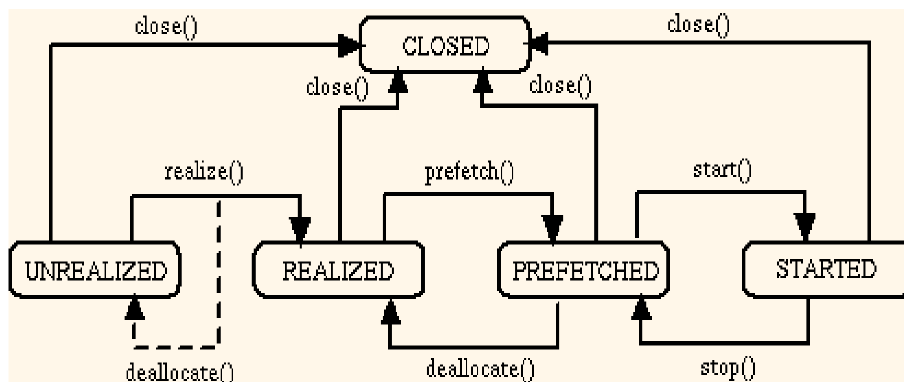

Fig. 4.1: Diagramma di transizione degli stati di Player

- **UNREALIZED**: lo stato iniziale alla creazione del Player, quando non ha informazioni per gestire le risorse con il quale 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 si utilizza un'oggetto *RemoteControl* istanziato a partire dal Player stesso, fornendogli uno stream vuoto da riempire con i dati audio. RemoteControl deve essere avviato assieme al Player per effettuare la registrazione, e concluso con il metodo *commit()* prima della chiusura del Player.

Per il resto le operazioni compiute da RtpStreamSender sono le stesse descritte in precedenza per l'incapsulamento dei dati audio, con la differenza

che lo stream da cui vengono letti i frame e successivamente incapsulati in datagram RTP è quello registrato durante l'esecuzione dello user agent.

Una volta che la chiamata è conclusa, ovviamente RtpStreamSender provvede anche ad eseguire le operazioni di chiusura del Player, e di commit di RecordControl. Per RtpStreamReceiver il discorso è più complesso: se nella implementazione di base il flusso viene semplicemente ricostruito (bufferizzando il payload dei datagram ricevuti) e poi riprodotto, nel nostro caso è necessario riprodurre l'audio contenuto in ogni singolo datagram al momento della ricezione per poter godere di una conversazione con una latenza accettabile, e garantire QoS adeguato.

Tale problema è stato risolto soltanto in parte ed è tuttora in fase di sviluppo, ma sono stati individuati gli strumenti chiave per poter raggiungere questo obiettivo.

RtpStreamReceiver si affida sempre all'entità Player precedentemente descritta; a questa però non vengono dati in input direttamente i payload dei datagram RTP in arrivo sul socket. Se così fosse, infatti, 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 è però in grado di gestire in input un'interfaccia *DataSource* che garantisce un'astrazione sui dati che arrivano in streaming da una fonte esterna (ad es. da un server o un'altro user agent nel nostro caso) e fornisce al Player i dati da riprodurre, ma nasconde il modo in cui vengono effettivamente recuperati. 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 pi importanti di DataSource sono *connect()* e *disconnect()* che servono per istanziare un'implementazione di ogni flusso di dati audio da gestire. Questa implementazione riferita ad una interfaccia che rappresenta un flusso audio *SourceStream*. 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.

Veniamo al cuore di questo sistema, che è per l'appunto l'implementazione di SourceStream. Questa interfaccia contiene le segnature di vari metodi

dei quali i più importanti sono *start()* e *close()* per avviare e chiudere lo stream e *read()* per leggere i dati in dal flusso e passarli al Player.

Nella nostra implementazione dell'interfaccia, *RTPSourceStream*, le implementazione dei metodi start() e stop() servono per creare (e chiudere) una connessione RTP, che è l'astrazione di una connessione UDP per i datagram RTP, esattamente come per RtpStreamSender.
Il metodo read ha in input tre parametri:

- un buffer, costituito da un array di bytes per memorizzare i dati che verranno automaticamente inviati al player;
- un offset, per specificare da quale posizione del buffer partire per le operazioni di scrittura;
- la dimensione dei dati in bytes da inserire nel buffer.

Nella nostra implementazione i datagram RTP vengono ricevuti con una chiamata (ovviamente bloccante) al metodo receive() della connessione instaurata precedentemente, dopodiché viene estratto il payload di ogni pacchetto e dopo alcuni controlli per prevenire problemi di *overflow*, ne vengono copiati i dati nel buffer in modo che il Player li possa riprodurre in modo automatico. Vediamo di seguito un diagramma riassuntivo di questo sistema di ricezione e riproduzione di dati streaming:

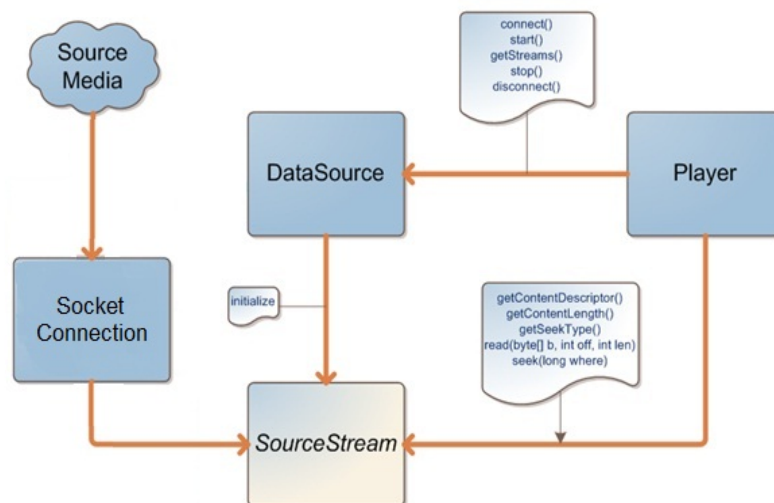

Fig. 4.2: Diagramma dei componenti per trasferimento e riproduzione di stream da fonte remota

4.4 Problemi, limiti di simulazione e TODO

Il testing dell'implementazione presentata è ancora in fase di attuazione; ad ora siamo riusciti ad ottenere una comunicazione vocale one-way da uno UA all'altro per alcuni istanti, ma permangono problemi come la lentezza nella fase di registrazione vocale e di riproduzione dei payload audio contenenti registrazioni di durata nell'ordine delle decine di ms.

Occorre ancora indagare a fondo il reale funzionamento del Player, possibilmente su dispositivi reali. Questo al fine di comprendere se il comportamento del Player in registrazione e riproduzione sia o meno lo stesso anche con i simulatori utilizzati. Sempre riguardo al comportamento del Player, sarebbe utile capire se è in grado di richiedere ulteriori dati quando non ne ha a disposizione a sufficienza e in caso negativo sperimentare tecniche di bufferizzazione e verificarne il costo in termini di delay.

Un altro limite verificato con i simulatori è dato dal fatto che l'unica codifica audio che si è rivelata funzionale con i test finora effettuati ed usata per la registrazione e riproduzione da parte del Player, è stata AMR (*Adaptive Multi-Rate Audio Codec*) [16], con bit rate a 12.2kbps, frequenza di campionamento a 8000Hz. Con questa codifica l'ideale sarebbe ridurre la dimensione dei payload audio nell'intervallo 244-488 bytes (con bit rate predefinito, conterrebbero da 20 a 40 ms di conversazione, come da specifiche di QoS VoIP). AMR è una codifica espressamente pensata per le reti GSM/3G e sarebbe opportuno un testing su dispositivi non simulati ed utilizzando reali reti Wi-Fi.

Una volta risolti i problemi descritti, alcune funzionalità da sviluppare dovrebbero avere la precedenza. Anzitutto è necessario implementare la comunicazione *full duplex* o bidirezionale, utilizzando un solo canale UDP per ricezione e trasmissione dei dati tra gli UA. Potrebbe apparire semplice a prima vista, attivando su ogni UA RtpStreamReceiver e RtpStreamSender. Ma ricordiamo che essi sono in realtà due thread con una connessione UDP bloccante in condivisione ed è necessaria quindi una sincronizzazione per evitare blocchi improvvisi della conversazione.

In secondo luogo è importante sperimentare l'utilizzo di codec audio multipli: il Player su piattaforma BlackBerry supporta registrazione e riproduzione di stream audio con codifiche diverse da AMR, tra cui il già citato PCM. Ricordiamo che se il proposito si rivelasse realizzabile, sarebbe opportuno effettuare alcune modifiche a livello transaction con l'utilizzo delle API MjSip. Nel dettaglio, in fase di handshake SIP si dovrebbe specificare ad ogni esecuzione la codifica audio usata per i pacchetti RTP, mettendo in condizione i due UA di accordarsi su questo parametro.

Le prove dell'applicazione svolte sui simulatori Blackberry hanno richiesto

molto tempo, anche a causa del fatto che il debug con l'utilizzo simultaneo di due diversi simulatori (uno per ogni UA) è assai oneroso, sia in termini di costo computazionale, sia in quanto a tempo necessario per l'avvio di ogni simulatore.

Un'altro ostacolo pratico è stato dovuto al fatto che la modalità specifica di debug, sebbene consenta di continuare l'esecuzione delle applicazioni a dispetto delle eccezioni rilevate, non permette comunque di terminare e riavviare più volte una stessa applicazione durante il *life cycle* del simulatore.

Questo ha implicato inevitabilmente la necessità di chiudere e riavviare l'istanza del simulatore eseguita ogni qualvolta si è incorsi in un errore che comprometteva l'esecuzione dell'applicazione.

Conclusioni e sviluppi futuri

Nel percorso svolto finora abbiamo toccato diversi argomenti, partendo dalle basi teoriche su cui si è fondato il nostro lavoro di ricerca e sviluppo e proseguendo nella descrizione del lavoro di ricerca e sviluppo del software.

Abbiamo mostrato cosa sia la tecnologia VoIP ed i protocolli su cui fa affidamento per i suoi servizi, nonché quali requisiti ed obiettivi occorre raggiungere per garantire QoS nelle applicazioni che ne fanno uso.

Si è descritto il modello di architettura ABPS, esplicando le proposte per l'utilizzo in campo multi-homing. Senza soffermarci in maniera approfondita sulle implementazioni specifiche, abbiamo spiegato le modifiche ai protocolli SIP ed RTP da approntare, gli obiettivi da raggiungere e gli strumenti ausiliari, software e hardware, necessari per la realizzazione di questo modello.

Abbiamo poi approfondito meglio la piattaforma BlackBerry su cui abbiamo lavorato per lo sviluppo di un'applicazione VoIP. Sono state descritte le sue funzionalità, i servizi offerti dai dispositivi mobili e quelli per la creazione ed il testing delle applicazioni.

Successivamente è stata trattata l'implementazione dell'applicazione, partendo dai servizi a livello di trasporto con protocollo UDP e dalle sperimentazioni per un futuro utilizzo del modello ABPS sui dispositivi BlackBerry.

Si è passati quindi al livello di sessione illustrando uno stack SIP, MjSip, nella sua struttura generale e che si è provveduto a portare su piattaforma BlackBerry, modificandolo ed adattandolo relativamente alle API offerte su questa piattaforma proprietaria.

Nella parte finale abbiamo illustrato le attività relative alla sessione di comunicazione vocale, presentando un sistema per l'invio e la ricezione di pacchetti RTP audio compatibile con i dispositivi BlackBerry. Quest'ultima parte è ancora in fase di completamento e necessita di estensioni importanti.

Oltre alle migliorie già suggerite in precedenza, possiamo individuare una serie di ulteriori obiettivi futuri consistenti in:

- Studio di un'implementazione completa di client ABPS (possibilmente integrato nell'applicazione VoIP) con particolare riferimento a tecniche

di monitoraggio dei pacchetti persi o in ritardo, mediante l'utilizzo delle API BlackBerry;

- Implementazione o porting da piattaforme esterne dei servizi di sicurezza utilizzati nel protocollo ABPS da noi trascurati in questo documento, ma necessari per garantire sicurezza ed autenticazione;
- Correzione, modularizzazione e *refactoring* del codice per garantirne stabilità e robustezza, soprattutto nell'ottica di prove accurate con dispositivi BlackBerry;
- Creazione di un'interfaccia grafica completa di uso intuitivo utilizzando le API User Interface di BlackBerry.

Tutto il codice e la documentazione prodotta durante il lavoro sono a disposizione di chiunque intenda collaborare al raggiungimento di questi obiettivi.

Bibliografia

[1] RIM. *Research In Motion*. URL:http://www.rim.com/, 1995.

[2] Vittorio Ghini Giorgia Lodi Fabio Panzieri. *Always Best Packet Switching:the Mobile VoIP Case Study*. University of Bologna, October 2009.

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

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

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

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

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

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

[15] Sun MicroSystems Intel Silicon Graphics. *Java Media Framework*. URL:http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html, 2003.

[16] 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.